A System for the Analysis of Musical Data

Stuart Pullinger

5th August 2010

This thesis is submitted in fulfilment of the requirements of the degree of Doctor of Philosophy

Department of Electronics
& Electrical Engineering

University of Glasgow

Abstract

The role of music analysis is to enlighten our understanding of a piece of music. The role of musical performance analysis is to help us understand how a performer interprets a piece of music. The current work provides a tool which combines music analysis with performance analysis. By combining music and performance analysis in one system new questions can be asked of a piece of music: how is the structure of a piece reflected in the performance and how can the performance enlighten our understanding of the piece's structure?

The current work describes a unified database which can store and present musical score alongside associated performance data and musical analysis. Using a general purpose representation language, Performance Mark-up Language (PML), aspects of performance are recorded and analysed. Data thus acquired from one project is made available to others. Presentation involves high-quality scores suitably annotated with the requested information. Such output is easily and directly accessible to musicians, performance scientists and analysts.

We define a set of data structures and operators which can operate on musical pitch and musical time, and use them to form the basis of a query language for a musical database. The database can store musical information (score, gestural data, etc.). Querying the database results in annotations of the musical score.

The database is capable of storing musical score information and performance data and cross-referencing them. It is equipped with the necessary primitives to execute music-analytical queries, and highlight notes identified from the score and display performance data alongside the score.

Contents

List of Tables

List of Figures

I would like to thank my supervisor, Dr Nick Bailey, and Professor Graham Hair for their invaluable assistance with this thesis. I would also like to acknowledge the help of Carola Böehm, Ingrid Pearson and Margaret McAllister.

I would also like to thank my colleagues at the Centre for Music Technology for their help: Jered Bolton, Alasdair Anderson and Bryony Buck. Special thanks must go to Jennifer MacRitchie for her help in collecting data presented here, Douglas McGilvray for writing his excellent tools and Tom O'Hara for his technical assistance.

Finally, I could not have completed this thesis without the support of my wife, Georgina Humphreys.

Chapter 1

Introduction

The aim of this work is to create a computer system which is capable of combining music and performance analysis. We define music analysis here as understanding a piece of music from it's written form and performance analysis as understanding how a performer interprets a piece of music. It is the intention that a system which is capable of combining music analysis and performance analysis will enable new questions to be asked about music and its performance. It is hoped that new insights can be gained into the way in which a performer reflects the structure of a piece in their performance and how analysis of musical structure can provide insights into musical performance.

This introduction will describe how others have approached using computers for musical research; in particular for music analysis and performance analysis. I will use this work to formulate a set of requirements for a computer system which is capable of music and performance analysis. The combination of music and it's performance in a computer requires the storage and manipulation of data in many different forms: audio, MIDI, video etc. Therefore, an analysis of the approaches to the representation of different musical domains follows which shows how others have approached the problem of music representation. Finally, an outline is given of the remaining thesis.

1.1 What Is Music?

Any investigation should begin with a definition of the object which is under investigation. In the present case, the problem of defining what we mean by music is deceptively difficult. We may feel that we know what we mean by the term 'music' but actually putting that feeling into a concise and accurate definition is problematic. To further complicate matters, any definition of any art form provides the inspiration for an artist to subvert that very definition. One only has to think of Marcel Duchamp's *Fountain* or John Cage's *4'33"* for examples of artwork whose very existence challenged the contemporary definition.

In the Bloomsbury Dictionary of Music [103], a definition that is suitably broad is given, though even this broad definition is qualified with a statement of it's limitations:

> *music* sound organised in space and time. A more specific definition would depend on the definer and the period of history, as the concept of what constitutes music has changed greatly over time. At present the term has a rather broad scope, esp. in its use by avant-garde classical, jazz and rock musicians.

So 'sound organised in space and time' tells us that music is temporal and exists in some sort of space. But what is this space? This is not the 3 dimensions of physical space. This is the space of pitch and harmony.

A simplistic approach would assume that a score is the perfect representation of this space: a graph with time on the horizontal axis and pitch on the vertical axis. But a score also includes performance indications such *allegro*, which are not reflected by the score and are only realised when the piece is performed. So a score does not completely capture a piece of music. But neither does a performance. A performer may interpret a piece in many different ways introducing new notes that were not in the original score. One can think of countless examples in jazz where wildly different interpretations exist of the same piece of music. For example, compare the performances of *Honeysuckle Rose* by Louis Armstrong and Thelonius Monk.

So a score exists as a record of a piece of music but also functions as an *aide-memoir* to a performer. The performance exists as the performer's interpretation of that piece but can change between performances. Audio and other performance data act as a record of the performance. All these artifacts are manifestations of some thing which we could call the musical object. The computer system described in this thesis is, broadly speaking, intended to aid in the understanding of this musical object. More specifically, it aims to aid in the understanding of the *process* of performance by combining two different manifestations of this musical object: the musical score and it's performance. Since a performance involves a performer analysing a piece of music (either from a printed score or from memory) and then providing an interpretation of that analysis through performance, any analysis of this process must combine analysis of the score and of the performance.

The musical score manifested in the current system is a representation of the data contained in a graphical score. The performance can take the form of many different recordings of performance data such as MIDI or pitch-tracking data. By combining these two views it is hoped that new insights into the music and it's performance can be gained and, consequently, a new understanding of part of this musical object.

Implicit in the above is the scope of the current work. It aims to aid in the understanding of music which is written in a score — music which is purely improvised is outside of this scope. Also outside of this scope is music which is not notated with Common Music Notation. Although the system could be extended into this area, it is not attempted in the current work. However, what remains is the vast majority of music performed in the western art music tradition so this is where the scope of the current work lies.

1.2 Computers in Music

We have seen that the musical object is manifested in many different ways when we try to represent it in a computer. Before we establish the requirements for a system to manipulate this data, it is worth looking at how others have approached employing computers to help in the understanding of this musical object.

An attempt to define the scope of the definition of music as it relates to computers is Standard Music Description Language (SMDL)[97]. It aims to provide a description of a standard music format to represent music from several *domains*. It is an application of HyTime [115] hypermedia structure and Standard Generalised Markup Language (SGML)[67]. Four domains are associated with any piece of music:

- **The Logical Domain** is the latent information contained in a piece of music; the information that is encoded in the score.
- **The Visual Domain** is the graphical representation of the logical domain. It describes how the glyphs which represent the notes will be rendered on the page.
- **The Gestural Domain** is the data from a performance.
- **The Analytical Domain** is the theoretical analysis of the piece's content.

The *domains* were further expanded by Selfridge-Field in [113] to include:

- **The Phonological Domain** is the domain of recorded audio artifacts.

Since we are concerned with the application of computers to the understanding of music, it is useful to survey the ways in which computers have been used in these domains of music.

In the logical domain, computers have been used for composition. There are a large number of software packages, both commercial and otherwise, for the composition of music. Commercial software would include sequencing packages such as Cubase [5] and Logic [17] whereas free alternatives exist such as Rosegarden [29] and Muse [19]. Some of the packages support composition through the laying out of graphical scores so they would also fall into the visual domain.

The previously mentioned software packages are aimed primarily at sequencing MIDI data so have an event driven interface: music is created by creating note events and modifiers which operate on them. Another approach used in computer composition is programmatic: a piece is created as a program which is compiled into an audio file. This approach is used in CSound [50], SuperCollider [32] and ChucK [3].

There are also the Max family of programs that allow for music to be programmed and manipulated in real-time such as the free Pure Data [104] and the commercial Max/MSP [14].

In the visual domain there exist several notation packages both commercial and free. Of the commercial packages the most popular must be Sibelius [30] and Finale [8]. Of the free software packages, the most popular would be NoteEdit [23] and Lilypond [15]. Lilypond has a different interface to the graphical point-and-click interface of the other programs as it uses a text file to represent a score which is subsequently processed to produce a pdf file.

Other tools are used for the visualisation of analyses such as the commercial MatLab [18] software and the similar free software Octave [12] combined with the plotting program GnuPlot [13]. One can trace the use of technology to record musical performance data at least as far back as 1938 when Seashore [112] used a kind of modified player piano to record the durations of performers' keypresses. This technique is analogous to using MIDI data today.

One could consider MIDI data as gestural data since its original purpose was to transport and store piano keyboard performances. However, it is now used as a complete music representation though it is inadequate for this purpose [94]. It has been used successfully as a tool for gesture analysis [52].

Studies in gestural analysis such as [122] have used MatLab to process data from 3D tracking systems. Technology has been used to augment instruments with extra sensors to provide more information on a performer's gestures: in [57], Young introduced the *hyperbow* — a violin bow with additional sensors for position, acceleration and strain. In [38], North Indian instruments were augmented with sensors to capture performance data.

The data from such augmented instruments has been used in teaching and more creatively. In [85], Ng *et al* described the i-Maestro system which uses gesture tracking and other performance data in an interactive learning environment. In [42], violin performance was recorded and analysed to improve a model for violin synthesis. In [45], Moody described a new instrument which used sensors to control a synthesiser.

In the analysis of music, the HumDrum toolkit [74] and its **kern [6] representation is used frequently. The HumDrum toolkit consists of a number of small programs which are chained together using the Unix command line. It uses **kern (a textual representation) to represent the music at each stage and regular expressions to select portions of the file.

The AMuSE system [76], in use at Goldsmith's University, implements the Abstract Data Type (ADT) introduced by Wiggins *et al* in [123] and the CHARM specification introduced by Harris *et al* in [70]. Wiggins' ADT abstracts the set of classes required to represent music allowing a single interface to be used with multiple back end representations. The CHARM specification provides a framework for the manipulation of musical structures. The AMuSE system attempts to implement both specifications and provide a single interface for computational music analysis to diverse collections of music using different representations.

Few generalised, programmable systems such as HumDrum or AMuSE exist, however. More research into computer analysis of music is conducted with ad-hoc or specialised systems such as the Rameau system for automatic harmonic analysis [100] or the Schenker system [117] for computer aided Schenkerian analysis [111].

In the phonological domain, there exist many software packages for editing music for production, composition and mastering. Programs such as the commercial Adobe Audition [28] and the free Audacity [9] allow for detailed editing of single audio files. Programs such as Pro-Tools [26] and Ardour [2] allow multitrack music to be edited.

The phonological domain should also cover the analysis of audio - an area where much research is currently underway. Software such as Sonic Visualiser [31] and the CLAM [4] toolkit allow detailed analyses of audio for the purpose of pitch-tracking and chord-naming among other uses.

So there are many fields in which computers are being used to study and manipulate music. Commercial software tends to concentrate on music composition and production. Academics tend to concentrate on specialised systems tailored to testing a particular idea: be it in the field of music analysis, gestural analysis or audio analysis. Few systems exist which are capable of accepting a wide range of queries. Such systems, where they do exist, concentrate on a single domain, e.g. HumDrum concentrates on the analytical domain, and do not include data from other domains.

1.3 Computers in Performance Musicology

At the beginning of this chapter we outlined the various artifacts which are manifestations of the musical object: the score and performance data. We have seen in Section 1.2 that computers have been used in the creation of scores, the performance of those scores and the recording and production of the performance. But how have computers been used in the investigation of the *process* of performance? How has the understanding of the conversion of a written score into an audible performance been enlightened by the use of computers?

There are two fields in which computers have been used for the study of musical performance: in analysing performance data; and in displaying and making available results of performance analyses.

General purpose software has been used in studies of musical performance. In [122], Wanderley *et al* investigated the movement of clarinet players during performance. Performers were recorded with a motion tracking system and the resulting data was analysed in MatLab.

Entire systems have been designed purely for analysing performance data. In [36] and [35], Camurri *et al* outlined the EyesWeb system for processing and visualising musical performance. The system allows patches to be created between processing blocks (inspired by analogue synthesisers). These patches can process video, sensor data, audio and MIDI to create analyses of expressiveness in music and dance performance. The system outputs is to audio, MIDI and video.

Technology for recording performance data has been used for improving computer performances. In [42], researchers used 3D motion tracking systems to record violin performances. This data was then used to create a model of the performance. Parameters extracted from the performance were then used to create a more expressive synthesised computer performance.

Systems have been devised for the visualisation of performance data synchronously with audio recordings. In [87], Kurth *et al* describes a system which allows performances stored in a database to be searched and played on a remote computer through a special client program. The client software plays the audio and shows a MIDI piano roll representation of the performance.

Systems have also been devised for the visualisation of performance data alongside music notation. In [51], Bresin *et al* introduces the Director Musices system for synthesising expressive musical performance from MIDI files. The output of the program is viewable in graphs which have notation shown aligned to a real-time axis.

All of the above systems align performance data to a real-time axis. Even the Director Musices system which can show notation shows it aligned to a real time rather than score time resulting in a rather ugly and obscured score. Since the performance is a manifestation of the score and not the other way around it makes sense to align performance data to the score. Furthermore, this allows visualisation of different performances and different data sources alongside the very information they intend to convey.

In a survey of tools used by members of the Music Information Retrieval community [33], Lamere listed 67 tools in 7 categories, none of which are able to show performance data aligned to a score.

So much research is being undertaken in computer systems for score analysis, for audio analysis and performance data analysis. There are competent systems for all of these fields but none which are capable of combining data from all three analyses and none which can display the results aligned with the score.

1.4 Requirements of the System

We have seen from the previous section that current tools lack the ability to make available musical scores and their performance data for generalised queries and lack the ability to display results in a way that is suitable for musicological analysis (i.e., alongside the score). The need for such a system is clear if we are to enlighten the process of performance. The current work describes a system that aims to address this need. Before detailing this system it is appropriate to outline what the requirements for such a system are.

In [93], Balaban states that a good knowledge representation framework should:

> ‘respect the requirements of the real world problem, should respect the characteristics of the available information about the problem, and should take into account user demands, such as convenient media of expression.’

The requirements below aim to satisfy Balaban’s criteria. The requirements aim to respect the real world problem by selecting appropriate representations for the data; respect the characteristics of the available information by providing suitable functions to manipulate the data; and take into account user demands by remaining interoperable with existing systems where possible and presenting performance data alongside the score.

1. **A representation of the score used for the performance should be stored without loss of data.** The intention here is to provide a record of the logical information in the score from which the performer performed. Specifying ‘without loss of data’ means that the representation should be chosen which can store the score maintaining the musical meaning of the contents so that it can be processed by music analytical queries. The detail should be sufficient that the score can be reproduced from the data stored. Specifically, MIDI would be unsuitable here as it would be unable to distinguish between certain notes and unable to store score annotations such as key signature.
2. **The representation should be able to cope with music written for different tuning systems.** As a test for the applicability of the chosen method of representation and the associated processing functions, it is proposed to test the system with music written for different tuning systems. If the representation correctly encodes musical meaning it should be able to cope with music in different tuning systems. This would also broaden the range of music that the system could analyse wider than other systems which are hard-coded for 12-tone equal temperament.
3. **A recording of the performance should be stored.** Clearly to make a comparison between a musical score and it’s performance, some form of performance data must be stored. This may take the form of:
    (a) Audio
    (b) Video
    (c) Other performance data such as MIDI or motion sensing data.

4. **The system should enable the analysis of the score, the performance and the score and performance combined.** To enable analysis of the process of performance, the means by which a performer interprets and reflects the musical structure, score and performance analysis must be combined. To achieve this:

   (a) **The system should be able to represent musical structure such as bars and ties.** It is not enough to simply represent those aspects of a score which describe sound i.e., notes and rests. Those marks on a score which reflect some of the piece's structure must also be stored so that they are available to later analyses.

   (b) **The system should be able to represent segmentations and analyses of performance data** It is not enough to simply store raw performance data. The system should allow the storage of the results of analyses of the performance data.

   (c) **The performance data must be pre-processed to include score-performance matches.** In order to allow investigation of musical performance, the correspondence between notes in the score and events in the performance must be established. For the current work, the data will be pre-processed using the score-performance matching tools created at the Centre for Music Technology, Glasgow University, by Douglas McGilvray for his doctoral thesis [58].

   (d) **The system should allow for the manipulation of data by the computer.** Manipulation may take the form of musical operations such as:

      i. transposition
      ii. interval calculation

   (e) **The system should be capable of running many different and diverse queries.** It should be a generalised, programmable system (like HumDrum) rather than a specialised solution (like Rameau). This will allow for the widest possible range of queries into the relationship between musical structure and performance, including those which have not yet been thought of.

   (f) **Able to present results in such a way that the performance data can be seen alongside or in the context of the score.** In order to understand the process of performance, by which a performer analyses and interprets a score in their performance, we should be able to display performance data in the context of the score from which it originates.

5. **Runs and is accessible from as many platforms and in as many different programming environments as possible.** Lamere's survey [33] of tools in use by the MIR community shows that no single computer environment is more popular than others — researchers used Windows, Macintosh and GNU/Linux operating systems. For the current system to be of use to the widest possible community of researchers, it should run on all the different operating systems in use by the community. The previous sections have shown that there are many different systems in use for the analysis on music and performance. These systems are created in different programming languages and environments. For example, the Rameau, AMuSE and Lilypond systems all use LISP or a variant; SonicVisualiser and CLAM tools use C++; CSound and ChucK have a C-like syntax. The current system should be accessible from as many of these programming environments as possible for it to be of any use to the existing research community.

6. **Compatible with existing formats — both input and output.** The system will not operate in isolation but will rely on the outputs of other tools and will provide data for further analysis with other software. As we have seen in Requirement 5, researchers currently use a diverse range of tools. For the system to be of most use, it should try to remain interoperable with as many of these as possible.

7. **The interface to the data should be created so that it can be understood by musicians, analysts and programmers alike.** The system will be useful to researchers who may not be computer programmers. Therefore, the interface to the system should enable non-programmers to access and query the data.

So we require a system which can store and analyse a score and store and analyse performance data. To be of most use, it should be able to accept many diverse queries rather than be tailored to performing a single type of analysis. Since researchers use a diverse range of software, it should be able to interoperate with as much of this as possible. Finally, it should be understood by all the different possible users who might operate the system.

1.5 Representation

Since a system that fulfils all the requirements in Section 1.4 will need to represent data from many different sources, it is useful to review how other researchers have approached the issue of representation. The five domains of music information used earlier are used here to structure this investigation.

1.5.1 The Logical Domain

The logical domain is described in [97] as the essential parts of a piece of music that are common to the other three domains: visual, analytical and gestural. The logical domain must contain the minimum information so that an automaton can:

- produce a minimally acceptable synthetic performance of a piece
- produce a minimally acceptable printed score

It could be argued that a MIDI representation can fulfil the first requirement. However, its limited representation of pitch (see Chapter 2) is not capable of producing a score. A more detailed representation is required.

The field of file interchange has conceived many formats for the description of a musical score such as the Music Encoding Initiative (MEI) [109], WEDELMUSIC [46] and MusicXML [68]. MusicXML stands out for its wide adoption by both commercial and open-source software. This makes it a good candidate to fulfil Requirement 6 — for interoperability with existing systems. In spite of (or perhaps because of) MusicXML's use as an interchange format between notation software, it holds little information about the position of glyphs on a page. Instead it stores each note as a combination of note name and accidental, with a duration given in fractions of a crotchet. Clefs, key signatures and time signatures are all implied, as in common western notation, to apply to all subsequent notes unless cancelled by the appearance of another. The MusicXML provides enough detail to enable a basic printed score and a basic synthesis of a piece of music so provides sufficient information for the logical domain.

1.5.2 The Visual Domain

The graphical representation of scores has followed two patterns: representation using existing graphical file formats and representation using music specific formats. The automated notation typesetting systems abc [1] and Lilypond [16] take text files as input and output scores to purely graphical representations. Abc produces Postscript whilst Lilypond can produce Postscript, Portable Document Format (PDF) and Scalable Vector Graphics (SVG). Representing scores in existing graphical formats has the advantage that the score will be available on more systems (see Requirement 6 in Section 1.4) however the disadvantage is that the conversion is one way. Once the score is in the graphical format it is no longer possible to turn that data back into the logical score it represents. Since the SVG format is XML-based, it provides the opportunity for extension to represent the logical score alongside the graphical score. Preliminary efforts towards this are examined in Chapter 7.

The second pattern uses music specific formats for the score representation. Perhaps the oldest such format is the Digital Alternate Representation of Musical Scores (DARMS) [114]. It indicates exactly how a score should be printed on the page whilst maintaining the link between the musical meaning of the symbols. It does, however, imply some structure. For example, a G♯ in the key of G♯ would be given the same representation as a G in the key of G. The absolute pitch is implied by the presence of the key signature rather than explicitly stated.

The Notation Interchange File Format (NIFF) [69] was designed as a interchange format between notation software. Both DARMS and NIFF are considered obsolete as they are not supported by current notation software.

1.5.3 The Gestural Domain

The study of gesture in music has taken many forms from a more artistic interpretation of the term to a purely mechanistic interpretation. For this reason it is useful to examine some definitions of the term before engaging in discussion of the research in this area.

A dictionary definition would provide a basic insight what is meant by the term gesture.

> **gesture** *noun* (MOVEMENT) a movement of the hands, arms or head, etc. to express an idea or feeling. *verb* to use a gesture to express or emphasise something. [34]

In [49] Bolton further separates the above into 3 key concepts:

- 1. Motion
- 2. Expression
- 3. Idea or emotion

The composer expresses their idea through the score which is interpreted (akin to adding emotion) by the performer. The composer can add further emphasis through score markings which the performer can express through changes in tempo and dynamics. Finally, the motion of the performer acting on their instrument is what causes the sound. These motions are further separated into *excitation gestures* which *cause* sound - such as the plucking of a string - and *control gestures* which modify a sound - such as the position of a finger on a fretboard.

In [122] a three-tiered system of gesture classification is used after [56]. *Effective gestures* (like excitation and control gestures above) are those responsible for actually producing sound. *Accompanist gestures* are body movements that are not necessary to produce sound such as movement of the head and shoulders. *Figurative gestures* are audible gestures with no direct correspondence to a physical movement such as note articulation.

For the practical measurement, storage and analysis of gestural data the classification may be restructured to describe three levels of complexity as in Figure 1.1. At the bottom level is the recording of gestural parameters such as movement of the clarinet bell or the MIDI recording of a keyboard performance. At this, the *data* level, raw data is recorded and stored. Basic analyses can be made at this level such as calculating the duration of notes from MIDI data. The *aggregation/segregation* layer collects or separates the gestural data into basic units of gesture. For example, in [122] x, y and z coordinates are joined to represent a circular motion of the clarinet bell. Currently this work is usually carried out by a human operator [66] however, with the correct programming and established taxonomy of gestures, the aggregation/segregation of gestural data is a good candidate for the computerisation of performance analysis. At the information level significant gestures are examined for their information. For example, a circular motion of the clarinet bell may be used to signify the end of a phrase of music.

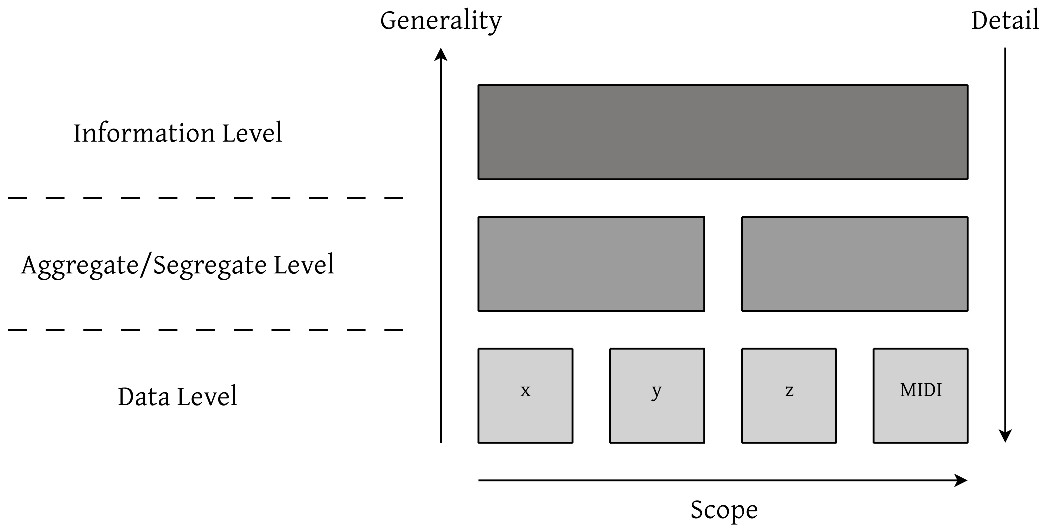


Figure 1.1: Gesture levels

The level of detail in the data increases as we descend the levels: The data level has the finest detail of the gesture. Conversely, the generality of the information decreases as we descend the levels: the x, y and z coordinates stored at the data level do not explicitly tell us what the performer was signalling with that gesture. Alternatively , we can say that there is a broader scope at the information level.

The problem of representation and storage at the data level is addressed in [88] which describes the Gesture and Motion Signals (GMS) data format based upon the widely used Interchange File Format (IFF) [95]. The format is a binary file that, at its lowest level, describes a single track of mono-dimensional sampled data which is recorded from motion sensors or other gestural capture devices. These tracks are grouped together into channels according to their related geometric dimensionality. For example, a channel might consist of several dimensions such as x, y and z co-ordinates. Units gather several channels that are related structurally. For example, a hand unit could comprise of the channels for each finger. Finally, a scene is constructed of units that are not dynamically linked e.g., the units representing each musician in a performance.

The GMS file format provides a model for the storage of gestural data. However, to process the data usefully, more work needs to be done at the aggregation/segregation layer, such as producing a taxonomy of gestures for a particular instrument.

The Performance Markup Language (PML) [25] (Section 5.3.1) goes some way towards providing a representation for the Aggregation Layer. It is an XML file format which allows performance events in external files to be indicated.

1.5.4 Analytical Domain

Representation in the analytical domain is characterised by two main approaches: text based representations and MIDI data.

The HumDrum [74] toolkit uses the **kern [6] text file format. Music is arranged in rows and columns. Time increases from left to right and simultaneous events are placed in the same column. Pitches are represented in text i.e., G♯ is represented as `G#` A text representation allows files to be processed with standard Unix command-line tools which are highly efficient at processing text files. For example, a particular phrase can be searched for using a regular expression[1]. The representation, whilst being easy to read, suffers from a lack of musical meaning. For example, a sensible assumption for a text representation would be that G is followed by H and then I. There is nothing inherent in the representation which expresses that a G is followed by an A.

The Rameau [100] system for automatic harmonic analysis uses the Lilypond [15] file format which is a textual representation of a score using TEX-like syntax. The representation is easy to read but suffers from the same problem as the **kern format in that it does not embody any musical meaning.

The POCO system [72] for analysing, modifying and generating expression in musical performance used the MIDI format for all its input and output. Whilst the

[1]Regular expressions are a language for defining text matches. The syntax has become standardised and is available as an optional library for most programming languages. An excellent reference can be found in 'Mastering Regular Expressions'[64]

MIDI format does encode some musical meaning (increasing the pitch value increases the musical pitch) there are numerous other problems already mentioned such as the inability to distinguish between enharmonically equivalent but functionally distinct pitches [94].

1.5.5 Phonological Domain

The representation most commonly found in the domain of audio artifacts is sampled, pulse-code modulated audio usually found inside a Microsoft Wave (or wav) file. The Sun au and Apple aiff file formats are used little in modern computer music software.

Also of interest in the phonological domain are the various compressed audio formats. The most prolific of these is MPEG 1 Layer 3 (or mp3), but there are others such as the Advanced Audio Codec (aac), MPEG4 Audio and Ogg Vorbis.

1.5.6 Modelling Music

In the previous sections we have looked at the different domains of music and the attempts to represent musical data in that domain. We can take an alternative approach and investigate the different data modelling approaches that researchers have used to describe musical data and its relationships.

Most well-known data modelling paradigms have been applied to music research. In [93], Mira Balaban categorises the different approaches to music representation in different music processing systems. Firstly, she discounts graphical representations such as DARMS [114] since a picture of music is not suitable for music processing. She goes on to identify grammar-based approaches, such as Smoliar [117] or Lerdahl and Jackendoff's Generative Theory of Tonal Music (GTTM) [63], which manipulate strings of sounds or events. Other systems manipulate multi-dimensional arrays of sounds or events, most often consisting of pitch and time, as described in [65] where Loy describes the variety of uses to which MIDI data is being put. Another group of systems adopt an object-oriented approach such as Rodet and Cointes' FORMES system [37] and Steven Travis Pope's MODE system [110]. Finally, Balaban describes hierarchy-based systems, such as her own Music Structures approach [89] and Smail *et al* 's CHARM representation [116] which process structures of musical entities.

The entity relationship model proposed by Chen [102] was extended for music applications by Rubenstein in [121]. The concepts of ordering and hierarchies were needed to fully represent aspects of musical structure such as membership of a chord. Rubenstein outlined a set of musical primitives which are needed to represent Common Music Notation such as notes, rests, chords and bars and their hierarchical relationships.

In [75], Lane and Punch presented a relational database approach to storing and querying music. A simple relational schema was populated from MIDI files. A regular expression syntax and parser was developed to search the database and provide matches. The performance of the system was shown to be theoretically sound and practical for general use.

In [44], Eaglestone *et al* used an extended relational database to store different versions of a composers artifacts to support the composition process. The model was designed using an object-oriented methodology which was then mapped onto the extended relational database POSTGRES. The model was described in more detail in [43].

An object-oriented approach was taken by Alvaro *et al* with the EV meta model in [79] where a model was proposed for computer-aided composition. The model consists of three core classes which are extended to specialise for different applications. The classes are events, parameters and dynamic objects. Events consist of a start position (in time), length (duration), a list of parameters, a list of events (thereby allowing recursion) and a position function which modifies the time position. A parameter consists of a name and value. The value is a dynamic object. The dynamic object could be a value or a curve describing many values or a sequence or any of a number of different objects.

When designing a data model there must inevitably be a trade off between the simplicity of the primitives of the model and their descriptive completeness. The EV model defines a very simple set of primitives. Any practical realisation of the model would require a large amount of specialisation to turn the simple primitives into useful recognisable musical objects.

An object-oriented approach was taken by Hirata and Aoyagi in [84] where Lerdahl and Jackendoff's Generative Theory of Tonal Music was implemented in a deductive object-oriented database. They defined a set of objects and a set of primitive operators describing polyphony and used the GTTM to analyse a set of pieces with partial success.

In [101], Roland proposes XML as a suitable format for music information retrieval citing a number of reasons including the openness and advantages in interoperability which it provides. Roland's proposal was implemented in [81] where Ganseman, Scheunders and D'haes used the XQuery XML query language to query a database of musical scores represented in MusicXML format. The queries shown are generally statistical or meta-data based. They show one query of rhythmic pattern matching and admit that more work needs to be done to provide a library of functions which can perform music analysis.

In [62], Deliège and Pederson propose Music Warehouses. These take the idea of a data warehouses and apply it to music. A data warehouse represents data as a multi-dimensional cube. Queries are answered using an approach called Online Analytical Processing (OLAP) where aggregations are pre-computed along selected axes to reduce query times. A data warehouse can be implemented in a dedicated database (called Multidimensional OLAP or MOLAP) or built upon a relational database (Relational OLAP or ROLAP). Deliège and Penderson expect OLAP to provide faster query response times for larger databases but outline a number of issues which must be overcome before the approach can properly be applied to music.

So many different data modelling paradigms have been applied to music data. No single model has been taken up as a standard and applied consistently for music data modelling. Rather researchers have used whatever model has been appropriate to represent music for their specific application. The wide variety of models in use reflects the difficulty in creating a model which captures all aspects of music data for all applications. All researchers can hope to do is choose an appropriate model for their application from the currently available arsenal of tools.

1.5.7 Issues in representation

The previous sections have shown that many different representations are in use for the different domains of music information. These different representations fulfil to different degrees the requirements set out in Section 1.4.

In the logical domain, MusicXML provides enough detail for representing the logical information of a score and for reproducing the score. It is therefore a suitable model for fulfilling Requirement 1 for a score representation. In the visual domain there exist widely used representations for scores but none which retain the musical meaning of the score they represent. Using a purely graphical representation might be sufficient in most cases but the ability to interoperate with other systems would be improved if the system outputted the musical information along with the graphical presentation. There is potential for SVG to fulfil this and support Requirement 6 for interoperability.

In the gestural domain, there are representations for low-level data but few which can aggregate this data into meaningful gestures. PML provides the capability to aggregate and segregate this low-level data and so provides a model for fulfilling Requirement 4b to allow the analysis of performance data.

In the analytical domain there exist several representations in use but none which encode musical meaning. The requirements to enable analysis and manipulation of the data (Requirements 4 and 4d) would be expedited by a representation which encodes the musical meaning. In the phonological domain, using the most popular representations will ensure the system remains interoperable with other software.

The above makes clear that there are many choices to be made when creating a music representation. Some combination of current representations would be needed to fully meet the requirements laid out in Section 1.4. In [73] Honing outlines several factors required of a music representation scheme which should help in this choice. He differentiates between types of knowledge representations: declarative versus procedural. Declarative knowledge is information *about* something whereas procedural knowledge is information describing *how* to do something. Problems arise with procedural representations because the structure is implicit and often difficult to inspect at any point in the procedure. As we have seen, most representations are declarative since they allow explicit access to the data.

A further issue in knowledge representation is decomposability: the ability to reduce the subject to appropriate and meaningful primitives. In the field of music representation there is broad agreement that the basic primitive of music is the note. All the representations of symbolic music we have seen share this in common; though what is actually meant by a note is poorly defined. Do all notes have a fixed or clearly defined pitch? Do all notes have a duration which can be defined as a power of 2? From a Western music point of view these questions can seem unimportant. Nevertheless they do need careful consideration when creating a music representation system which might represent both early, modern and non-Western music.

Finally the choice between continuous or discrete knowledge representations is discussed. This is of particular significance to the representation of time. Note durations in scores are represented as a discrete multiple or fraction of a whole note. This relational method is dependant on the tempo and the speed of the performer. Conversely durations from audio files are represented in continuous time: milliseconds or seconds.

Honing outlined a set of issues with the structure within a music representation:

- **Tacit** The data has no structure. For example, MIDI data is just a stream of primitives.
- **Implicit** The structure can be derived or calculated from the representation.
- **Explicit** The structure is explicitly described in very few cases.
- **Types of structure**
  - *part-of* a note can be described as being *part of* a chord.
  - *is-a* a note *is a* crotchet.
  - *N-ary* a binary relation can exist between a score symbol and its corresponding MIDI note. A more complex relation could exist between a note, the key signature and the tuning system for the piece.
- **Dedication vs generalisation** A dedicated structure defines primitives for every type of musical construct whereas a generalised structure defines a general primitive from which others are derived.

- **Relation to time** Objects' temporal relations to each other can be explicitly stated. For example, this bar follows that bar, this note overlaps that note.
- **Multiple representations** Including multiple representations of the musical data allows greater flexibility.

In his conclusion, Honing states that a representation should be as formal as possible (not requiring knowledge from outside the formal definition), declarative, explicit, include multiple representations and relate all objects to time.

1.6 Summary

This chapter began by outlining that the aim of the current work is to create a computer system capable of both music and performance analysis with the hope that such a system would be able to enlighten the process of performance. The review of the use of computers in music found that they were being used in 5 domains: logical, visual, gestural, analytical and phonological. Few systems exist which can accept a wide variety of queries — they tend to be more specialised. Where such systems exist, they tend to concentrate on a single domain.

We looked at computers in performance musicology and found that they were used in the analysis of performance and in the displaying of results. Whilst there were competent systems in this area, few combined results from analysis of different domains and none aligned results with the musical score.

These findings became a set of requirements for a system for the analysis of musical performance: that it should allow analysis of the score and performance data and it should provide results aligned with the score. Additionally, such a system should be capable of representing microtonal scales as a test of the music representation; and it should be compatible and interoperable with existing software.

Since such a system would need to represent data from many different sources, a review of the different methods employed in the representation of music data was given. This reviewed representations under the 5 domains mentioned above. It also looked specifically at the area of music data modelling where most of the data modelling techniques from outside music have been applied to musical data with no single model gaining broad acceptance. Rather, researchers choose the model which is most suited to the application.

The final section looked at the issues in representation and looked at Honing's criteria [73] for a music representation.

This chapter has shown that to understand the process of performance, it is necessary to combine score analysis and performance analysis in a system which is generalised rather than specialised to a single type of analysis. Such a system does not yet exist. The design of such a system will need to pay special attention to representation of musical data if it will successfully combine data from many different sources.

1.7 Outline of this thesis

The current work aims to provide a system which is capable of enlightening the process of performance by providing the ability to analyse scores and performance data together and present results alongside each other. As shown above, representation is a critical issue in creating such a system, so the following 3 chapters are dedicated to describing the representations chosen for the current work. Following these is a chapter explaining how the system has been implemented to enable generalised queries. Examples of queries on real data are given followed by a discussion of the success of the system.

A pitch representation is described in Chapter 2 which is based on the spiral of fifths. Whilst the idea of using the spiral of fifths to represent pitch has been suggested before [86, 55], musical transformations using this representation have not been described. Furthermore, the current work extends the representation to microtonal[2] scales based on the diatonic scale of 7 tones (C, D, E, F, G, A, B). It is not able to represent all microtonal scales: scales such as the Partch [99] and the Bohlen-Pierce [90] would require a different representation.

[2]Microtonal scales use intervals smaller than a semitone

A representation of musical time is outlined in Chapter 3. The representation is based on that used by the MusicXML file format with the addition of explicit start times for all musical events.

A set of musical functions necessary for the manipulation of music is described in Chapter 4. These functions provide the basis for computer-aided score analysis and use types similar to those introduced in Wiggins *et al* 's CHARM specification [116].

The entire system architecture is described in Chapter 5 from how data was gathered and processed to the design of the database system used for storage of the data, how the system is queried and how presentations of performance data aligned with score data are created.

The database implements the pitch and time representations and musical functions outlined in the previous chapters. The database system is designed so that the representation is declarative and well defined. For example, operators (such as '+' and '-') have been redefined so that they 'do the right thing' on custom types; explicit: for example, all notes have an onset time rather than an implied onset time as found in many other representations; enables multiple representations: the representation uses score notes and note groups to provide a flexible method to extend the representation; related to time: all objects have explicit start times and, where appropriate, durations or end times.

The system aims to fulfil all of the requirements in Section 1.4 including Requirement 4e (to be queryable): the new musical functions and types added to the database extend the SQL query language and enable multiple different queries to be created of the same data; Requirement 5 (to be accessible): the database and associated software run on all major operating systems and are available free of charge; and Requirement 6 (to be interoperable): the system accepts scores in the MusicXML format and outputs results to popular image formats such as PDF (Portable Document Format), PNG (Portable Network Graphics) and SVG (Scalable Vector Graphics).

A series of queries of real performance data are included in Chapter 6. The queries show the system coping with music in different tuning systems; calculating different types of performance data; using the musical functions; and displaying results in different ways. The queries used here have formed part of the investigations of musical performance in [80, 78, 77].

The system and its implementation is discussed in Chapter 7 and conclusions made in Chapter 8. The system is also described in [118].

The Appendix lists the Lilypond markup generated by the queries in Chapter 6 (Appendix B) in full.

Chapter 2

Pitch

The aim of the current work is to create a system capable of analysing musical scores and score-aligned performance data and displaying the results of those analyses alongside each other. To achieve this it is necessary to choose an appropriate representation of the musical data so that it may be analysed in a computer. The next 3 chapters will be dedicated examining the representation of the musical score data for this purpose. In particular, this chapter concentrates on the representation of musical pitch.

In Section 1.4 in the previous chapter, several requirements were given which a music representation must fulfil. Of particular relevance to the topic of this chapter are Requirements 1 and 2 which state that the system should represent the logical data of the score in a musically meaningful way and should be capable of representing microtonal scales. In the chapter that follows we will first examine some examples of previous work in computational pitch representations and assess them against these requirements. We will examine in some detail the pitch representation chosen for the current work and look at how it can be extended to represent microtonality and how it can be manipulated to achieve musical transformations necessary for musical analysis.

Pitch is the primary percept involved in the understanding of music and is one of the most important factors which distinguish musical sound from other sounds. It is therefore essential that any musical application be able to adequately represent pitch.

2.1 The Need for a Pitch Representation

The system described here is intended to process and analyse musical scores in a computer. The system is capable of converting queries expressed in musical terms into computable problems. To achieve this it is necessary to have an internal pitch representation which behaves in a musically correct fashion.

One may think that it would be sufficient to store a textual representation of a pitch: a 'c' character for the pitch C. However it does not take long to realise that a representation such as this would present difficulties: how do we add an interval to the pitch? We could evaluate the difference in the notes' alphabetic ordinality (making sure we have not crossed an octave boundary), and according to whether the interval is a major or minor we might move a semitone in either direction. We have to be aware that in the 12-tone system there are no semitones between an E and an F and that if we are in a different tuning system, this might be different.

All these consistency checks require programmers to program them and computing power to perform. It is therefore preferable to use a representation which can properly reflect the tonal structure whilst being easily computable without a large number of consistency checks.

2.2 The Requirements of a Pitch Representation

The notion of pitch in Western classical music theory has several facets. Pitches can be ordered from low to high; they can be grouped according to their class e.g., D, D ♭ and D♯ are all in the same class; notes an octave apart are sometimes considered equivalent. Thus, for a representation to retain this multitude of meanings it must incorporate some form of periodicity and allow grouping.

Many text-based representations exist, such as ***kern* used by the HumDrum music analysis toolkit [74]. Whilst a text description of music allows for easy creation and editing of music files by humans, it does not fit well with the key aptitude of a computer for numerical manipulation. Since computers deal mainly in numbers then any pitch representation would have to be capable of being manipulated numerically.

There is a great deal of modern music written using microtonal intervals. For example, the works of Ligeti, Ives and Ferneyhough all include compositions in 24 divisions of the octave. Many older works, going back to the 16th and 17th centuries, expanded the repertoire of pitch names beyond the diatonic seven. For example, the piece “Ut, re, mi, fa, sol, la.” by the 17th century composer John Bull notates 18 different pitch names and it is highly unlikely that tuning systems of that era would have pitched the notes now considered enharmonically equivalent as the same.

The microtonal scales mentioned above and those dealt with in this chapter are from the class of scales based on the diatonic scale: that is the 7 familiar note names with a varying number of accidentals in addition. We do not attempt to represent non-diatonic scales such as the Partch scale [99] or the Bohlen-Pierce [90]. Such scales are outside of the scope of the current work as they would require significant work to represent. The quantity of work added by extending into diatonic microtonal scales is significant enough to make the exercise worthwhile.

The work of Easley Blackwood [48] suggests that pitch-representation needs to make allowances for the functionality of pitches and not just their frequency values, and that one avenue of approach to the general theory of pitch representation would be to test pitch-representation systems in the context of microtonal music with more than 12 notes to the octave. Such a system may also extend to historic tuning systems with more than 12 pitch signs per octave. Therefore we will require the pitch representations to be capable of representing music written for microtonal tuning systems.

2.3 MIDI Representation

An efficient representation of pitch could assign each pitch with an integer. For example the 12-tone scale could be assigned with numbers from C $\leftarrow$ 1 to B $\leftarrow$ 11

C	C♯/D♭	D	D♯/E♭	E	F	F♯/G♭	G	G♯/A♭	A♯/B♭	B	C
1	2	3	4	5	6	7	8	9	10	11	12

Table 2.1: 12-tone number-line scheme similar to that used in the MIDI code. Each note of the 12-tone scale is assigned an integer value from 1 to 12.

1	2	3	4	5	6	7	8	9	10	
C♭	C	C♯	D♭	D	D♯	E♭	E	E♯	F♭	
11	12	13	14	15	16	17	18	19	20	21
F	F♯	G♭	G	G♯	A♭	A	A♯	B♭	B	B♯

Table 2.2: 21 note-name number-line representation. Each of the 21 note names, consisting of the 7 naturals, the 7 sharps and the 7 flats, which make up the familiar set of note names most often encountered in 12-tone music, are assigned a number from 1 to 21

as in Table 2.1.

This is the approach adopted by the MIDI protocol. Though efficient and succinct it fails to distinguish between notes which are enharmonically equivalent. For example, both D♭ and C♯ are assigned the same number. Table 2.1 could be expanded to separate out the note names which are enharmonically equivalent and give them a unique number assignment.

Suppose the notes of the twelve tone scale were assigned numbers starting at C♭ $\leftarrow 1$ to B♯ $\leftarrow 21$ as in Table 2.2.

The interval of a major 2^{nd} from C to D is 3 steps where C $\leftarrow 2$ and D $\leftarrow 2+3=5$. However the same interval from E is 4 steps: E $\leftarrow 8$ and F♯ $\leftarrow 8+4=12$. This presents difficulties when computing the interval between two notes. The requirement for interval invariance was introduced in [71]. It is the property of a representation which determines that the number of steps of a given class of interval remains the same no matter what the value of the starting note. So a major 2^{nd} would always be 3 steps and never anything else. Clearly the 21 name scheme fails to satisfy this criteria.

0	1	2	3	4	5	6	7	8	9
C𝄫	C♭	C	C♯	C𝄪	-	D𝄫	D♭	D	D♯
10	11	12	13	14	15	16	17	18	19
D𝄪	-	E𝄫	E♭	E	E♯	E𝄪	F𝄫	F♭	F
20	21	22	23	24	25	26	27	28	29
F♯	F𝄪	-	G𝄫	G♭	G	G♯	G𝄪	-	A𝄫
30	31	32	33	34	35	36	37	38	39
A♭	A	A♯	A𝄪	-	B𝄫	B♭	B	B♯	B𝄪

Table 2.3: The Base-40 pitch numbering scheme. Each of the 35 note names of the 12-tone scale, that is the 7 sharps, 7 flats, 7 double sharps, 7 double flats and 7 naturals, are given a number from 0 to 39 with gaps inserted between note names separated by whole-tone intervals. The gaps ensure the representation has the property of interval invariance: an interval is always represented by the same number of scale steps no matter what the starting note.

2.4 Base-40

The Base-40 system [71] addresses the issues raised above: that of interval invariance. It represents pitches in such a way that the difference between any two pitches separated by the same interval is always constant. The system uses a 40 position number line to equate to the 35 note names produced by the ♯s, ♭s, 𝄪s, 𝄫s and ♮s of the 12-tone scale as in Table 2.3. The scheme introduces gaps of unassigned numbers to provide the required interval invariance. For example, when comparing Tables 2.2 and 2.3, it can be seen that in the Base-40 system a major 2$^{\text{nd}}$ is always 6 steps. The gaps have created a system where each musical interval will require the same number of steps no matter what the starting note.

Another useful property of the Base-40 system is octave wrapping. The number line continues above 39 and below 0. The note number can be extracted from values outside of the range 1-40 by finding the remainder as in Equation 2.1.

$$\text{note} \leftarrow \text{Rem}(k, 40) \tag{2.1}$$

where k is the note value and $\text{Rem}(x, y)$ is the remainder when x is divided by y and is given by[1]:

$$\mathrm{Rem}(x, y) = x - y \left\lfloor \frac{x}{y} \right\rfloor \tag{2.2}$$

The octave can be extracted by dividing by 40 as in Equation 2.3[2]. The result (as an integer) gives the octave relative to the the 40 point number-line.

$$\text{octave} \leftarrow \frac{k}{40} \tag{2.3}$$

2.4.1 Extending to Base-X for Microtonality

The system described above can be extended to represent microtonal scales. We can start by generalising the creation of the representation thus:

$$X \leftarrow 7 * (2n + 1) + 5 \tag{2.4}$$

where n is the number of sharps or flats required per semitone and X is the base of the system. For the Base-40 system, $n \leftarrow 2$ thus $X \leftarrow 40$. This can be extended to include tones less than a semitone apart:

$$X \leftarrow 7 * (2nz + 1) + 5 \tag{2.5}$$

where z is the number of divisions per semitone. Setting $n \leftarrow 2$ and $z \leftarrow 2$ we can produce a system where $X \leftarrow 73$ and we can represent music written in the 24-tone scale. An extract of this system is shown in Figure 2.4.

There remains a problem with all Base-X representations: that of the gaps or holes having ambiguous pitch. Of course it is possible to increase the value of n so that the transposition lands on an unambiguous pitch, however, even then, it remains theoretically possible to add another transposition which will land on another gap. To

[1]The result of the calculation given in Equation 2.2 is the same as that returned by the modulo operator (%) in the Python programming language. However, it is not the same as that returned by the modulo operator or the remainder function in C and many other programming languages. Care must be taken when translating the equations given here to program code.

[2]All equations from here assume integer arithmetic

0	1	2	3	4	5	6	7	8	9	10
C𝄫	C𝄭	C♭	C𝄳	C	C𝄲	C♯	C𝄰	C𝄪	–	–
11	13	14	15	16	17	18	19	20	21	22
D𝄫	D𝄭	D♭	D𝄳	D	D𝄲	D♯	D𝄰	D𝄪	–	–
23	25	26	27	28	29	30	31	32	33	34
E𝄫	E𝄭	E♭	E𝄳	E	E𝄲	E♯	E𝄰	E𝄪	F𝄫	F𝄭
35	37	38	39	40	41	42	43	44	45	46
F♭	F𝄳	F	F𝄲	F♯	F𝄰	F𝄪	–	–	G𝄫	G𝄭
47	49	50	51	52	53	54	55	56	57	58
G♭	G𝄳	G	G𝄲	G♯	G𝄰	G𝄪	–	–	A𝄫	A𝄭

Table 2.4: Extract from base-73 pitch representation. This representation extends the Base-40 system shown previously to include microtonal intervals.

implement any Base-X system, the programmer must either trust that no unfeasible transposition will ever take place (and risk corrupted data if it does) or check the integrity of each requested transposition and take steps accordingly.

So the Base-X representation succinctly encodes musical meaning — increasing a value increases the pitch, modular arithmetic can be used to encode octaves — but it suffers from a practical problem when used in a system which mixes music in different tuning systems. No single version of the representation can correctly encode for all possible combinations of pitches and intervals without introducing ambiguities which must be checked for.

Furthermore, when manipulating music in a computer it is often necessary to know which notes are enharmonically equivalent — for example, when transcribing a MIDI performance. This information is not encoded into the Base 40 system. The data could be made available through the use of look-up tables for each tuning system however a more elegant solution is offered by the Binomial system described in the next section.

	C	D	E	F	G	A	B
♭	<11, 0>	<1, 1>	<3, 2>	<4, 3>	<6, 4>	<8, 5>	<10, 6>
♮	<0, 0>	<2, 1>	<4, 2>	<5, 3>	<7, 4>	<9, 5>	<11, 6>
♯	<1, 0>	<3, 1>	<5, 2>	<6, 3>	<8, 4>	<10, 5>	<0, 6>

Table 2.5: The 12-tone Binomial representation. Each pitch is represented by a tuple indicating the pitch class and name class of the pitch. Pitches with the same pitch class but different name class are enharmonically equivalent.

2.5 Binomial Representation

The Binomial system [53] is capable of representing pitches and intervals and achieves interval invariance. It also encodes for enharmonic equivalence.

The Binomial system encodes each note as an integer couple of the pitch class (pc) and name class (nc), denoted as `<pc,nc>` The pitch class is the *sounding* pitch of the note numbered from 0 to 11 for the 12 notes in the 12-tone scale. The name class is the *letter name* of the note – one of {C, D, E, F, G, A, B} – and is numbered from 0 to 6. The most frequently used note couples are given in Table 2.5.

Enharmonic equivalent notes can be found by comparing the pitch class of two notes. If it is the same then the notes are enharmonically equivalent. Transposition is as simple as addition or subtraction. For example, to transpose A up a perfect fifth we add the binomial representation of the interval to the couple which represents A:

$$\begin{array}{ccccccccc} <9,5> & + & <7,4> & = & <(9+7) \bmod 12, (5+4) \bmod 7> & = & <4,2> \\ \text{A} & + & \text{P5} & = & & = & \text{E} \end{array}$$

This example also demonstrates the use of modular arithmetic to track and wrap around octaves.

The Binomial system can be extended to represent diatonic scales with more or fewer divisions of the octave than 12. The number of pitch classes would be extended while the number of name classes would remain the same. For example, the 19-tone scale can be represented as in Table 2.6.

Comparing this table with Table 2.5 we see that the binomial couples are different

	C	D	E	F	G	A	B
♭	`<0, 0>`	`<3, 1>`	`<6, 2>`	`<8, 3>`	`<11, 4>`	`<14, 5>`	`<17, 6>`
♮	`<1, 0>`	`<4, 1>`	`<7, 2>`	`<9, 3>`	`<12, 4>`	`<15, 5>`	`<18, 6>`
♯	`<2, 0>`	`<5, 1>`	`<8, 2>`	`<10, 3>`	`<13, 4>`	`<16, 5>`	`<0, 6>`

Table 2.6: The 19-tone Binomial representation. Notes are given different values to the 12-tone binomial system

for the same pitch name. For example, D♭ is given the couple `<1,1>` in the 12-tone Binomial system and `<3,1>` in the 19-tone system. The consequence of this is that there is a different set of pitch couples and interval couples for each choice of octave division. This complicates making comparisons across musical pieces written using different divisions of the octave.

So the binomial system can represent pitches and intervals, achieving interval invariance and representing enharmonic equivalence. It does this at the expense of requiring a different representation for each number of divisions of the octave. Using the same representation for pitch and intervals, no matter how many divisions of the octave were used, would expedite comparisons across pieces written for different tuning systems, should make the system easier to learn and maintain and therefore should save programmers' time.

2.6 Spiral of Fifths Representation

This section outlines a pitch representation which aims to match the capabilities of the previous representations and surpass them in it's ability to represent tonal and microtonal scales using the same representation. It uses the spiral of fifths, familiar to anyone who has studied music theory, to order pitches. This gives each pitch a unique identifier yet still allows for calculation of enharmonic equivalence. It achieves interval invariance and can represent some microtonal scales.

The spiral of fifths is a graphical representation of the process by which diatonic scales are constructed through repeatedly adding the interval of a fifth and scaling the result to within an octave. The more common form of the cycle or circle of fifths represents the case where the position of a note coincides with a previous

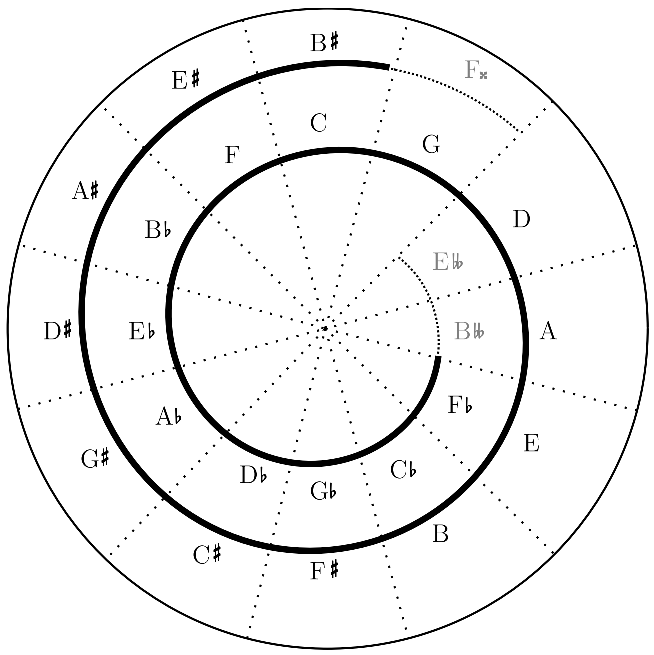


Figure 2.1: The 12-tone spiral of fifths — pitches increase by a perfect fifth in the clockwise direction and decrease by a perfect fifth in the anti-clockwise direction. Pitches in the same segment are enharmonically equivalent in the 12-tone equally-tempered scale.

note after a certain number of repetitions (scaling to within an octave). This exact correspondence occurs in an equally tempered scale. The repeated note is therefore enharmonically equivalent with the previous note at the same position and the complete circle represents a scale.

Figure 2.1 shows the spiral of fifths with the notes of the 12-tone equally-tempered scale around the circle. The enharmonic equivalence of notes 12 steps apart is indicated by their presence in the same segment. The relationship can be expressed mathematically as in Equation 2.6.

$$\left(N \times C_{P5,N}\right) \bmod 1200 = 0 \tag{2.6}$$

where $C_{P5,N}$ is the interval of a perfect fifth in cents and N is number of divisions in the octave. A result of zero shows that after N additions of the perfect fifth, we have got back to where we started. Using a 12-tone equally tempered fifth of 700 cents:

$$(12 \times 700) \bmod 1200 = 8400 \bmod 1200 = 0$$

Using a 19-tone equally tempered fifth of approximately 694.736842105 cents:

$$(19 \times 694.736842105) \bmod 1200 \approx 13200 \bmod 1200 = 0$$

The spiral of fifths pitch representation described here combines the efficiency of a number line using integer maths with a representation which is closely related to the principle underlying the construction of diatonic scales. Similar representations have featured in the work of Kolosick [86] and Meredith [92]. The representation I have developed is slightly changed from those used elsewhere and has been extended to support microtonality. It retains the same value for the same note in scales with different numbers of divisions of the octave and is capable of representing some microtonal scales.

The simplest form of the spiral of fifths pitch representation is given by representing 12- and 19-tone scales. An explanation of this representation and the associated musical operations is given in the next sub-section. This is actually a simplified special case of the more generalised system for representing all diatonic tonal and microtonal scales which is explained in the following sub-section.

2.6.1 The 12- and 19-tone Spiral of Fifths Representation

To construct the number line for the spiral of fifths we start numbering from F♮ $\leftarrow 0$ and step round the spiral incrementing the number by 1 at each note. Thus C♮ $\leftarrow 1$, G♮ $\leftarrow 2\ldots$ and in the opposite direction B♭ $\leftarrow -1$, E♭ $\leftarrow -2\ldots$ The complete number line for the most common note names is given in Table 2.7.

The number line is shown in a table for brevity. The top of the spiral of fifths (F♮) is shown in the top centre of the table (F♮ $\leftarrow 0$). Continuing clockwise along the spiral of fifths is equivalent to travelling down the columns (C♮ $\leftarrow 1$, G♮ $\leftarrow 2$...B♮ $\leftarrow 6$, F♯ $\leftarrow 7$...). Moving anti-clockwise along the spiral of fifths is equivalent to travelling up the columns.

	𝄫	♭	♮	♯	𝄪
F	-14	-7	0	7	14
C	-13	-6	1	8	15
G	-12	-5	2	9	16
D	-11	-4	3	10	17
A	-10	-3	4	11	18
E	-9	-2	5	12	19
B	-8	-1	6	13	20

Table 2.7: The spiral of fifths number line for scales with 1 division per semitone.

2.6.2 Musical Operations

It is clear from Table 2.7 that the interval of a perfect 5th is obtained by adding 1 to the note value: adding 1 moves one step around the spiral of fifths. Consequently a perfect 4th can be obtained by subtracting 1 from a note value. A major 2nd is obtained by adding 2: C $\leftarrow 1, 1+2=3=$ D. The list of the most common intervals and their values are given in Table 2.8. Thus transposition correlates to addition (Equation 2.7).

$$k_{\text{transposed}} \leftarrow k + i \tag{2.7}$$

where k is the note value and i is the interval value.

Interval calculation correlates to subtraction (Equation 2.8).

$$k_1 - k_2 \leftarrow i \tag{2.8}$$

where k_1 is a higher pitch than k_2. This raises the question of how to calculate whether one pitch is higher than another. It should be clear from Table 2.7 that a modulo 7 operation on any given pitch will yield its name class (i.e., its *row* in the table). We use the remainder function described above:

$$k_{\text{nc}} \leftarrow \text{Rem}(k, 7) \tag{2.9}$$

where k_{nc} is the name class (in the spiral of fifths representation) of the pitch k.

It should also be clear from Table 2.7 that the number of accidentals (i.e., the *column* in the table) can be calculated by dividing the pitch by 7:

$$k_{\text{accidental}} \leftarrow \frac{k}{7} \tag{2.10}$$

	i		
unison	±0	±0	octave
aug. unison	+7	-7	dim. octave
dim. 2nd	-12	+12	aug. 7th
minor 2nd	-5	+5	major 7th
major 2nd	+2	-2	minor 7th
aug. 2nd	+9	-9	minor 7th
dim. 3rd	-10	+10	aug. 6th
minor 3rd	-3	+3	major 6th
major 3rd	+4	-4	minor 6th
aug. 3rd	+11	-11	dim. 6th
dim. 4th	-8	+8	aug. 5th
perfect 4th	-1	+1	perfect 5th
aug. 4th	+6	-6	dim. 5th

Table 2.8: Interval values in the spiral of fifths pitch representation. Intervals of the same class are grouped vertically — all types of 2nd are shown together. It should be clear that each member of an interval class is 7 steps from the next member of that class meaning that a modulo operation can be used to identify membership of an interval class. Furthermore, intervals and their inversions are shown on the same row. It should be clear that a change of sign inverts the interval.

where $k_{\text{accidental}}$ is the number of accidentals of the pitch k giving positive values for sharps, negative values for flats and zero for naturals.

We can calculate whether one pitch is higher than another by comparing the pitch order positions given by the name classes and the number of accidentals. The pitch order position is the position of a pitch ordered from low to high. In this ordering, a C $\leftarrow$ 0, D $\leftarrow$ 1, E $\leftarrow$ 2... The value can be retrieved by using the name class in spiral of fifths representation (k_{nc}) as the index into a list of values as in Equation 2.11.

$$k_{\text{pop}} \leftarrow \{3, 0, 4, 1, 5, 2, 6\}[k_{\text{nc}}] \tag{2.11}$$

where k_{pop} is the pitch order position of k and is found by using k_{nc} as the index into a list of values. We can now compare the values to find the higher note. The procedure to achieve this is shown in Algorithm 1.

Algorithm 1 Algorithm to show the procedure to establish the higher note

```
if $k_{1\text{pop}} > k_{2\text{pop}}$ then
    $k_1$ is greater
else if $k_{1\text{pop}} < k_{2\text{pop}}$ then
    $k_2$ is greater
else if $k_{1\text{pop}} = k_{2\text{pop}}$ then
    if $k_{1\text{accidental}} > k_{2\text{accidental}}$ then
        $k_1$ is greater
    else if $k_{1\text{accidental}} < k_{2\text{accidental}}$ then
        $k_2$ is greater
    else if $k_{1\text{accidental}} = k_{2\text{accidental}}$ then
        pitches are in unison
    end if
end if
```

This may be best illustrated with a couple of worked examples. If we take $k_1 \leftarrow 9$ (G♭) and $k_2 \leftarrow 10$ (D♯) then, to calculate which is the highest pitch, we first find the name class. According to Equation 2.9, $k_{1\text{nc}} \leftarrow 2$ and $k_{2\text{nc}} \leftarrow 3$. We use the name class in Equation 2.11 to find the pitch order position: $k_{1\text{pop}} \leftarrow 4$ and $k_{2\text{pop}} \leftarrow 1$. Using Algorithm 1, we can see that $k_{1\text{pop}} > k_{2\text{pop}}$, therefore k_1 is the higher pitch.

Following the same procedure for two notes with the same name class: If $k_1 \leftarrow -4$

(D♭) and $k_2 \leftarrow 17$ (D𝄪) then $k_{1\text{nc}} \leftarrow k_{2\text{nc}} \leftarrow 3$ and $k_{1\text{pop}} \leftarrow k_{2\text{pop}} \leftarrow 1$. Following Algorithm 1, we see that we will need to calculate the number of accidentals for each pitch. So, using Equation 2.10, $k_{1\text{accidental}} \leftarrow -1$ and $k_{2\text{accidental}} \leftarrow 2$. Therefore $k_{1\text{accidental}} < k_{2\text{accidental}}$ and k_2 is the higher pitch.

Interval inversion is achieved by changing the sign of the interval value as in Equation 2.12.

$$i_{\text{inverted}} \leftarrow -(i) \tag{2.12}$$

The inversion of a minor 3rd is a major 6th

$$(-3)_{\text{inverted}} \leftarrow -(-3) = +3 \tag{2.13}$$

Calculating The Octave

It can be seen from Table 2.8 that the interval of the octave and unison both result in the an unchanged value of k. This presents some difficulty in distinguishing between notes which are in unison and notes which are 1 or more octaves apart.

By convention pitches in the name class range C to B are in the same octave. Since the spiral of fifths does not order notes in strict pitch order, the note value k must be transformed to its pitch order position k_{pop} as above.

Then when raising by interval i:

$$k_{1\text{pop}} < k_{2\text{pop}} \Rightarrow \text{octave} + 1 \tag{2.14}$$

When lowering by interval i:

$$k_{1\text{pop}} > k_{2\text{pop}} \Rightarrow \text{octave} - 1 \tag{2.15}$$

2.6.3 Scales

Scales can be generated from the sequence of intervals which make up the scale. For example, the major scale is generated from the sequence {major 2nd, major 2nd, minor 2nd, major 2nd, major 2nd, major 2nd, minor 2nd}. This equates to the interval sequence {+2, +2, -5, +2, +2, +2, -5}. To generate the scale: the starting note is chosen, the first interval is added to it and each additional interval is added to the previous accumulated result. This can be generalised to Equation 2.16[3].

$$k_m \leftarrow k_{\text{tonic}} + \sum_{n \leftarrow 0}^{m-1} I(n) \tag{2.16}$$

where k_m is the m^{th} note in the scale, k_{tonic} is the tonic of the scale and I is the sequence of intervals. Note that this equation does not describe the melodic minor scale which would be slightly more complicated.

The chromatic series for a given number of divisions of the octave can be generated by repeatedly raising the tonic by an augmented unison. For example, C $\leftarrow 1$, C♯ $\leftarrow 1 + 7 = 8$, C𝄪 $\leftarrow 8 + 7 = 15\ldots$ This quickly brings up the issue of calculating the enharmonic equivalent.

2.6.4 Enharmonic Equivalence

Inspecting Table 2.7 and following the 12-tone chromatic scale we can see that enharmonically equivalent notes are 12 steps apart. For example, C♯ $\leftarrow 8$ and its enharmonic equivalent in the 12-tone scale D♭ $\leftarrow 8 - 12 = -4$. A similar relation can be found for the 19-tone scale: A𝄪 $\leftarrow 18$ and its enharmonic equivalent B♭ $\leftarrow 18 - 19 = -1$, 19 steps apart. It is tempting to assume from this relation that in any scale of N divisions of the octave, the enharmonic equivalent of a note can be calculated as $k_{\text{equivalent}} \leftarrow k \pm N$. Unfortunately the generalisation is not quite so simple.

To find the enharmonic equivalent of a note in terms of the next note name above it, it is raised by a major 2nd and then reduced by the number of chromatic scale steps in a major 2nd. For example to find the enharmonic equivalent of E♯ $\leftarrow 12$ in the 12-tone scale, it is raised by a major 2nd $(12 + 2 = 14)$ and reduced by the number of chromatic scale steps in a major 2nd $(14 - (2 \times 7) = 14 - 14 = 0)$ which results in F♮. This relation is expressed in Equation 2.17:

$$k_{\text{enharmonic}} \leftarrow k \pm (i_{M2} - 7x) \tag{2.17}$$

[3]Indices into arrays are 0-based

where i_{M2} is the interval of a major 2nd and x is the number of chromatic scale steps in a major 2nd. Finding the enharmonic equivalent in terms of the name class above the current note equates to addition in the above equation and subtraction for the name class below.

The relation expressed in Equation 2.17 ($k \pm i_{M2} - 7x$) resolves to $k \mp 12$ for the 12-tone scale and to $k \mp 19$ for the 19-tone scale. (Where we previously used $+$, we now use $-$ and where we previously used $-$ we now use $+$). This allows the use of the Rem() function or modular arithmetic to keep the result of any transformation within the gamut of the chromatic scale note names (i.e., excluding 𝄪s and 𝄫s).

The expression in Equation 2.17 can be used with all scales generated from a single size of fifth. In the range from 12 to 19 notes per octave this includes 12, 17 and 19 notes per octave. The consequence of constructing a scale from a single size of fifth is that there is a single size for each interval of the scale: there are an equal number of chromatic scale steps for each interval no matter which note is the starting note. For example, Equation 2.18 relates the number of steps in an octave to the diatonic scale.

$$N \leftarrow x + x + y + x + x + x + y = 5x + 2y \tag{2.18}$$

where N is the number of notes in an octave, x is the number of chromatic scale steps in a major 2nd and y is the number of chromatic scale steps in a minor 2nd.

Equation 2.18 shows how the octave is generated from the diatonic scale and how the diatonic scale is generated from the major and minor 2nds. Substituting 2 and 1 for x and y respectively evaluates to $N \leftarrow 12$, 3 and 1 gives $N \leftarrow 17$ and 3 and 2 gives $N \leftarrow 19$.

Other scales in the range 12–19 notes per octave require an extended version of Equation 2.17 to find the enharmonic equivalent. As an example, let us use the 18 note equal-tempered (18ET) scale shown in Figure 2.2.

The major 2nd from D to E takes 3 chromatic scale steps whereas the major 2nd from B to C♯ takes 2 chromatic scale steps. Thus Equation 2.17 must be modified to find the enharmonic equivalent in 18ET, since there is no longer a single value of x for all ks.

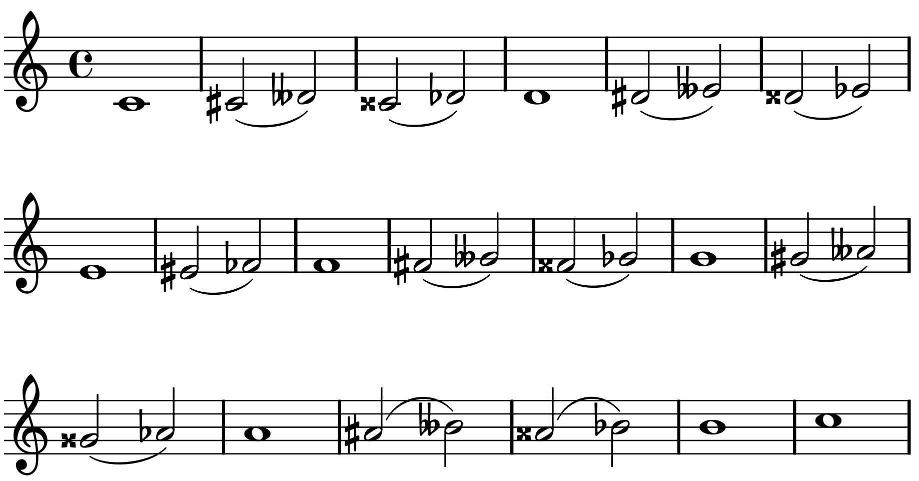


Figure 2.2: The 18 tone equal-tempered scale as defined in [47]. Tied notes indicate enharmonic equivalents.

Name Class	F	C	G	D	A	E	B
No. of steps	10	11	10	11	10	11	10

Table 2.9: The number of chromatic scale steps travelled in the 18ET cycle of fifths. For example, travelling a fifth from F to C takes 10 chromatic scale steps. Travelling a fifth from C to G takes 11 chromatic scale steps. This can be verified by counting notes in the 18ET chromatic scale shown in Figure 2.2.

The 18ET scale is constructed from 2 sizes of fifth. Counting the number of chromatic scale steps travelled in the 18ET cycle of fifths results in a repeating sequence of 7 intervals as in Table 2.9. Using the name class of the note (calculated from Rem(k, 7)) as an index into this array, the modified form of Equation 2.17 is:

$$\begin{aligned} k_{\text{enharmonic}} \;\leftarrow\; & k \pm \Big(i_{M2} - 7\big(X_{\text{scale}}\big[\text{Rem}(k,7)\big] \\ & +X_{\text{scale}}\big[\text{Rem}(k \pm 1, 7)\big] \\ & -N\big)\Big) \end{aligned} \tag{2.19}$$

where X_{scale} is the sequence of chromatic scale steps in the cycle of fifths of *scale* such as the one given in Table 2.9. As in Equation 2.17, addition is used to find the enharmonic equivalent in terms of the name class above and subtraction for the name class below.

So we have seen how to construct the spiral of fifths and use it to create unique identifiers for each pitch. We have seen how to add and subtract intervals from pitches, how to calculate octaves, how to construct scales and find the enharmonic equivalent. This pitch representation is interval invariant like the Base-X and Binomial systems. Unlike the Base-X system, it has no 'holes' of ambiguous pitch and unlike the binomial system, it uses the same representation for pitches and intervals in 12-tone scales and 19-tone scales. But what about microtonal scales with more divisions of the octave? What we have seen so far is the special case. To represent all diatonic scales, including microtonal scales, we need to make slight adjustments to the calculations presented so far.

2.6.5 Beyond 19 Divisions of the Octave

The scales mentioned so far take their note names from the gamut of names including seven ♯s, seven ♮s and seven ♭s. The maximum number of notes in a diatonic scale that can be represented by these note names is 19: $7 \times 3 = 21$ less 2 enharmonic equivalents. To represent scales with more divisions of the octave, we must introduce accidentals which lie between the accidentals already mentioned. The new accidentals form a separate spirals of fifths which can be interleaved with the existing spiral of fifths representation. For example, we can add 1 accidental between each semitone and give them the labels: semisharp(𝄲)[4], sequisharp (𝄰)[5], semiflat (𝄳) and sesquiflat (𝄳♭). We re-number the notes in the table as in Table 2.10.

The ♮s, ♯s, ♭s,𝄪s, and 𝄫s of the previous table (Table 2.7) now lie on the even numbers - F♮ $\leftarrow 0$, C♮ $= 2$, G♮ $= 4$ etc. In between them lie the additional accidentals: 𝄲s, 𝄳s, 𝄰s and 𝄳♭s. The additional notes are aligned to preserve the 7-step chromatic series property: adding or subtracting 7 steps moves around the chromatic scale. For example, starting at F♮ $= 0$, F♮ $+ 7 \rightarrow$ F𝄲, F𝄲 $+ 7 \rightarrow$ F♯, F♯ $+ 7 \rightarrow$ F𝄰.

The other calculations mentioned so far will no longer work with this extended representation. However, they can be retained with little modification. We introduce an additional parameter z which stores the number of chromatic steps in a semitone. In Table 2.10, for example, $z = 2$.

[4]Or half sharp.

[5]Or sharp and a half.

	-/𝄫/𝄳♭	𝄳♭/♭/𝄳	𝄳/♮/𝄲	𝄲/♯/♯𝄲	♯𝄲/𝄪/-
F	-28	-14	0	14	28
A	-27	-13	1	15	29
C	-26	-12	2	16	30
E	-25	-11	3	17	31
G	-24	-10	4	18	32
B	-23	-9	5	19	33
D	-22	-8	6	20	34
F	-21	-7	7	21	35
A	-20	-6	8	22	36
C	-19	-5	9	23	37
E	-18	-4	10	24	38
G	-17	-3	11	25	39
B	-16	-2	12	26	40
D	-15	-1	13	27	41

Table 2.10: The Spiral of Fifths pitch representation for scales with 2 divisions to the chromatic semitone. The notes of the previous table (Table 2.7) now lie on the even numbers at twice their previous value. Between them, on the odd numbers, lie the notes of the interleaved spiral of fifths. The new spiral of fifths is aligned to preserve the 7-step chromatic scale property.

For transposition:

$$k_{\text{transposed}} = k + iz \tag{2.20}$$

To calculate whether the octave boundary has been crossed we first convert the value of k to one in the range $\{0...6\}$ representing the cycle of fifths' name classes i.e., {F, C,...,B}. (Previously we used $\text{Rem}(k, 7)$ to achieve this.)

So:

$$k_{\text{nc}} = \frac{\text{Rem}\Big(k + 7\big(\text{Rem}(k, z)\big), 7z\Big)}{z} \tag{2.21}$$

where k_{nc} is the name class of k in the spiral of fifths order {F,C,...,B} and $z > 1$.

and $k_{\text{accidental}}$ is calculated thus:

$$k_{\text{accidental}} = \frac{k - zk_{\text{nc}}}{7} \tag{2.22}$$

where $k_{\text{accidental}}$ now represents the number of accidentals above the natural in the extended representation. Where previously if $k = 8$ (C♯), $k_{\text{accidental}} = 1$ i.e., 1 accidental over the natural, if $z = 2$ and $k = 16$ (C♯) then $k_{\text{accidental}} = 2$ that is: a 𝄲 and a ♯ above the natural. The calculation of the pitch order and the method of comparison of two pitches remains unchanged.

Scales can be generated using the same arrays of intervals as used previously:

$$k_m = k_{\text{tonic}} + z \sum_{n=0}^{m-1} I[n] \tag{2.23}$$

The calculation of enharmonic equivalence remains largely unchanged from Equations 2.17 and 2.19. Only the interval of an major 2nd is replaced with zi_{M2}.

$$k_{\text{enharmonic}} = k \pm (zi_{M2} - 7x) \tag{2.24}$$

and

$$\begin{aligned} k_{\text{enharmonic}} &= k \pm \Big(zi_{M2} - 7\big(\mathrm{X}_{\text{scale}}\big[\mathrm{Rem}(k,7)\big] \\ &\qquad + X_{\text{scale}}\big[\mathrm{Rem}(k \pm 1, 7)\big] \\ &\qquad - N\big)\Big) \end{aligned} \tag{2.25}$$

So this extended spiral of fifths representation can be used to represent all diatonic scales whether tonal or microtonal. Pitches are represented by a fraction consisting of the spiral of fifths position (k) as the numerator and the number of divisions per semitone (z) as the denominator. For example, C♮ is expressed as either $\frac{1}{1}$ with 1 division of the semitone (as in Table 2.7) or $\frac{2}{2}$ with 2 divisions of the semitone (as in Table 2.10). Similarly, intervals can be represented as a fraction consisting of the interval value i as a numerator and the number of divisions of the semitone z as the denominator. The musical operations can be performed taking care to replace the z value with the number of divisions per semitone.

2.7 Summary

In this chapter we aimed to investigate pitch representations which could fulfil 2 of the requirements given in the previous chapter: to represent music in a musically meaningful way and to enable representation of microtonal scales. We looked at the MIDI representation and found that, whilst it correctly encodes increasing pitch, it could not distinguish between enharmonically but functionally distinct pitches. We extended the representation to provide distinct pitch names but found it was problematic as it did not have the property of interval invariance.

We went on to look at two representations which did achieve interval invariance: Base-X and the Binomial representation. The Base-X representation suffered from 'holes' of ambiguous pitch and the binomial representation had a different version for each number of divisions of the octave.

A new extended spiral of fifths representation was introduced which achieved interval invariance, had distinct names for pitches without holes of ambiguous pitch and used the same representation for all diatonic tonal and microtonal scales. A set of musical operations were outlined which can form the basis of more complex musical score analyses.

This pitch representation is the first part of achieving the aim of the current system which is to create a system which enables analysis of both score and performance data.

Chapter 3

Time

This thesis aims to create a system which is capable of storing music and performance in a computer in a form which makes it available to multiple queries. In the introduction, we saw that it is necessary to use an appropriate representation for music in a computer if the system is going to be used for diverse queries. In the previous chapter we looked at the difficulties in representing pitch. In this chapter we will look at the issue of representing time.

Since the system described here must represent both score and performance data, the current work uses two different time lines: the time line of a score (score time) and the time line of a performance (real time). The way in which one time line is mapped onto the other is part of the interpretation which constitutes a musical performance. For analysis to take place the different representations must be combined along this common axis. There must be mappings between the representations to allow significant data from one source to be located in another.

Musical time also has its own logic: events can occur before, during or after other events; events can occur simultaneously or sequentially. This *temporal logic* creates relations between events and provides some structure for the music. These relations form the creative palette for rhythm and tempo.

It is beyond the scope of the current work to provide a full treatment of the fields of temporal logic or rhythmic analysis. Instead, this work aims to provide the necessary primitives and basic operators which will enable others to investigate these areas more thoroughly.

This chapter describes a representation for musical score time, performance time and the mapping between them. In the next chapter we will see some basic operations on this time representation. In the Results chapter (Chapter 6) we will see these operators used in a query. In the Discussion (Chapter 7 we will discuss how these operators can be combined to create the operators of temporal logic.

3.1 The Requirements of a Time Representation

In Section 1.4, we outlined the requirements for the current system. These included the requirements for representing the score and representing the performance and representing both in a way that will enable analysis. The previous chapter addressed the requirements for representing pitch and this chapter addresses the requirements for representing time.

In [73] some of the main difficulties in representing time is examined:

- **Tacit vs implicit vs explicit** In a time tacit representation there is no reference to time; there is only the present. An example of this would be the messages which are sent down a MIDI cable - they must be acted upon immediately to preserve the correct timing of the performance. Implicit time is *implied* by the order in which the notes appear whereas explicit time states the position along a time axis.
- **Points vs intervals** Primitives that are represented as points exist as single events on the time-line such as the note-on and note-off messages of the MIDI protocol. Interval primitives have a duration such as 'minim', 'crotchet' etc.
- **Absolute vs relative** The time-base can be represented as either absolute values such as 300ms or as relative values such as $\frac{1}{8}$th of a whole note.
- **Discrete vs continuous** Discrete time is the representation of events quantised to a time grid whereas continuous time represents events as a function of a continuous variable.
- **Explicit time structuring** Events in a music representation can be structured

by grouping all events for a single voice or instrument into one stream, with the streams for other voices following after (horizontal time-slicing). The alternative is to group all events which occur at the same time together, regardless of the voice (vertical time-slicing).

- **Declarative vs Procedural** Structures can be merely stated as existing in the representation (declarative) or precise instructions for creating them included. For example, a trill could be declared or could be described at the level of the individual note.

Many representations use an implicit measure of time. For the purpose of storage and interchange (e.g., MusicXML), where the music will only parsed over once, it is reasonable to use an implicit representation of time since the application only has to maintain a single counter of the current location. However, for score analysis, where the music will be parsed over several times, an explicit measure of time is preferable (such as in **kern). This allows locations in the music to be found without having to parse the data from the beginning each time.

Point primitives tend to occur in performance representations (such as GDIF [66], GMS [88] and MIDI) where raw sampled data or event-based representations are used. There is no reason why interval primitives cannot occur in performance representations (as they do in PML) but they would tend to occur at the level above raw data — that is the aggregation/segregation layer in Figure 1.1 — where parts of the data are aggregated into gestures. There are few representations which do this.

Interval primitives are used more than point primitives in score representations where the most basic primitive is often the note comprised of a pitch and a duration. This underlines the unsuitability of MIDI as a score representation as its event-based representation is at odds with the concept of a note as a single entity.

The absolute versus relative time distinction occurs again mainly between performance representations (absolute or real time) and score representations (relative time). Representing a score in relative time allows for the separation of note durations from the overall tempo of the piece. Combining the note durations with a tempo in beats per minute allows an approximate real time duration for each note to be calculated. This will be altered greatly by the performer during their performance.

The distinction between discrete time and continuous time is usually, but not always, one of the distinction between score representations and performance representations. Score representations tend to use discrete values for durations and locations to match those used in Common Music Notation. Performance representations tend to use continuous time as this best reflects the real time axis of performance. Exceptions do exist such as MIDI which is a performance representation but uses a discrete time representation of 96 clicks per note. Computer music composition programs may use a real time axis for describing musical time.

The inherent 1-dimensionality of a data stream leads most representations to choosing either horizontal time structure or a vertical time structure i.e., either each channel or voice is encoded time-ordered, one after the other or each time instant is encoded with each channel or voice's data grouped together. Regular sampled data can get around this limitation by structuring the data in a vertical time structure and ensuring that each datum takes the same amount of space. Thus a regular offset will reveal each datum in a particular channel. This interlaced approach is used in RIFF (wav) audio and GMS files.

For non-sampled data such as a musical score, where data are not organised at regular intervals, such an approach will not work. The solution has been to create two representations: one horizontally sliced and one vertically sliced. Standard MIDI Files use two formats — one horizontally time-sliced (Format 1) and one vertically time-sliced (Format 0). MusicXML defines 2 formats — `score-partwise` for horizontal slicing and `score-timewise` for vertical slicing.

Musical score analysis requires that the data be available in both forms — horizontal for melodic analysis and vertical for harmonic analysis. Analysis data structures such as the **kern format and the doubly-linked list structure described in [41] allow for score data to be sliced both ways.

The distinction between declarative and procedural representations depends on the purpose of the representation. Storage and interchange formats tend to be declarative: they use primitives from a predefined set. Programmed music tends to be procedural. An interesting hybrid is found in the Lilypond typesetting system. For most music, the required set of primitives is already defined and is made available through the TEX-like markup. However, a file can include scheme code alongside the markup to draw extra symbols or manipulate the score. This greatly increases the flexibility of the system for representing non-standard notation. It does limit the usefulness of the representation for other purposes as one cannot know the musical meaning of a piece of scheme code to draw lines and dots on a page.

So there are differing requirements for representing time depending on whether it is score time or performance time being represented and what the goal of the representation is. A score representation which will be used for analysis should be explicit in its representation of time to allow easy movement through the data; the time primitives should be relative, discrete intervals and declarative to closely match those in the score; and time should be structured so that it can be accessed horizontally and vertically.

At its lowest level, performance data will consist of points (often samples) placed on an absolute, continuous time scale. At this low level of regularly sampled data, we have seen that it is relatively simple to change between horizontal and vertical time structuring. This is the data level of the 3 gesture levels in Figure 1.1. At the higher level of aggregation/segregation, the data becomes intervallic rather than point-based. At the highest level, the information level, time may require a procedural representation rather than the declarative approach used so far as many data aggregations are composed into a meaningful gesture. Since the current work only deals with performance data at the 2 lower levels, we will not treat the representation of time at the information level here.

3.2 MusicXML Time

In Chapter 1, we saw the various approaches to representing music and decided that MusicXML provided sufficient information for representing the logical domain of music and that its wide adoption as an interchange format made it a good candidate for a model for representing scores. The previous section has outlined some of the requirements for a representation of time in scores and performances. So how is time represented in MusicXML and can we adopt this model unchanged for the current application? Like many representations, MusicXML chooses to imply time.

In MusicXML [68] the onset location of a note in a score is implied by the sum of the preceding notes in the score. This alone would make it necessary to parse from the beginning of a file to find any location in it. However the situation is further complicated by the inclusion of the `<backup>` tag which can occur at any point in a file and operates as a 'rewind'-like function to move the following notes' locations back in the file by a given number of steps. This requires that the entire file must be parsed, and the ordering of the notes rebuilt, before the data held at any location in the file can be known. It should be noted that the MusicXML file format was not created for the purpose of performance analysis but for the interchange of music notation.

MusicXML durations are encoded as a pair of numbers representing the numerator (`duration`) and denominator (`divisions`) of a fraction of a crotchet beat. The denominator is given once at the beginning of the file so that only the numerator is included for each note's duration.

MusicXML separates the note's performed duration from the glyph used to represent it on the printed page. For example, a dotted crotchet would have a `duration` of $\frac{3}{2}$ to indicate its performed duration and both a quarter `type` and a `dot` tag to indicate a dotted-quarter note (dotted crotchet) should be used in the printed score.

So MusicXML time is relative, discrete and declarative which matches the score notation it is representing. Times are implicit rather than explicit and time is structured either horizontally or vertically. In order to satisfy the requirements in Section 3.1 above, time would need to be made explicit and would need to enable easier movement between horizontal and vertical structuring.

3.3 Spoff Time

The representation of score time used in the current work extends that of MusicXML to make timing explicit. It uses the same fraction-of-a-crotchet time-base used by MusicXML. From here it will be referred to as Spoff time — after the spiral-of-fifths pitch representation outlined in the previous chapter.

We define a Spoff time instant as a fraction of a crotchet and use the same representation for note onsets and durations. Every note is given a duration and, unlike MusicXML, an onset time relative to the beginning of the piece. This makes the time representation explicit rather than implicit using interval primitives rather than points.

Bar (or measure) numbers are deliberately ignored when representing score time in Spoff time. Bar numbers are a useful method of defining location in musical scores. One could devise a system which defined a location as the bar number combined with the number of beats from the start of the bar. However such a system presents problems. Music that has a changing time signature or music with two different, simultaneous time signatures (polychronous music) would be difficult to traverse using a representation which relied on bar numbers for note locations. For example, a piece written with one voice in $\frac{3}{4}$ and another in $\frac{4}{4}$ would, after 12 beats, be in bars 4 and 3 respectively. Thus a timing system which uses bar numbers for locations increases complexity over a beat-based location system.

We have addressed the need for explicit timing here by creating Spoff time as an extension of that used in MusicXML. We have not addressed the need for easy movement between horizontal and vertical time structuring. This will be addressed in Chapter 5 where the facility to alternate between horizontal and vertical time structuring is provided ‘for free’ by the database query language.

3.4 Performance Time

In Chapter 1, we formulated the requirement that the current system should store performance data and make it available to many diverse queries. Additionally, the requirements in Section 3.1 above state that performance time should be represented on an absolute and continuous time scale featuring point-based primitives for low-level data and intervallic primitives for aggregations. This section will look at how these requirements are addressed in the current system.

Since all performances have the same time base in common, performances are represented in real time, that is, seconds. This allows direct comparison between performance data that may have different sources or have been recorded at different sample rates.

Sampled data is positioned on a real time axis with each sample given a time as a decimal fraction of a second. Segmented performance data is given a start time and duration in the same format. This reflects the intervallic structure of performance segments.

In this way we have used an absolute and continuous time line for all performance data without regard to sample rates which might be different for different sources of performance data. By using the same time base for all sampled data, we make it easier to query data from different sources.

3.5 Score-Performance Mapping

The current system aims to enable the analysis of musical performance by making both score and performance available for querying. It aims to allow new questions to be asked of music by allowing analysis of music and performance data in the same place rather than in separate systems. For this analysis to take place, it is not enough to represent the two data streams isolated, a mapping between data in one domain and the other must be created.

Dannenberg has been successful in this field, presenting a monophonic matcher [108] and then a polyphonic matcher [82] using dynamic programming[1]. Later, in 1993, Large [60] used dynamic programming for score-performance matching to investigate errors in piano performances. Hoshishiba [120] added post-processing to the dynamic programming technique to improve the results.

The current work uses tools developed by Douglas McGilvray at the University of Glasgow to perform score-performance matching. The software (described in [58]) uses an algorithm based on Dannenberg [108]. The algorithm uses dynamic programming to find the optimal correspondence between two sequences. The algorithm

[1]dynamic programming aims to solve a complex problem by breaking the problem into a series of small steps which can be solved recursively.

includes novel extensions to allow matching of polyphonic music.

3.6 Temporal Logic

In this chapter, so far, we have dealt with the representation of musical time primitives and the correspondence between two time lines of these primitives. Within these time lines there are relationships between the musical time primitives. We use phrases such as 'during' or 'after' to describe the relationship between notes. This is known as the temporal logic.

An excellent introduction to the issues surrounding temporal logic and music can be found in Alan Marsden's “Representing Musical Time: A Temporal-Logic Approach” [40]. In [39] Marsden created a system for investigating the processes in music theory. The system was programmed in the Prolog logic programming language and was able to describe music using a relatively simple set of operators such as `precedes(X, Y)` (asserting that note X occurs before note Y) and `duration(X, D)` (asserting the duration of X).

In [91], Bergeron and Conklin described a set of logical operations which encode some of the logic of musical time. The temporal relations described were: notes start together, note starts during another note and notes end together. The work included relations to describe vertical and horizontal (melodic) intervals as well. This set of relations was used to analyse 185 Bach chorale harmonisations for suspensions, parallel fifths and melodic patterns at cadences.

Of particular relevance to the present application is the use of temporal logic in databases. In Chapter 5 we will outline the use of databases for the current application. However it is worth looking at the use of temporal logic in databases here.

Richard T. Snodgrass has written extensively on the subject of temporal databases [106] having led the creation of Temporal SQL (TSQL2) [107], the temporal extension to the SQL database query language. A temporal database has been used for a system which facilitated collaborative writing. In [61], a temporal database was used to implement a temporal data model which modelled the lifetime of a document as it was edited by different people.

In [83], Aberer and Klas outlined temporal extensions to an object-oriented database for representing multimedia such as video and audio. They described operations such as timed events, sequential compositions (where events occur one after another) and parallel compositions (where events occur at the same time). By combining these operations, they were able to model temporal multimedia compositions.

In this section we have seen some of the operations needed to realise the temporal logic of music. The use of temporal logic in music and multimedia applications has been shown. The use of temporal databases in this field seems to be limited. The main focus of research in temporal databases seems to be in creating more reliable and robust database systems and the technology does not seem to have been applied to music representation.

3.7 Summary

The current system aims to represent musical data from scores and performances. To do this it must represent two different time lines: score time and performance time. In addition, to allow analysis of both, it must include a mapping between the two time lines.

In this chapter we have investigated the requirements for representing musical time. Score time should be explicit, relative, discrete, declarative and be structured both horizontally and vertically. Performance time will consist of points at its lowest level and intervals at the aggregation/segregation level. An extension of MusicXML time was proposed for score time and a common time-base of seconds was chosen for performance time with mappings between the two timelines provided by external software.

The relation between timed musical events has been investigated with researchers using temporal logic to describe these relations. The potential for temporal databases to be of use in this field seems underexploited.

The representation of time presented in this chapter, along with the representation of pitch introduced in the previous chapter, provide the basis for the music representation used in the current work. The next chapter will formalise how these representations are combined and how they are manipulated by musical functions to enable computational music analysis.

Chapter 4

Musical Functions

The aim of the current work is to create a system capable of querying musical scores and performance data inside a computer to produce analyses which provide insight into the process of musical performance. In the first chapter we looked at other computer systems which manipulate musical data and used this overview to create a list of requirements for the current system. In the previous two chapters methods of representing pitch and time have been shown which aim to fulfil some of these requirements.

In this chapter we define a set of musical types for describing symbolic music and the algorithms which manipulate them. The musical types combine the representations described above. The purpose of these types and functions is to enable the analysis of the musical data by providing an interface to the underlying data through the expression of musical queries. This interface intends to fulfil the requirements to allow analysis of musical data and provide access to the data in a way which can be understood by programmers and computer-literate musicologists.

To make the algorithms easier to read, once a function is defined, it is thereafter referred to using its operator. For example, once the `addInterval` function is defined, it is referred to with a '+' symbol.

4.1 Basic Musical Types

In order that we may perform music analysis on the scores stored in the computer, we need to define a core set of data types (these could be primitives or objects) which we will manipulate. These datatypes are not intended as a replacement for a score and do not presume to be a complete representation of a musical work. They are however sufficient for a good deal of musical analysis and form the basis of more complex queries.

The types described here are similar to those described by Wiggins. In [123], Wiggins describes abstract types for pitch and time. These are combined in tuples to create events — synonymous with notes here. Events are grouped into constituents which are synonymous with note-groups here. Wiggins' abstract data model has been implemented in the AMuSE system [76] for music analysis and Wiggins used it himself to implement algorithms for rhythm and similarity analysis.

4.1.1 Pitch

The full requirements for a pitch representation can be found in Chapter 2. It is worth repeating here that a pitch representation ought to be able to distinguish between pitches regardless of tuning and enharmonic equivalence, should be able to extract name class, should be able to be ordered in pitch height order, should be able to extract intervals from pitches and add intervals to pitches and finally should represent octaves.

Additionally there are more practical requirements that a representation should not be taxing on the processor of the computer that is manipulating it and should not require large amounts of storage space.

The work presented here has the extra requirement that the pitch representation should be capable of doing all of the above with microtonal tuning systems. The spiral of fifths pitch representation outlined in Chapter 2 satisfies all of these criteria so has been chosen for the current work.

In addition to storing the pitch in the spiral of fifths representation, it is necessary to store 2 further parameters for each pitch: the number of divisions per semitone (z in Chapter 2) and the octave. Though it may be possible (and more space preserving) to store the parameter z once for an entire score, the current implementation stores the parameter alongside the pitch as it allows the musical functions described in Section 4.2 and later the queries to be simplified. Since the spiral of fifths representation does not preserve pitch order it is necessary to separately store the octave of each pitch. Thus a complete pitch is stored and processed as a triple:

$$p := (k, z, o) \tag{4.1}$$

where k is the pitch in the spiral of fifths representation, z is the number of divisions per semitone and o is the octave with 4 representing the octave beginning on middle-C [124].

4.1.2 Interval

As soon as one begins manipulating pitch one soon finds the need to represent an interval. An interval representation ought to give unique representations to all intervals, to be able to extract the class of an interval[1], to be able to extract the interval type[2], to be able to invert intervals and combine them into scales.

The interval is represented in an almost identical way to pitch: a triple which stores the interval in the spiral of fifths representation, the number of divisions per semitone and the number of octaves this interval traverses.

$$v := (i, z, o) \tag{4.2}$$

where i is the interval in the spiral of fifths representation and z and o are the same as in Equation 4.1.

[1]The class of an interval is given by the number of diatonic steps i.e., all thirds share the same class whether major, minor, augmented or diminished

[2]The type of an interval is given by the number of diatonic steps and the modifier i.e., major, minor etc. It ignores octaves so a major 10th has the same type as a major 3rd

4.1.3 Time

The full requirements for the representation of time can be found in Chapter 3. A time representation ought to allow for adding and subtracting time, should use the same representation for note onsets, offsets and durations, should be independent of time-signatures, meters and measures and should be capable of representing any division of a note.

The representation chosen here matches that of MusicXML in that it represents time as a fraction of a crotchet. However, where MusicXML stores the denominator of the fraction globally for an entire piece, here we store the denominator with each fraction numerator. This allows some simplification of the functions in Section 4.3 and later in queries.

Thus time is represented as a tuple:

$$t := (n, d) \tag{4.3}$$

where n is the numerator and d is the denominator of the fraction of a crotchet represented by t.

4.1.4 Notes

We define notes as those units of music which have pitch, a duration and a start time. We use the representations for pitch and time introduced above along with a unique identifier for each note to create a tuple representing a score note:

$$s := (u, p, t, d) \tag{4.4}$$

where u is the unique identifier for this note s, p is the pitch triple, t is the start time of the note (expressed as a time tuple) and d is the duration (expressed as a time tuple). It is important to note that the duration represented here does not necessarily match the duration indicated by the notehead. The duration of the note holds the duration after all modifiers have been taken into account: such as dots, tuples etc.

Unlike Wiggins' abstract data model in [123], we do not include a component to describe timbre. The inclusion of timbre inside a note tuple in Wiggins' data model could have originated in the MIDI data model. MIDI includes a velocity value with each note event. This makes sense in a performance representation where intensity will be measured for each note event. However, this section describes a representation for musical scores. In music notation, dynamic instructions such as *crescendo* or *pianissimo* apply to groups of notes rather than individual notes. For this reason, in the current application, timbre will be indicated by note groups described below.

4.1.5 Note Groups

Since the note description given above is incomplete in representing much of the additional information contained in a score, we implement note groups to reflect notes membership of certain categories such as membership of a chord or the presence of a key signature. The note group also provides the means to implement analysis groups which can overlap.

$$g := (u, y, v, m) \tag{4.5}$$

where u is the unique identifier for this group g, y is the type of this group, v is a list of values for this group (for example, for a key signature group, v would hold the key) and m is the list of members of the group.

In Wiggins' abstract data model [123], the equivalent groupings to note groups are called constituents. Constituents comprise of a unique identifier, a label describing their type and a set of particles. Particles can be notes or other constituents. By allowing constituents of constituents, Wiggins allows for the creation of hierarchical structures which the model described above does not. Hierarchical structures have been shown to be a useful method of organising music for analysis [117], [63]. A small modification to the current data model which allows hierarchical structures will be described in the Discussion (Chapter 7.

4.2 Pitch Functions

The basic musical types outlined above allow us to represent notes consisting of a pitch, start time and duration and note groups to combine notes together which share a common membership such as key signature. In order to manipulate the note representation, and create the basis for more complex processes, we define a set of functions which operate on the note tuple.

Previous work in this area includes that of Harris *et al* [70]. In this work, the necessary functions to manipulate notes are adding and subtracting intervals from pitches, extracting the interval from two pitches, adding and subtracting durations to time instances and extracting the duration from two time instances. Harris *et al* also indicate that ordinal operators (i.e., greater-than $>$, less-than-or-equal-to $>=$ etc.) will be needed.

In this section, we show the algorithms which implement the functions outlined in Harris *et al* for the current work's pitch representation. In the next section we will see the algorithms for manipulating time followed by musical intervals. In addition to those in Harris *et al* , we show the algorithms to implement the different types of equality which arise from considering music. These include, for example, the equality where two pitches are exactly the same (in unison) and the equality where two pitches are the same but in different octaves.

4.2.1 Ordinal Operators

The ordinal operators are used here to place pitches in pitch height order (not spiral of fifths order) and therefore require the pitches being compared to first be converted to pitch height order. The ordinal operators are less than ($<$), less than or equal to ($<=$), greater than ($>$) and greater than or equal to ($>=$). The algorithm for the less than ($<$) operator for pitch is shown in Algorithm 2.

Algorithm 2 Algorithm to compare two pitches (p_1 and p_2) and return true if p_1 is lower in pitch order to p_2.

```
1: function LessThanPitch($p_1$, $p_2$)
2:     if $p_1(o) < p_2(o)$ then                              ▷ $p(o)$ is octave of pitch $p$
3:         return True
4:     else if $p_1(o) > p_2(o)$ then
5:         return False
6:     else if $p_1(k_{\text{pop}}) < p_2(k_{\text{pop}})$ then▷ $p(k_{\text{pop}})$ is the pitch order position of pitch
       $p$ calculated with Equation 2.11. Assumes pitches are in the same octave
7:         return True
8:     else if $p_1(k_{\text{pop}}) > p_2(k_{\text{pop}})$ then
9:         return False
10:    else if $p_1(k_{\text{pop}}) = p_2(k_{\text{pop}})$ and $p_1(k_{\text{accidental}}) < p_2(k_{\text{accidental}})$ then   ▷ Name
       class is the same but $p_1$ is flatter than $p_2$
11:        return True
12:    else
13:        return False
14:    end if
15: end function
```

4.2.2 Equality

When considering equality between pitches one must not only consider pitches in unison as equal but also those an octave apart as also differently equal. We therefore define 2 types of equality between pitches in Algorithm 3.

Algorithm 3 Algorithm to calculate two types of equality between pitches: unison and ignoring octaves.

```
function equatePitch($p_1$, $p_2$)
    if $p_1(o) = p_2(o)$ and $\frac{p_1(k)}{p_1(z)} = \frac{p_2(k)}{p_2(z)}$ then        ▷ Pitches are in unison
        return True
    else
        return False
    end if
end function

function approxEquatePitch($p_1$, $p_2$)
    if $\frac{p_1(k)}{p_1(z)} = \frac{p_2(k)}{p_2(z)}$ then        ▷ Pitches are the same, ignoring octaves
        return True
    else
        return False
    end if
end function
```

The comparison of pitches must take account of the number of divisions per semitone which the pitch is represented in. For example, consider the pitch F♯4. It can be represented as $\{7, 1, 4\}$ with 1 division per semitone or as $\{14, 2, 4\}$ with 2 divisions per semitone (refer to Tables 2.7 and 2.10). Clearly comparing these two triples' k value alone would lead to incorrectly concluding that they are not equal. Therefore it is necessary to compare the pitches in the form $\frac{k}{z}$. Additionally one must realise that this is not integer division as in Chapter 2. Consider the comparison between A♮3 = $\{15, 2, 3\}$ and F♯3 = $\{14, 2, 3\}$. If we were to use integer division we would incorrectly conclude that the pitches were equal thus we should use normal division here.

4.2.3 Adding and Subtracting Intervals

To add or subtract an interval is superficially simple: all one has to do is add or subtract the pitch value and the interval value. However, in practise one must take account of the transformation crossing the octave boundary. For example, the transformation of B♮3 to C♯4 by the addition of a major 2nd crosses the octave boundary i.e $\{6,1,3\}+\{2,1,0\}=\{8,1,\mathbf{4}\}$. The complete algorithms for addition and subtraction are given in Algorithm 4.

Algorithm 4 Algorithm to add a pitch and an interval

```
function addInterval($p$, $v$)
    $\frac{p_{new}(k)}{p_{new}(z)} \leftarrow \frac{p(k)}{p(z)} + \frac{v(i)}{v(z)}$
    $p_{new} \leftarrow \{p_{new}(k), p_{new}(z), p(o)\}$
    if $p_{new} < p$ then                              ▷ If resulting pitch is lower
        $p_{new}(o) \leftarrow p_{new}(o) + 1$                   ▷ increment the octave
    end if
    $p_{new}(o) \leftarrow p_{new}(o) + v(o)$
    return $p_{new}$
end function

function subtractInterval($p$, $v$)
    $\frac{p_{new}(k)}{p_{new}(z)} \leftarrow \frac{p(k)}{p(z)} - \frac{v(i)}{v(z)}$
    $p_{new} \leftarrow \{p_{new}(k), p_{new}(z), p(o)\}$
    if $p_{new} > p$ then                             ▷ If resulting pitch is higher
        $p_{new}(o) \leftarrow p_{new}(o) - 1$                   ▷ decrement the octave
    end if
    $p_{new}(o) \leftarrow p_{new}(o) - v(o)$
    return $p_{new}$
end function
```

4.2.4 Pitch Difference

Now that we have introduced how to add and subtract intervals we must see how the inverse is achieved: that is how to extract an interval from two pitches. Since intervals are always quoted from the lower pitch to the higher pitch we must first establish the lower pitch. Once this is achieved we subtract the lower pitch triple from the higher pitch triple to give the interval. Next we check if we have crossed the octave boundary and adjust accordingly. Finally we subtract the octave. The complete algorithm is given in Algorithm 5.

Algorithm 5 Algorithm to compute the interval between two pitches

```
function lowest($p_1$, $p_2$)                    ▷ returns the lowest of the two pitches
    if $p_1 < p_2$ then
        return $p_1$
    else
        return $p_2$
    end if
end function

function highest($p_1$, $p_2$)                   ▷ returns the highest of the two pitches
    if $p_1 > p_2$ then
        return $p_1$
    else
        return $p_2$
    end if
end function

function getInterval($p_1$, $p_2$)
    $p_{lowest} \leftarrow \text{lowest}(p_1, p_2)$
    $p_{highest} \leftarrow \text{highest}(p_1, p_2)$
    $\frac{v(k)}{v(z)} \leftarrow \frac{p_{highest}(k)}{p_{highest}(z)} - \frac{p_{lowest}(k)}{p_{lowest}(z)}$
    $v(o) \leftarrow p_{highest}(o) - p_{lowest}(o)$
    return $v$
end function
```

4.2.5 Generate Scale

In order that we might distinguish between notes that belong in a key signature and those which do not, we need to be able to generate the scale of pitches and test for a pitch's membership of that scale. We construct a scale by following a sequence of interval transformations. In its initial form, this sequence will give us the major scale. If we start on the 5th interval of the sequence we get a harmonic minor scale[3]. Starting at other places in the sequence gives us the church modes. Thus we construct a sequence of pitches by transforming the initial pitch by each of the intervals in turn.

Algorithm 6 Algorithm to generate a scale

```
function scale($k$, $m$)$\triangleright$ where $k$ is the pitch of the starting note and $m$ is the mode (for the major scale $m \leftarrow 0$ and for the minor scale $m \leftarrow 5$)
    $V_{\text{scale}} \leftarrow [v_{M2}, v_{M2}, v_{m2}, v_{M2}, v_{M2}, v_{M2}, v_{m2}]$
    $K_{\text{scale}}(1) \leftarrow k$
    $i \leftarrow 2$
    for all $v \in V_{\text{scale}}$ do
        $K_{\text{scale}}(i) \leftarrow K_{\text{scale}}((m + i - 1) \bmod 6) + v$
        $i \leftarrow i + 1$
    end for
    return $K_{\text{scale}}$
end function
```

4.2.6 Scale Membership

To allow us to check whether a pitch belongs in a particular key we must define a function to compare a pitch against each member of a scale. Furthermore we can generalise the function to compare any pitch against any array of pitches. Finally, we must define two versions of this comparison function: one to check for exact matches and one to check for approximate matches i.e., ignoring octaves.

4.3 Time Functions

Since we have defined a time type (in Section 4.1.3) we must also define how to manipulate it. Of primary importance when manipulating time is to place events in time order – to calculate whether an event occurred before another – hence we define the ordinal operators for the time tuple. We will also wish to calculate offset

[3]For the melodic minor, which alters by one pitch on the descending portion, a more complicated algorithm would be needed. This is discussed in Chapter 7

Algorithm 7 Algorithms to check a pitch's membership of a set of pitches

```
function elementOfPitchArray($k_{\text{test}}$, $K$)
    for all $k \in K$ do
        if equatePitch($k_{\text{test}}$, $k$) then
            return True
        end if
    end for
    return False
end function

function approxElementOfPitchArray($k_{\text{test}}$, $K$)
    for all $k \in K$ do
        if approxEquatePitch($k_{\text{test}}$, $k$) then          ▷ equality ignoring octaves
            return True
        end if
    end for
    return False
end function
```

time from the onset and the duration and vice versa so we define the addition and subtraction.

For the sake of clarity, we reiterate that t represents a time tuple consisting of two values: n which is the numerator of the fraction representing divisions of a crotchet and d which is the denominator of the fraction.

4.3.1 Ordinal Operators

Since the time tuple represents a vulgar fraction, implementing the ordinal operators is simple (provided the language is capable of manipulating fractions). The algorithm for the less than ($<$) operator is given in Algorithm 8.

Algorithm 8 Algorithm for the less than operation for the time tuple

```
function lessThanTime($t_1$, $t_2$)
    if $\frac{t_1(n)}{t_1(d)} < \frac{t_2(n)}{t_2(d)}$ then
        return True
    else
        return False
    end if
end function
```

4.3.2 Addition and Subtraction

Similarly, implementing addition and subtraction with the time tuple is trivial with a programming language which is capable of mathematics with vulgar fractions. Algorithm 9 shows how addition is achieved with the time tuple.

Algorithm 9 Algorithm to add two time tuples

```
function addTime($t_1$, $t_2$)
    return $\frac{t_1(n)}{t_1(d)} + \frac{t_2(n)}{t_2(d)}$
end function
```

4.4 Interval Functions

Having defined the functions for manipulating pitch and time, we must finally define the functions for manipulating intervals.

For the sake of clarity, we reiterate that v represents an interval tuple consisting of three values: i which is the interval value in the spiral of fifths representation; z which is the number of divisions per semitone; and o which is the number of octaves in this interval.

4.4.1 Equality

As with pitch, there is more than one kind of equality between intervals. There is complete equality where the two intervals are exactly the same; there is type equality where the intervals are the same but with different octaves for example, a major 3rd has the same type as a major 10th; and there is class equality where two intervals are in the same class, for example a major 2nd and a minor 2nd are both of the interval class 2nd. We therefore define three functions in Algorithm 10 to calculate interval equality.

Algorithm 10 Algorithms to calculate equality between intervals

```
function equateInterval($v_1$, $v_2$)
    if $v_1(o) = v_1(o)$ and $\frac{v_1(i)}{v_1(z)} = \frac{v_2(i)}{v_2(z)}$ then
        return True
    else
        return False
    end if
end function

function approxEquateInterval($v_1$, $v_2$)
    if $\frac{v_1(i)}{v_1(z)} = \frac{v_2(i)}{v_2(z)}$ then
        return True
    else
        return False
    end if
end function

function equateIntervalClass($v_1$, $v_2$)
    if $\mathrm{Rem}(\frac{v_1(i)}{v_1(z)}, 7) = \mathrm{Rem}(\frac{v_2(i)}{v_2(z)}, 7)$ then
        return True
    else
        return False
    end if
end function
```

4.5 Other Functions

4.5.1 Text Representation and Conversion

To support the conversion from an external format to the format used in the database a number of conversion routines are necessary. It is also convenient to have a number of functions to convert between the representations used in the database and more familiar textual representations. This eases the job of formulating queries to the database and makes the resulting queries easier to read.

We therefore define a text format to represent musical pitches which is easy to read and relatively easy to convert to the computer representation. The format consists of a pitch class letter (upper or lower case), followed by optional accidental characters, followed by an optional octave number. The accidental characters are `#` for ♯, `b` for ♭. For example, the D♯ above middle C would be represented as `D#5`.

Algorithms 11 and 12 show the algorithms for converting between familiar text representations and the pitch representations used in the database for scales without semisharps.

4.6 Summary

In this chapter we have introduced a music representation which is based on Wiggins' abstract data type [123]. The current implementation differs in not representing timbre as part of a note and not allowing groups of groups (thereby eliminating one means of creating hierarchical structures). Timbre was excluded because it would be better represented at the note group level. Alterations to the music representation which will allow hierarchical structures are given in the Discussion 7.

Musical functions have been described based on those found in Harris *et al* [70]. Additional functions to deal with the many types of equality found in music and to convert between more familiar formats have also been described.

Algorithm 11 Algorithm to convert from the text representation of pitch and the representations used in the database.

```
naturals ← mapping(F ← 0, C ← 1, G ← 2, D ← 3, A ← 4, E ← 5, B ← 6)
function text2pitch(text)
    matches ← regex(([a-gA-G])([#b])*(-?[0-9]*), text)
    pitchClass ← naturals[matches[1]]
    if matches[2] then                                  ▷ If there is are # or b symbols...
        accidentals ← matches[2]                          ▷ ... assign them to accidentals
    else
        accidentals ← ‘’                                 ▷ ... else assign an empty string
    end if
    if matches[3] then                                    ▷ If there is an octave number...
        octave ← matches[3]                                       ▷ ... assign it to octave
    else
        octave ← 4   ▷ ... else assume octave is four (octave starting on middle C
    end if
    if stringContains(‘#’, accidentals) then
        alter ← 7 * stringLength(accidentals)
    else if stringContains(‘b’, accidentals) then
        alter ← -7 * stringLength(accidentals)
    else
        alter ← 0
    end if
    $k$ ← pitchClass + alter
    $z$ ← 1
    $o$ ← octave
    $p$ ← $\{k, z, o\}$
    return $p$
end function
```

Algorithm 12 Algorithm to convert from the representation of pitch used in the database and the text representation.

```
naturals ← mapping(F ← 0, C ← 1, G ← 2, D ← 3, A ← 4, E ← 5, B ← 6)
function pitch2text($p$)
    pitchClass ← $((p(k) + ((p(k) \bmod p(z)) * 7)) / p(z)) \bmod 7$
    accidental ← $(p(k) - (\text{pitchClass} * p(z))) / 7$
    pitchString ← keyOf(pitchClass, naturals)
    if $p(z) = 1$ then
        if accidental $> 0$ then
            accidentalSign ← ‘#’
        else if accidental $< 0$ then
            accidentalSign ← ‘b’
        else
            accidentalSign ← ‘’
        end if
        accidentalString ← stringRepeat($\frac{|\text{accidental}|}{p(z)}$, accidentalSign)
        text ← stringConcatenate(pitchString, accidentalString, octave
    end if
    return text
end function
```

These types and functions combined are intended to allow diverse queries to be asked of musical scores. This should fulfil the requirement for the current system that it should be capable of music analysis, in particular, that it should enable manipulation of musical data. Once implemented in a computer programming language, they should enable the accomplishment of another requirement: that the system should be queryable. The system which aims to achieve that goal is described in the next chapter and the use of the representation and functions is demonstrated in the Results (Chapter 6).

Chapter 5

System Workflow

5.1 Overview

The system described here is intended to enable the analysis of music and its performance. In the first chapter we formulated a number of requirements for such a system including a requirement to represent musical data so that it is available to music analysis. The following three chapters examined how to represent and manipulate musical score data to enable music analysis. This chapter will show how the types and functions are implemented in a working system.

Where the first three chapters were theoretical in their approach, this chapter and the one following it, will include more practical information. This chapter outlines the procedure for collecting data, processing that data outside of the system, loading the data into the system and getting results. It describes in detail how each step has been achieved and which tools have been used. By describing how music and performance data is collected, it is hoped that a better understanding of the capabilities and limitations of the current system can be attained.

In this chapter we will describe the method of collection of two types of performance data which can be used as examples for other types of data. Firstly, we will examine piano performance. There are several reasons for choosing piano performance. Firstly, piano performance data is most often represented in the form of MIDI data. Since many existing systems use MIDI data to represent musical performance, choosing piano performance here demonstrates the current system interfacing with the most popular format for storing music performance. Additionally, piano performance provides many interesting areas for investigation. In [105], Parncutt and Troup examined expressive gesture in piano performance. They found that a number of interactions between timbre, loudness and timing where a number of different performance gestures combine to influence the manipulation of these three musical expressions. Rather than finding a simple mapping between key velocity and loudness and key onset and timing, they found that combinations of different performance expressions produced the different expressive effects. This complexity validates the choice of piano performance for investigation.

Secondly, we will examine vocal performance. This provides a useful contrast to piano performance. When analysing piano performance we will be using the point primitives of MIDI data. Alternatively, when we analyse vocal performance, we will be using continuous data taken from an audio recording. In this way we can demonstrate the system analysing and displaying different types of data. Additionally, we can use vocal performance to demonstrate the system analysing music written using microtonal intervals.

In the follwing results chapter we will use this data to analyse the performances.

Figure 5.1 gives an overview of the system as a sequence of inter-connected processes. Each process is described in detail in a subsequent section of this chapter.

In a typical scenario of recording a piano performance, the performer will play from a printed score. A digital copy of that score in MusicXML format [68] will be combined with a recording of the MIDI data from the performance to create a Performance Markup Language (PML) file (Section 5.3.1) which contains both sets of data. The PML file is processed by the matching software (Section 5.3.2) which adds links between each performance event and the corresponding score note which the event realises. The matched PML file is uploaded to the database (Section 5.4) where after it is available for querying (Section 5.6) with the musical functions described in Chapter 4. A document can then be created to display the query results (Section 5.7).

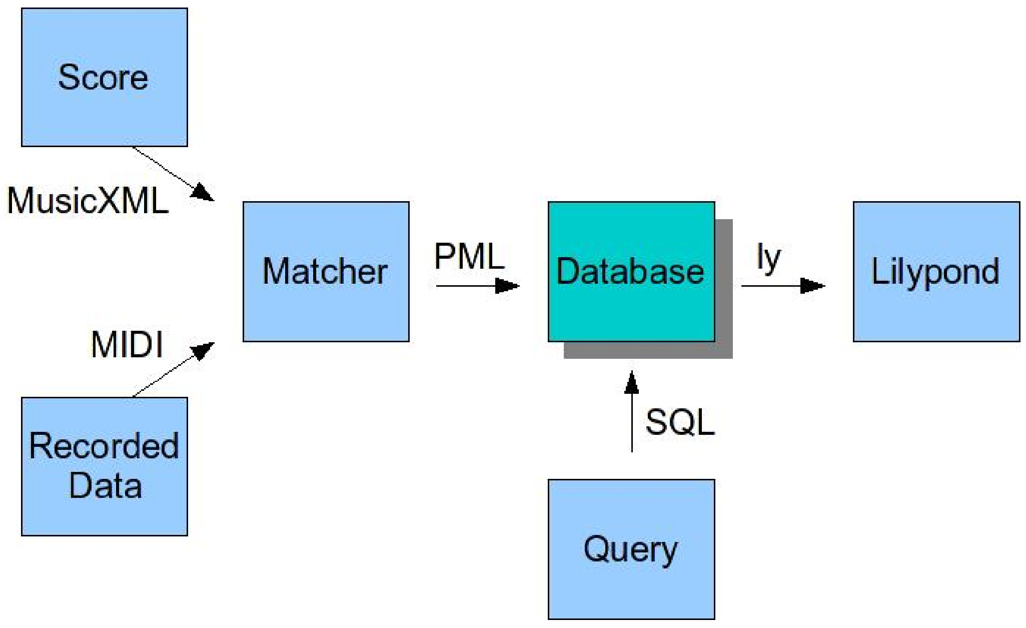


Figure 5.1: Diagram to show the flow of processes in the design of the system. Scores (in MusicXML format) and performance data (as MIDI data here) are processed by the matching programs to produce a matched performance in a PML file. The matched performance contains performance events which point to their corresponding notes in the score. The data from the PML file is loaded into the database. Queries are sent to the database in the SQL language. Results from queries are written to a Lilypond (.ly) file which is processed by the Lilypond program to produce a graphical score annotated with performance data

5.2 Data Gathering

The first stage, before the analysis of performance, is collecting together the various data sources for each performance. Figure 5.1 shows that this requires a score and some performance data. The current system accepts scores in MusicXML format and performance data in PML format. The PML file can include data from a MIDI performance or from a pitch-tracked vocal performance. This section will describe the 3 sources of data used in the next chapter.

5.2.1 MusicXML

Digital copies of the score are required in MusicXML format [68] since the subsequent processing stages need to use PML (a super set of MusicXML). Future versions of PML may support other XML score formats such as MEI [109].

MusicXML is an XML (eXtensible Markup Language) [7] format designed for the interchange of musical scores between music typesetting software. It is developed by Recordare, a company which sells plugins for commercial notation software such as Finale [8] and Sibelius [30] which use MusicXML to allow interchange between notation software. The specification for MusicXML is released under an open licence ("MusicXML™ Document Type Definition Public License Version 2.0" [22]).

There are 3 versions of the specification available: 1.0, 1.1 and 2.0. The differences between versions 1.0 and 1.1 are minor. Version 2.0 introduces a new compressed format. The specification is released as a commented Document Type Definition (DTD) and additionally in XML Schema language (XSL) for version 2.0. Where a DTD allows for the definition of tags and their location, XSL allows for stricter definition of their contents. XSL, however, is not as widely supported as DTD by XML validation software. The files used in the software described here aim to be compliant with version 1.1 of the standard.

The reasons for using an XML-based file format, and MusicXML in particular, for interchange are numerous and compelling. An XML file is both human-readable and machine-readable. This aids software development and debugging significantly as files can be read and understood by a programmer simply by opening them in a text editor. There are mature application programming interfaces (APIs) available for most languages making XML a language-agnostic choice. Furthermore, the APIs are standardised as the Document Object Model (DOM) and the Simple API for XML (SAX) which makes moving between systems easier for the developer.

There is much software available to produce MusicXML files: plugins are available for the 2 major commercial music notation programs: Sibelius and Finale as well as free software such as NoteEdit [23], MuseScore [10] and Rosegarden [29].

All these reasons make XML, and MusicXML in particular, excellent choices for file interchange. For an application which relies heavily on analysis and stored procedures, they are not suitable. In a MusicXML score, time is implied rather than explicit: a note's onset is the sum of the previous note's onset and duration which is the sum of the previous note's onset and duration and so on. Time can also be adjusted with the `<backup>` and `<forward>` tags which move the current time 'cursor' backwards or forwards. Furthermore, the presence of a `<chord>` tag means that the current note's onset is the same as the previous. All these reasons (and more) mean that MusicXML is not a good choice for analysis.

5.2.2 The Piano Bar

The Piano Bar is a device made by Moog Music Inc [96] to enable the recording of MIDI data from a standard 88-key acoustic piano. The device consists of an array of infra-red sensors on a toothed bar which rests on the cheeks of the piano. The 'teeth' extend between each black key. The infra-red beams shine down onto the white key and are reflected back up to the sensor. The beam is broken when the key is depressed. The beams shine across the black key to the sensors and are connected when the key is depressed. The key presses are converted to MIDI messages by the attached control box. The device is capable of sensing key velocity and has a separate sensor which can sense pedal movements which are converted to MIDI sustain and soft pedal messages.

The piano bar enables the recording of musicians using existing acoustic pianos thus enabling a more realistic setting for the recording of performance. Care must be taken when inferring from this data as a key press duration is not the same as the duration of the sound. The piano bar is not wholly unobtrusive. The bar reduces the length of the key available and flashes small lights above each key when a keypress has been detected. Several musicians have commented that this makes them uncomfortable.

The device is placed a small distance (< 5mm) above the keyboard and calibrated. The calibration involves adjusting the height till all the keys are correctly sensed (as indicated by the lights). Keyboards naturally flex and bend over time and individual keys can vary in their height as they 'bed in' therefore meaning that the distance from the sensor to the key is not always the same. It is not known whether this might affect the timing or velocity measurements. There is also no data available to describe how the velocity measurement relates to the actual speed of the keypress: how fast does the key have to move to generate a velocity value of 64? Additional measurements would have to be performed to provide reliable data. However, even if such data were available, it may not be entirely useful as differences between different manufacturers of pianos, between the pianos themselves, room acoustics and many other factors would produce subtle changes in performers playing which might confound any attempt to gain reliable objective measurements of velocity and timing. All that can be hoped for at present is a reliable relative difference between keypress timings and velocities in any given performance.

5.2.3 Audio, Video and MIDI Recording

In a typical recording session, audio and video will be recorded alongside the MIDI data. Audio is recorded by a microphone connected through a USB audio interface to a laptop computer. MIDI data is recorded from the Piano Bar through the interface to the same computer. Video is recorded from a high frame rate camera placed above the piano keyboard attached via firewire to the laptop. A full inventory of the equipment used is given in Appendix A.

Currently the database is not capable of making use of audio and video data directly though it is theoretically possible to store it. The functionality has not yet been added as there exists no software capable of combining the results from the database with audio and video. Steps towards achieving this end have been made and are described in Chapter 7 Section 7.8.4.

5.3 Data Processing

The previous section describes the collection of data. Scores are collected in MusicXML format and performance data is collected in MIDI files from the piano bar and audio files from a microphone. The next stage is the processing of those files to create the PML files ready to transfer into the database. This processing takes the form of establishing correspondances between the performance data and notes in the score.

This processing stage uses software developed by Dr Douglas McGilvray as part of his PhD [58]. In particular, this project uses the pitch tracker and segmenter (described in [98]); the matching software (Section 5.3.2) for score-performance matching and the Performance Markup Language (Section 5.3.1) (PML) as an intermediate format to store performance data before it is loaded to the database.

First the MusicXML file containing the score is processed to create a PML file containing the score and an empty performance section. Then the PML file's performance section is populated with data from a MIDI file or segmented audio file. Finally the PML file is processed by the matching software which matches the key-press events from the MIDI file to the score note from the MusicXML file. The matches are stored in the PML file for uploading to a database later.

5.3.1 Performance Markup Language

The process of matching score and performance data for musicological analysis presents the problem of where and how to store the results. Performance Markup Language (PML) aims to solve this problem by specifying a file format which extends MusicXML to include performance data. Thus the file stores the score, the performance data and the correspondances between the two, in one place.

A PML document has a `<pml>` tag as its root node. This is usually followed by a `<score-partwise>` tag which contains the partwise MusicXML score. The MusicXML section is pre-processed to add a unique identifier (`<id>`) to each note and a start time in score time. A `<performance>` tag follows which contains the performance data.

The performance section contains optional tuning information (`<tuning>`) and one or more performance parts (`<perfpart>`). Performance parts contain the data from a single performance of the piece in the score as a sequence of events (`<events>`). Events contain such data as the onset time and duration (in milliseconds) of the event, the MIDI note number and velocity, the estimated frequency and pitch (for pitch tracking), a unique identifier for the event and the note in the score with which it has been matched.

5.3.2 Score-Performance Matching

The procedure for matching a score and performance uses several different programs. It is relatively easy to follow, though it does contain several steps. The following sections describe each tool in the order in which they are used, with the output of each tool being passed as the input to the next.

mxml2pml This tool converts a MusicXML file to a PML file. It adds a `<pml>` root document node above the MusicXML score and an empty `<performance>` node after it. Each note in a performance is given an unique identifier and a start time.

mergeparts A MusicXML score consisting of several parts must first be processed to merge each of the musical parts into one to enable the subsequent matching of notes polyphonically. This is an unfortunate procedure as information is lost. It is hoped that a future version of the software will address this issue.

Followed by either:

midi2pml A MIDI file is combined with the PML file. No matching occurs here - the data from the MIDI file is translated into `<event>` nodes and inserted into the PML file.

Or:

audio pitch tracker and segmenter An audio file is processed by a pitch tracker and the resulting pitch contour is segmented into likely notes and inserted into the PML file.

winmatch The PML file, which now contains the musical score from the MusicXML file and the performance data is processed by the matcher. The program takes additional parameters to determine the window size (the number of notes it will scan for a match) and thresholds for determining the confidence of a match.

intermatch Finally the file is processed with an interpolation matching algorithm which unaligns and realigns potential matches found by the winmatch program.

5.4 Uploading Data

The previous two sections have described the collection of scores and performance data and the processing of the two to create a PML file. This PML file now contains the score, the performance data and the result of the matching process: correspondances between performance events and score notes. This data is now ready to be loaded into the database.

A script was written to upload data from a PML file to the database using the Python programming language, the built-in XML processing libraries and the PsycoPG2 PostgreSQL database access library. The script adds the performance data to the database and can optionally add the score. This allows the score to be added once along with multiple performances which may be located in different files.

The intermediate pitch contour data from the audio pitch tracker was loaded into the database from a Comma Separated Value (csv) file. The SQL `copy` command was used to load the data into a temporary table from where it was loaded into the `timed_data` table.

5.5 Database Design

In the first chapter of this thesis, several requirements were outlined for a system which aims to analyse music and its performance. Among them, requirement 4e stated that the system should be queryable — that is, it should be able to receive generalised queries not pre-programmed to do a specific task.

In order to address this requirement, the system has been created upon an object-relational database management system (ORDBMS) which provides a query language. Additionally, the chosen ORDBMS (PostgreSQL) addresses another requirement. Requirement 5 for accessibility states that the system should run on as many systems as possible since the MIR community use many different systems for research. The database chosen for the current application, PostgreSQL, runs on Windows, Macintosh and GNU/Linux operating systems.

The database has 3 tables which hold the majority of the data that is accessed in queries: `score_notes` which holds the score; the `timed_data` table which holds individual sample values; and `segments` which holds the PML events. Other tables exist to satisfy the requirements of many to many relations and to reduce the storage requirement and number of columns in the previously mentioned tables.

The `segments` table holds performance data which has a start and an end time (these are the performance data versions of Honing's interval primitives [73]). The `timed_data` table holds sampled data (Honing's point primitives).

Figure 5.2 shows the tables and their relations to one another. In addition to the three tables already mentioned, there is a `note_groups` table which holds the groups to which notes belong. These groups could represent key signature, time signature or measures. Since there is a many-to-many relationship between score notes and note groups (one note can be a member of many groups and one group can hold many notes), the relation is stored in a separate table: `note_groups__score_notes`.

The tables on the right of the Figure 5.2 are the metadata tables. These are described in Section 5.5.6 below.

5.5.1 Types

In accordance with the design outlined in Chapter 4, several new types were created in the database to represent pitch, time and musical intervals. All were created as composite types and given the prefix 'spoff' (an abbreviation of spiral of fifths).

The `spoff_pitch` type consists of three small integers[1]: the pitch (i.e., the position on the spiral of fifths), the divisions per semitone and the octave.

The `spoff_interval` type is essentially the same as the `spoff_pitch` type with the pitch replaced with the interval position on the spiral of intervals.

The `spoff_score_time` type was named to avoid confusion with real time. It is essentially a fraction, however, the PostgreSQL database does not have a fraction

[1]In the PostgreSQL database, small integers (smallints) are stored in 2 bytes.

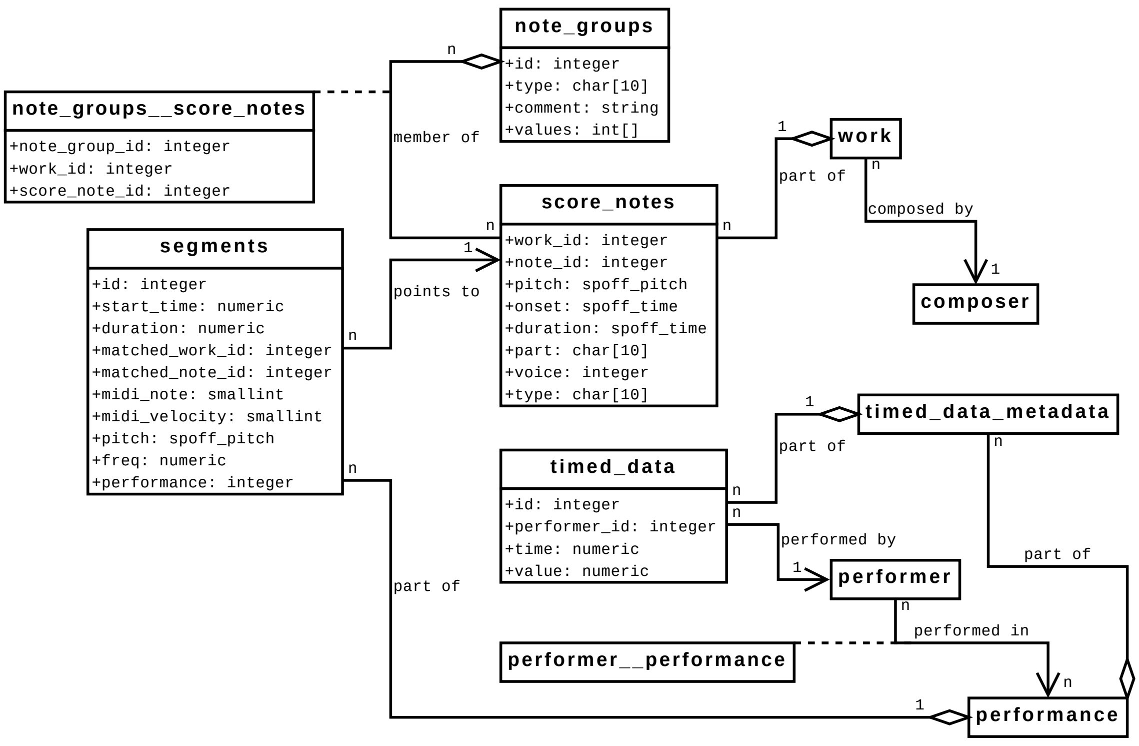


Figure 5.2: A UML diagram showing the main data holding tables of the database and their relations. The `score_notes` table holds the score data, the `segments` table holds the performance segmentation from the matcher and the `timed_data` table holds the sampled data. `score_notes` are members of `note_groups` in a many-to-many relationship. Tables holding metadata are shown on the right of the diagram and are explained in more detail in Figure 5.3

type so it is stored as 2 integers - a numerator and a denominator. The fractional representation matches that of MusicXML: it stores a fraction of a crotchet. Thus a quaver is stored as $\frac{1}{2}$ and a minim as $\frac{2}{1}$.

In addition to these three types, it is necessary to have a type in which to pass data between analyses and presentation functions. These types need to identify the note to which the datum belongs (using the work identifier and the note identifier) and carry a value. Two types have been created: `spoff_value_type` for carrying numeric values and `spoff_text_type` for carrying text values. In order to make it easier to implement sorting of the values for presentation in a score, both types include the part and voice of the note to which they belong.

5.5.2 Score Notes

The `score_notes` table holds the note data from the MusicXML score. The `score-_notes` table only holds those attributes which are common to all notes and is based on the MusicXML note type. A `score_notes` entry contains a pitch, duration, part, voice and type. The valid types are 'pitch', 'rest' and 'grace' though this could be extended to include some others. In addition to the MusicXML note, a `score_notes` entry contains an onset time, a note identifier and a work identifier.

The onset time is necessary to allow a note to stand independent of the other notes. MusicXML assumes that a note's onset is equal to the previous note's offset. By including the onset in each entry we lose the dependence of each note on those that went before it.

The work identifier and note identifier combine to make the composite key for the table. At first this might seem unnecessary: a properly normalised database should have a work stored in a separate table, a single identifier for each `score_notes` entry and another table to relate the work identifier to each note identifier. Difficulties arise because the PML file which is used to populate the database already contains note identifiers and relations from performance events pointing to those identifiers. When a MusicXML file is processed to convert it to a PML file, the note identifiers are added counting up from 0. Thus each PML file contains duplicates of the same note identifiers. We cannot therefore use the note identifiers directly as the database identifiers. We could create new identifiers for each `score_notes` entry, but we would have to store a map (in the database) between the PML identifiers and the database identifiers so that each subsequent performance of that score entered into the database could have the relational links between performance events and notes corrected. To eliminate this additional storage requirement and to simplify queries to the database (which would need an additional join if the work identifier to note identifier relation were stored in another table), we combine the work identifier and note identifier as the composite key.

5.5.3 Note Groups

Once the minimum amount of data is entered into the `score_notes` table, there remain further pieces of information which are not common to all notes but only a subset of the notes such as membership of a slur. This information is represented as note groups and is stored in two tables: `note_groups` and `note_groups__score-_notes` The former stores information about a group and the second holds the relation between the group and a note.

The `note_groups` table holds an integer identifier, a short string describing the type of group, a space for a comment and an array of integer values. For example, notes which are all in the same measure are placed in a measure note group. The value array in this case holds the measure number and the start time numerator and denominator. The start time is necessary to avoid complex calculations in pieces with changing time signatures or polychronous music (where there are two or more time signatures operating in parallel.

The `note_groups__score_notes` table simply holds the note group identifier and the corresponding work identifier and note identifier of the member note.

Three notations of Western music are also represented as note groups. The clef, key signature and time signature are all stored in this way. This allows their value to change over the course of a piece and the affected notes to be indicated.

5.5.4 Performance Segments

The performance events from the PML file are stored in a table named `segments` The performance events are created by pitch-tracking software and score-performance matching software. As such they can contain different data. The implementation described here is discussed further in Section 7.7.2 including possible improvements to the design.

The segments entries consist chiefly of an integer identifier, a start time and duration and the work and note identifiers of the matched note. The start time and duration are stored as numeric types since they indicate the locations in performance time. In addition, a MIDI performance includes entries for MIDI note number and velocity and a vocal performance includes an entry for the estimated frequency in Hertz. Both types of performance include entries for the estimated unambiguous pitch of the event. Finally, the identifier for the performance to which the segments belongs is included. The performance table is described in Section 5.5.6.

5.5.5 Timed Data

Sampled data can be stored in the `timed_data` table. This table is intended to store continuous data such as sampled data. The table holds an integer identifier, a performer identifier, a time as a numeric type and the value also as a numeric type. In addition, each row also stores a metadata identifier which points to an entry in the `timed_data_metadata` table. This table stores information about the timed data such as the performance identifier and a comment.

5.5.6 Metadata Tables

There are also items of data which must be stored in addition to the contents of the score and the contents of a performance. This section describes the tables which hold this metadata.

Figure 5.3 shows the metadata tables and their relationships. On the left of the figure are the main data holding tables described above (Figure 5.2). On the right are the tables which hold the metadata.

There are two tables that hold information about people related to the data stored in the database. The performer table holds the first name and last name of each performer along with a unique identifier. The composer table holds the first name, last name, initials, date of birth and death and an identifier for each composer. In addition, there are attributes for initials, second name and alternative names. This allows users of the database to search for different parts of names and dates and still arrive at the correct composer. For example, ‘J. S. Bach’, ‘Johann Christian Bach’ and the various composers called ‘Johann Christoph Bach’ who had different dates of birth and death.

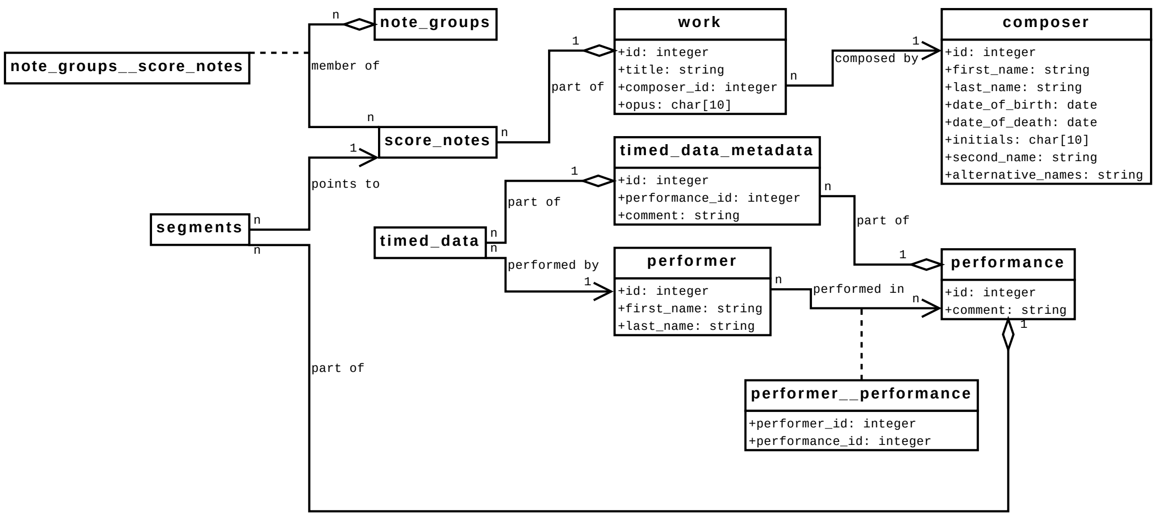


Figure 5.3: A UML diagram showing the metadata holding tables on the right and the main data holding tables on the left. The `work` table hold information about the musical piece. The `composer` and `performer` tables hold information about people. The `performance` table holds information about the performance and the `timed-_data_metadata` table holds information about timed data. The `performer` and `performance` tables are related by a many-to-many relationship.

The work table holds the relation between a composer and score data. Each work has an identifier, a text field to hold the title, a field to hold a reference to a composer id and a text field to hold the catalogue or opus number.

Each performance is stored in a table with simply an identifier and a text field for a comment. The relation between a performance and a performer is stored in a separate table since there could be many performers in a single performance or many performances by a single performer.

5.5.7 Indexing

The PostgreSQL database automatically creates an index for each table using the primary key. This allows the database to go directly to a table entry, of which the identifier is known, without searching for it from the beginning of the database. In practise, queries will need to access the database using attributes other than the identifier. To increase the performance of these queries, several indices were created for `score_notes` and `segments` tables.

The `score_notes` table has an index on the primary key — a combination of the work identifier and the note identifier. In addition, indices were created for the work identifier alone, the note identifier alone and the part identifier. These attributes all use PostgreSQL's built in types so their creation was relatively simple. Indices were also sought for the note onset time and note duration. These attributes are stored in custom types so the creation of the indices is a little more involved.

The entire set of comparison operators must be created: $<$, $>$, $<=$, $>=$, $=$. This involves writing a function for each operator and an operator entry so that the database knows which function to call for each operator. Additionally, a comparison function is required. This is a function which returns -1 if a $<$ b, 1 if a $>$ b, 0 if a $=$ b and false in all other cases. Finally the operators and the comparison function are grouped together by making an operator class entry which associates them with the type that they operate on.

The `segments` table has an index on its primary key - the segment identifier. Additionally, indices were created on the matched note identifier, the matched work identifier and the performance identifier.

5.6 Creating Documents

In the previous sections we have seen how to collect data, process it and load it into the database. The preceding section has shown how the design outlined in Chapter 4 has been implemented in the database. This section will outline the procedure for querying the database to create data structures (documents) which can be processed later to produce graphical scores annotated with performance data.

There are three stages to creating documents that contain the results of queries: firstly, the document is created and populated with notes from a score; secondly, the data to be presented is added to the document; finally, the document is converted to Lilypond markup for later processing with the Lilypond typesetting software.

5.6.1 Populating the Document

A document to contain results is created at the same time that we populate it with notes from a score. In this context, the term document refers to a data structure stored in the database where we hold note data and other data in a form that can be later processed to produce Lilypond markup. The PostgreSQL Python implementation provides a global dictionary object for each session which persists between function calls and queries. We use this to store our document data structure so that we can use several queries to add data to it.

The SQL code for populating the document is shown in Figure 5.1. The `populateDocument` function takes two arguments: a string containing the name of the document to be created and the input rows to be added to the document.

Program 5.1: A sample SQL query showing how to use the populateDocument function to populate a document called 'mydoc' with notes from a work with the identifier 4.

```
select populateDocument('mydoc', score_notes) from score_notes
where work_id = 4;
```

The `populateDocument` function is an aggregate function. In PostgreSQL, normal functions are those which take 0 or more arguments (such as a row from a table) and return a result. Aggregate functions take several rows as arguments and return a result. Examples of built in aggregate functions are `SUM` and `COUNT`. Aggregate functions are created by specifying a function that is called for each input row and a variable and its initial condition. The function takes the variable, 1 input row and any other variables as its arguments. The variable is updated with the value returned by the function after each call.

The `populateDocument` function calls another function called `addScoreNoteToDocument` which actually adds the note to the document. The `populateDocument` function sets the value of the initial condition variable to false. Each successful call to `addScoreNoteToDocument` returns true, setting the variable to true. The `addScoreNoteToDocument` uses the value of this variable to determine if it is the first run of the function. If it is (the variable is false), it creates an entry in the Global Dictionary with the document's name and adds a sub entry named `noteData` to store the notes. It then creates sub-entries for the work identifier and the note identifier and adds the remaining score note attributes under this entry. Thus each note is locatable by its work identifier and note identifier. An outline of the data structure is given in Program 5.2.

Program 5.2: Pseudo-Python code showing the document structure that is used to hold data for a presentation of results alongside a score. The **textUnderList** holds pieces of text which are displayed under notes such as melodic intervals. The **barGraphList** hold data which will be displayed in bar graphs under notes. Finally, the **noteData** list holds the score.

```
GD = {
 doc {
  "textUnderList" {
   part_id {
    voice [
     "melodic_intervals",
     "chord_names", ...]
   ...}
  ...}
  "barGraphList" {
   part_id {
    voice [
     "MIDI_velocity",
     "ioi", ... ]
   ...}
  ...}
  ... #other display methods
  "noteData" {
   work_id {
    note_id {
     "pitch": (0,1,0), "onset": (0,1,2), ... #rest of score_note type
     "melodic_intervals": 5, "MIDI_velocity": 99, "ioi": 32 ... }
   ...}
  ...}
 ...}
}
```

The data structure for a document is stored in a Python dictionary object (known as an associative array, map or mapping in other languages). The name of the document (used as the argument to the `populateDocument` function) is used as the key for the structure. The key `noteData` is used for the score note data. The work identifier and subsequently the note identifier are used to indicate the location of a note. The remaining attributes of the note are stored under this identifier.

Additionally the data structure is used to store data for display alongside the score. This will be described in Section 5.6.2. The pure Python data structure is used over any other structure (such as creating a custom object) in the hope that it will result in faster executing code following the advice of [27]. Since this structure is accessed and added to many times in a query it is important that it operates quickly.

5.6.2 Adding Data to the Document

Now that the appropriate data structure has been created and it has been populated with a score, we can add performance data to it. This will enable us to ultimately create a presentation of performance data alongside the score.

Three methods of adding data to a score have been created which allow for different types of data to be displayed: `addTextUnderNotes` allows text to be displayed under notes; `addBarGraphUnderNotes` allows a box containing a bar and a value to be drawn under notes; and `addLineGraphUnderNotes` draws a box containing a line graph plot from a sequence of values. The functions accept data in one of two types: `spoff_value_type` and `spoff_text_type`

The `addTextUnderNotes` function is an aggregate function which calls the `add-TextToNote` function for each `spoff_text_type` The function accepts a name for the data, the name of the document to which it is being attached and the value to be attached as its arguments. On its first run it creates an entry `textUnderList` under the document name. For each part and for each voice under each part, an entry is made under this entry with the name for this data. This is to allow easy iteration over all the `addTextUnderNotes` entries for each voice when we come to output the score later. Finally, an entry is made in the corresponding note entry with the data name as the key. The `addBarGraphUnderNotes` operates in a similar fashion.

The `addLineGraphUnderNotes` function operates a little differently as it requires a sequence of values to be attached to each note rather than a single value. The function is an aggregate function which calls `addLineGraphToNote` for each `spoff-_value_type` On the first run, it creates the entries under the document name. For each value, it checks whether there is a already an entry for the corresponding note with the current data name. If there is none, it creates a list with the current value as its only item. If it finds a list, it appends the current value.

5.6.3 Extracting the Data as Lilypond Markup

The previous two sections have shown how to create a data structure in the database and populate it with a score and various types of performance data. This section will show how to extract that data structure in Lilypond format ready to be turned into a graphical presentation of performance data alongside the score.

The `getLilypond` function is used to extract the named document in Lilypond markup format. An example query is shown in Program 5.3.

Program 5.3: An example SQL query showing how to extract a named document in Lilypond format.

```
select getLilypond('mydoc');
```

The simplicity of the query hides the complexity of the Python function that it calls. The `doc2lilypond` Python function takes the data structure outlined in Program 5.2 as its first argument and the `plpy` object as its second. The `plpy` object is provided to all Python functions called in the database. It provides access to the database from the Python environment allowing queries to be made in SQL and results to be returned.

The aim of the `getLilypond` function is to return a document in Lilypond markup including the score and any data plots that have been attached to it so that it can be later processed by the Lilypond command-line program to create a pdf file. It achieves this by creating Lilypond markup which contains the score and lyric lines.

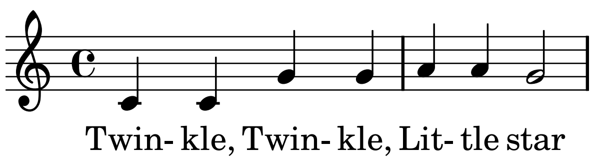


Figure 5.4: The result of processing Program 5.4 with Lilypond.

Each lyric line holds a line of data values rather than words. Custom functions for the creation of bar graphs and line graphs are included in a header to each file and are called from the lyric line to draw the appropriate graph under the note. The structure of the resulting file is outlined in Program 5.4 with the output from Lilypond in Figure 5.4.

Program 5.4: An example Lilypond file showing the basic file structure.

```
\book {                                   %one per document
  \score {                                %one per work
    \new Staff = "part_id" <<             %one per part
      \new Voice {                        %one per voice
        \clef treble \time 4/4 \key c \major
        c'4 c' g' g' a' a' g'2
      }
      \addlyrics {                        %one per data list
        Twin-- kle, Twin-- kle, Lit-- tle star
      }
    >>
  }
}
```

The `doc2lilypond` function first creates a string to hold the Lilypond markup and inserts the scheme code for the different data display functions. It then iterates over the data structure creating a score section for each work identifier it comes across. For each work, it creates a Python set[2] containing the identifier of each musical part. For each part, it creates a new staff in the Lilypond string and a set of each musical voice. It then creates a list of each note contained in each voice. For each voice it creates a voice section in the Lilypond string and a list of the data names of any data lines for the current voice.

[2]A Python set is like a list but each value can only occur once

For each note, it obtains a list of the note groups to which the note belongs by sending a query via the `plpy` object. By comparing to previous notes' values, it establishes if there have been any changes in key signature, time signature or clef and sets the value of the corresponding strings. It also establishes if the current note is the beginning of a tie, a slur or a chord and sets the values of the corresponding strings. It appends the strings to the Lilypond string in the order: `chordStartString`, `noteString`, `durationString`, `slurString`, `tieString` and `chordEndString`

Each data entry is appended to a lyric line string which is itself appended to the Lilypond string at the end of the voice.

5.7 Displaying Results using Lilypond

The preceding section has shown how to extract score and performance data from the database into a file in Lilypond markup. This file is now ready to be processed by the Lilypond typesetter to create the graphical presentation of a score annotated with performance data.

The text returned by the `getLilypond` function can be saved to a file from the psql PostgreSQL shell. This file can then be processed on the Linux command line using the Lilypond program to produce a PDF score annotated with graphs of performance data.

The functions used to annotate the score with graphs are shown in Program 5.5.

Program 5.5: The Lilypond scheme functions used to annotate scores with performance data graphs. The `valuebox` function is used to draw a bar graph with a text box containing the value. The `linegraphbox` function is used to draw a line graph.

```
\version "2.12.1"

valueboxheight = 10
valueboxwidth = 1
#(define-markup-command (valuebox layout props val) (number?)
 "Draws 2 boxes - one containing val as text,
  one containing val as a line graph - in a column"
```

```
   (interpret−markup layout props
    (markup
     #:center−column
      (#:override '(box−padding . 0.1)
       #:rounded−box
        (markup #:override '(font−size . −5)
         (format #f "~$" val))
          #:override '(box−padding . 0.1)
           #:rounded−box
            (#:combine
             (#:combine
              #:with−color (x11−color 'white)
               #:filled−box `(0 . ,valueboxwidth) `(0 . ,valueboxheight) 0
                #:with−color (x11−color 'blue)
                 #:filled−box `(0 . ,valueboxwidth) `(0 . ,(* 5.29 val)) 0 )
                  #:translate `(−0.5 . 0) #:draw−line `(,(+ 1 valueboxwidth) . 0))
))))

linegraphboxheight = 10
linegraphboxwidth = 5
linegraphscalefactor = 1
linegraphbias = 0

#(define (pslines prev_x x_inc y_list)
  (if (> (length y_list) 0)
   (format #f "~$ ~$ ~a~a" (exact−>inexact (+ prev_x x_inc)) (car y_list) "lineto
 " (pslines (+ prev_x x_inc) x_inc (cdr y_list)))
   ""
))

#(define (generate−ps y_list)
  (format #f "~$ ~$ ~a~a~a" '0 (* linegraphscalefactor (+ (car y_list) linegraphbias)) "moveto
 " (pslines '0 (/ linegraphboxwidth (− (length y_list) 1)) (cdr y_list)) "stroke"))

#(define−markup−command (linegraphbox layout props y_list) (list?)
 "draws a line graph from a list of values in fixed size box"
  (interpret−markup layout props
   (markup
    (#:rounded−box
     (#:combine
      #:with−color (x11−color 'white)
       #:filled−box `(0 . ,linegraphboxwidth) `(0 . ,linegraphboxheight) 0
      #:with−color (x11−color 'red)
       #:postscript (generate−ps y_list )
)))))
```

The first function, `valuebox` draws a bar graph with a text box containing the value. The function creates a column containing two rounded boxes. The top rounded box contains the value, formatted to 2 decimal places and printed in a small font. The bottom rounded box contains a white box. This a cheat which is not visible. It forces the rounded box to be a certain fixed size — it defaults to shrinking or expanding to fit its contents. On top of the white box, a blue box is drawn to the size of the value to be displayed and a line is drawn at the origin.

The next three functions draw the line graphs. Starting with the third function, `linegraphbox` it draws a rounded box filled with a white box as above. Inside this it calls the `generate-ps` function to generate the Postscript commands to draw the line graph. The `generate-ps` function outputs Postscript commands to move to the first value and to draw the line (`stroke` once the line has been defined. In between these commands, it calls the `ps-lines` function which outputs the `lineto` commands for each value in the list `y_list`

By repurposing the scheme interpreter and drawing commands of Lilypond to draw graphs rather than musical notation, it is possible to annotate scores with performance data.

5.8 Summary

In the introduction to this thesis we looked at systems designed for the analysis of music and systems designed for the analysis of performance. We proposed that, in order to better understand the *process* of performance, we require a system which can analyse both music and performance together. This system will allow us to understand how the structure of music is reflected in its performance and how the performance indicates the structure. Since there exists no such system in the literature, a set of requirements for such a system were produced.

The following three chapters explained the theoretical approach taken here to solving some of the problems faced in creating such a system: the representation of pitch; the representation of time; and the creation of functionality to manipulate these representations for music analysis. In this chapter we have seen how the theoretical information of the previous three chapters has been implemented in a practical system. This represents the culmination of the work of this thesis: the creation of a system capable of analysing musical scores and performance in one place and presenting the results in a musically appropriate way — a score annotated with performance data.

By creating scores annotated with performance data from a system which is capable of receiving diverse queries, the current system aims to fulfill the requirements in Chapter 1 for a system which is queryable, able to manipulate score and performance data for music analysis and able to present performance data in the context of the score.

The next chapter (Chapter 6) will demonstrate the use of this system for the analysis of several different performances. The performances and analyses have been chosen to demonstrate the capabilities of the system in analysing music (including microtonal music), analysing performance data and displaying different types of data. The entire system will be discussed in the chapter following the next chapter (Chapter 7.

Chapter 6

Results

In this chapter we use the database (Chapter 5) and the functions (Chapter 4) to create queries of several performances of different pieces of music. The queries are used to populate documents which in turn generate Lilypond code which can be processed to produce musical scores annotated with performance data.

Section 6.1 shows two types of analysis of a piano performance producing a score annotated with bar graphs. Section 6.2 shows an analysis of the intonation of a vocal performance producing a score annotated with line graphs showing the tuning and pitch contour over the course of each sung note. The analysis also shows the system is capable of representing and processing music in non-standard tuning systems. Finally Section 6.3 shows a combination of musical and performance queries producing a score annotated with normalised inter-onset intervals and musical intervals with dissonant intervals highlighted.

6.1 Displaying Performance Data - single values

In this section we will follow the process of creating a presentation of performance data alongside a score. The data used was collected from a performance of Chopin's Prelude No. 7 by the pianist Martin Jones — one of Britain's most highly regarded solo pianists. The recording setup used was that described in Section 5.2.3, namely:

a grand piano with the piano bar attached, a high frame-rate camera and a computer to record the MIDI, audio and video.

The MIDI data was processed as described in Section 5.3.2 to produce a PML file which was uploaded to the database.

6.1.1 The Query

The aim of the query described here is to annotate the performance with bar graphs showing the keypress durations (the length of time the performer holds a piano key) and the inter-onset interval (IOI) (the length of time between down keypresses). Both of these measures give an indication of the tempo and playing style of the performance. For example, a sequence of short-duration keypresses might indicate an accelerated tempo or it might indicate a staccato playing style. Only when this is combined with the IOI will it become clear which is in operation. Alternatively, a sequence of long duration keypresses might indicate a decelerated tempo or a legato playing style. Combining this information with the IOI allows the two to be distinguished. Combining this data with a score allows the musicologist to see how the structure of the piece is reflected in the performance.

The full query is given in Program 6.1.

Program 6.1: A query in SQL to create a document with two bar graphs under each note, one showing the keypress duration and one showing the inter-onset interval.

```
--set the performance id (used later) to be Martin Jones.
--This is a variable for the psql program and is not proper SQL.
\set performance_id 27

begin;

--add score notes to the Python data structure which holds the document.
--The finale work has id=5, the document is called 'mjones'.
select populatedocument('mjones', sn) from score_notes as sn where work_id=5;

--add ioi bargraphs under the notes. The name of this lyric line is 'ioi'.
select addBarGraphUnderNotes('mjones', 'ioi', data)
```

```
        from (
        select  first.work_id,
                first.note_id,
                first.voice,
                first.part_id,
                (seg_fol.start_time - seg_first.start_time) as value

                from score_notes first
                inner join score_notes following
                        on (first.onset + first.duration = following.onset
                        and first.work_id = following.work_id
                        and following.type = 'pitch'
                        and first.voice = following.voice
                        and first.part_id = following.part_id
                        and first.note_id <= 553)
                left join segments seg_first
                        on (seg_first.matched_work_id=first.work_id
                        and seg_first.perf_id = :performance_id
                        and seg_first.matched_note_id = first.note_id
                        and seg_first.align = 'correct')
                left join segments seg_fol
                        on (seg_fol.matched_work_id=first.work_id
                        and seg_fol.perf_id = :performance_id
                        and seg_fol.matched_note_id = following.note_id
                        and seg_first.align = 'correct')

                where first.work_id = 5
                and first.type = 'pitch')
        as data;

--add keypress duration bargraphs under the notes,
--the name of this lyric line is 'perf_dur'.
select addBarGraphUnderNotes('mjones', 'perf_dur',query_vals_st)

        from (
        select  segments.matched_work_id as work_id,
                segments.matched_note_id as note_id,
                sn.voice as voice,
                sn.part_id as part_id,
                segments.duration as value

                from segments,
                (select score_notes.note_id,
                        score_notes.work_id,
                        score_notes.voice,
                        score_notes.part_id,
                        score_notes.type,
                        score_notes.onset,
                        score_notes.duration,
                        score_notes.pitch

                        from score_notes
                        inner join
                        (select ng.score_note_note_id as note_id
                            from
                            (select *
                                from note_groups__score_notes
                                inner join
                                (select id
                                    from note_groups
                                    where type='measure'
                                    and value[1] <= 22)
                                as t2
                                on note_group_id = t2.id)
                            as ng
                            where ng.score_note_work_id=5)
                        as t3
                        on score_notes.note_id = t3.note_id)
                    as sn

                    where segments.matched_work_id = sn.work_id
                    and segments.matched_note_id = sn.note_id
                    and segments.perf_id = :performance_id)
                as query_vals_st;

--process the Python data structure and create the lilypond file.
select getlilypond('mjones');

commit;

--this is a psql command to save the result to a file
\g 'prelude_7_ioi+dur_mjones.ly'
```

The first line sets the variable `performance_id` to the value 27. We use this variable later on to find the correct performance. The value is the id of the performance that we want to display. The `BEGIN` directive begins a transaction for the current block of SQL. If any of the queries fail, then any changes made to the database are reverted to the state before the transaction began (called "rolling back" in database terminology).

The first sub query occurs at line 12 where the `populateDocument` function is called. A document called `mjones` is created and populated with the notes from the score with a work identifier equal to 5.

The next sub query calculates the inter-onset interval and attaches it to each note by calling the `addBarGraphUnderNotes` function. It achieves this by calculating several joins before picking out the columns to use as the `spoff_value_type`

In order to calculate the IOI for each note: we find the note that follows it; find the start time in the performance of the first note; find the start time in the performance of the following note and calculate the difference.

The first join is an inner join and included to find the following note for each note in the score. For each row of the table on the left and the table on the right which match the join criteria, a new row is created which has all the columns of the left table and all the columns of the right table. In this example, the table on the left is the `score_notes` table called `first` and the table on the right is the same table, this time called `following` The join criteria can be seen in lines 23-28: the following note's onset must equal the first note's duration added to its onset; the part identifier, voice and work identifier must be the same; the following note must be a 'pitch' type (i.e., not a rest or grace note) and the note identifier must be less than or equal to 553. This last criterion is a crude method of restricting the result to the the first portion of the piece. An improved, though more complex, method using bar numbers is given later. The criteria for the first note is given in the `where` clause at the end of the sub query (lines 40 and 41) where it is restricted to a given work identifier and only notes of type 'pitch.'

We use an inner join here so that we are left with a table which includes only those notes which match the join criteria. For the calculation of an IOI, we only want those notes which are followed by another note. We want to disregard those notes which are followed by a rest (because we cannot use them to calculate an IOI). When we calculate keypress durations, we want to include those notes which are followed by a rest, so we use a left join.

Of particular importance in these join criteria is the one which specifies that the following note's onset must equal the first note's onset plus duration. There is no implicit or explicit ordering of notes in the database: rows of a table are ordered according to the order in which they were created, although further processing will move them to different places. It is therefore not possible to rely on the database returning rows in a particular order unless that order is specified in the query. The simplicity of the join criteria also hides some complexity. The onset and duration of notes is stored in a custom type. The addition and equality operators call custom functions to operate on these types.

At this point we have a table with rows for each note in the score with additional columns for the note in the score which follows it.

The second join is a left join. For each row of the table on the left, a new row is created with all the columns of the left table and all the columns of the right table if the right row satisfies the join criteria. In this example, our result from the previous join is the table on the left and the `segments` table is the table on the right called `seg_first` The join criteria (in lines 30-33) state that the row from the `segments` table must be correctly matched with the work identifier and note identifier of the first note and the performance identifier matches the values assigned to the variable at the top of the query. We have now attached columns to our row which describe the first note's performance.

The third join is almost exactly the same as the second join except that this time we are matching against the following note. This subquery has used three joins to create a table with each note of the score followed by the following note in the score, the data for the matched first note from the performance and the data for the matched following note from the performance. Finally at lines 15-19, the identifiers for the work, part, voice and note for the first note are selected and the IOI is calculated as the following note's start time in the performance less the first note's start time. These are given as arguments to the addBarGraphUnderNotes function so that it attaches the IOI value to the first note.

The second subquery aims to add bar graphs under notes representing the duration of keypresses in the piano performance. The subquery takes a different approach than the subquery for the IOIs and is best understood by starting in the middle and working out.

The purpose of this first section is to find the group identifiers for the bars we are interested in and use these to select only those notes which are members of those groups. Starting at lines 72-766, we see a subquery to find the identifiers for all note groups which have the type 'measure' and the measure number less than 22. The next outer query is a join between the `note_groups__score_notes` table and the subquery containing the identifiers of the measure groups. This is an inner join so the result is rows consisting of the note group identifier and the score note identifiers of only those rows which are in the measures less than or equal to 22.

The next outer query (lines 67-79) picks out the note identifiers for all the rows whose work identifiers match the current work i.e., 5. We now have a table consisting of a single column of note identifiers which are in the current work from measures 1 to 22. This is an improved method to finding the notes in the first portion of the piece to the one described above since it does not rely on the note identifiers to imply the order of the notes.

This table is used to pick out the notes we need from the `score_notes` table by performing an inner join between the two tables (lines 56-86).

The penultimate layer of the subquery selects out those columns needed to satisfy the `spoff_value_type`: the identifiers for the work, part, voice and note and the value – calculated from the segments onset and offset of the segment which matches the note identifier, work identifier and the performance identifier. Finally the value is attached to the note by calling the addBarGraphUnderNotes function.

Now that the data structure has had all the data attached to it, we can generate the Lilypond file with the performance data. This is achieved in line 90 by calling the `getLilypond` function.

6.1.2 The Result

The Lilypond code resulting from the query is in Appendix B in full and is summarised here in Program 6.2 in its edited form.

Program 6.2: A summary of the Lilypond markup code generated by the query in Program 6.1.

```
\version "2.12.1"

         valueboxheight = 10
         valueboxwidth = 1
         valueboxscalefactor = #5.29
         #(define-markup-command (valuebox layout props val) (number?)
               "Draws 2 boxes - one containing val as text,
                one containing val as a line graph - in a column"
                  (interpret-markup layout props
                   (markup
                    #:center-column
                     (#:override '(box-padding . 0.1)
                     #:rounded-box
                      (markup #:override '(font-size . -5)
                       (format #f "~$" val))
                        #:override '(box-padding . 0.1)
                         #:rounded-box
                          (#:combine
                           (#:combine
                            #:with-color (x11-color 'white)
                             #:filled-box `(0 . ,valueboxwidth) `(0 . ,valueboxheight) 0
                            #:with-color (x11-color 'blue)
                             #:filled-box `(0 . ,valueboxwidth) `(0 . ,(* valueboxscalefactor val)) 0 )
                              #:translate `(-0.5 . 0) #:draw-line `(,(+ 1 valueboxwidth) . 0))
                     ))))


%\book {
 %\score { <<
  \new Staff = "Staff 1 "
   <<
    {
     \key a \major
     \clef treble
```

```
      \time 3/4
      r4 r4 e'4 cis''8. d''16
      <d' gis' b'>4 <d' gis' b'>4 <d' gis' b'>2 <d'' fis''>4
     }
     \addlyrics
     {
      \markup \valuebox #0.7406 \markup \valuebox #0.5876 \markup \valuebox #0.1666
      \markup \valuebox #0.6479 \markup \valuebox #0.6531 \markup \valuebox #1.4011
      \markup \valuebox #0.625
}
     \addlyrics
     {
      \markup \valuebox #1.0365 \markup \valuebox #0.6271 \markup \valuebox #0.2343
      \markup \valuebox #0.3062 \markup \valuebox #0.2292 \markup \valuebox #0.7115
      \markup \valuebox #0.7458
}
    >>
   \new Staff = "Staff 2 "
    <<
     {
      \key a \major
      \clef bass
      \time 3/4
      r2. e,4
      <e e'>4 <e e'>4 <e e'>2 r4
     }
     \addlyrics
     {
      \markup \valuebox #0.7094
      \markup \valuebox #0.6719 \markup \valuebox #0.6343 \markup {}
     }
     \addlyrics
     {
      \markup \valuebox #0.25
      \markup \valuebox #0.2813 \markup \valuebox #0.177 \markup \valuebox #0.8416
     }
    >>
 %>> }
%}
```

The Lilypond code in Program 6.2 has 2 sections: the first is the scheme function which draws the graphs; and the second is the Lilypond markup which holds the score and calls the scheme functions. The scheme function, called `valuebox` draws a bar graph to the height of its argument (scaled by `valueboxscalefactor` inside a box of height `valueboxheight` and width `valueboxwidth` The Lilypond section consists of 2 staves of Lilypond score notation, each followed by 2 lyric lines. In stead of containing lyrics, the lyric lines consist of a call to the markup command in place of each word. The argument to the markup command is the call to `valuebox` to draw the bar graph. In this way, a bar graph is drawn under each note. The output routines which generate the Lilypond code have to take account of the fact that Lilypond, understandably, does not allow lyrics under rests and the second note of a tie.

The score resulting from the Lilypond code in Program 6.2 is shown in Figure 6.1.

The top line of bar graphs in Figure 6.1 shows the inter-onset interval. The final note in the left hand has no IOI bar as it is followed by a rest. The bottom line shows the keypress duration. One can see a staccato playing style in the left hand where keypresses are short relative to the inter-onset interval. In the right hand, one can see an extra long pause on the first note of the third bar. This is indicative of a phrase ending — the performer indicates the structure through a pause here. The presentation of performance information alongside the score shows how the musical structure is reflected in the performance.

In this section we have seen how a presentation of data alongside a score is created. We have retrieved a score from the database and annotated it with two different types of performance data calculated from the data held in the database. We have shown that the system is able to analyse performance data and display single values as bar graphs aligned with the corresponding note in the score. This section shows the system fulfilling several of the requirements in Chapter 1: to enable the analysis of performance data in the computer (Requirement 4); and to present the results aligned with the score (Requirement 4f).

The next section will address presenting continuous data alongside the score and will address Requirement 2: to manipulate music in different tuning systems.

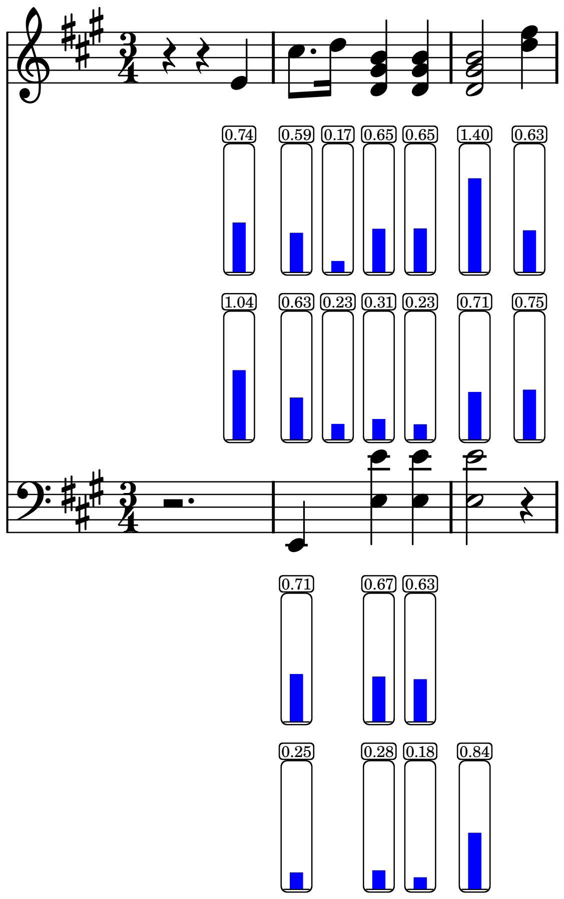


Figure 6.1: The resulting annotated score from the edited lilypond code in Program 6.2. The first 3 bars of Chopin's Prelude No. 7 is shown annotated with performance data from a performance by Martin Jones. The top line of bar graphs under each note shows the inter-onset-interval and the bottom line shows the keypress duration. One can see a staccato playing style in the left hand where keypresses are short relative to the inter-onset interval.

6.2 Displaying Performance Data - continuous values

In this section we will investigate displaying continuous data alongside a score. The data comes from a recording of the vocal part of the piece 'Ash' by Graham Hair. The singer is Amanda Morrison (The Sixteen, BBC Singers, The Tallis Scholars). The piece is written using a 19-tone equally-tempered scale and therefore presents a challenge to the singers to achieve the correct intonation. The presentation of the performance data will therefore show the accuracy of the intonation by displaying the results of pitch-tracking the performance. The data will be displayed as a line graph under each note showing the pitch contour over the course of the note, showing the deviation from the target pitch.

The data was recorded and processed using Douglas McGilvray's pitch tracker and audio segmenter. The segmentation was loaded from the PML file. The raw pitch contour data was loaded from the csv file produced at an intermediate stage of the processing. The segmentation from the PML file was then used to locate values within the pitch contour data.

6.2.1 The Query

The query to create the document calls the `populateDocument` function in a similar manner to that described above in Section 6.1.2, creating a document called 'ash'. The query to attach data to the document uses the `addLineGraphUnderNotes` function described in Section 5.6.2. The function operates slightly differently to the addBarGraphUnderNotes function in that it is called several times per note, each time appending a value to the Python list that is attached to each note.

The query uses several joins to create a table containing all the necessary data. Several columns are then selected out and used to call `addLineGraphUnderNotes` to append the data to the Python list for each note. The full query is given in Program 6.3.

Program 6.3: The query to attach line graphs to the score for Ash by Graham Hair showing the pitch contour of the vocal performance.

```
select addlinegraphundernotes('ash', 'pitch_contour',
        (val.work_id, val.note_id, val.voice, val.part_id, val.value) )
from (
      select
            score_notes.work_id,
            score_notes.note_id,
            score_notes.voice,
            score_notes.part_id,
            timed_data.time,
            1200*log(2, first.freq/timed_data.value)
```

```
                   − first.centsdiff as value
          from score_notes
          left join score_notes sn2 on
                  (sn2.onset = score_notes.onset + score_notes.duration
15                and sn2.work_id = score_notes.work_id
                  and sn2.type = 'pitch ')
          left join segments first on
                  (first.matched_note_id = score_notes.note_id
                  and first.matched_work_id = score_notes.work_id
20                and first.perf_id = 22
                  and first.align = 'correct')
          inner join timed_data on
                  (timed_data.metadata_id = 4
                  and timed_data.performer_id = 12
25                and timed_data.time >= first.start_time
                  and timed_data.time < first.start_time + first.duration
                  )
          where score_notes.work_id = 6
          and score_notes.type = 'pitch '
30        order by score_notes.work_id,
          score_notes.note_id,
          score_notes.part_id,
          score_notes.voice,
          timed_data.time
35  ) as val
    group by val.work_id, val.note_id, val.part_id, val.voice
    ;
```

The query uses joins in a similar way to the previous query: first joining the `score_notes` tables to itself to create a table of the first note and following note, then using subsequent joins to append the values from the performance to that data.

The first join (lines 13-16) takes all the notes in `score_notes` table which have the work identifier 6 and type ‘pitch’ (lines 28-29) and joins them with all the notes from the same table which satisfy the criteria: the following note’s onset must equal the first note’s onset plus its duration; the following note must have a work identifier of 6 and a type of ‘pitch.’

The second join (lines 17-21) joins the data from the segments table which satisfies the criteria: the first note and work identifier match; the performance identifier is 22 and the segment is correctly matched.

The third join attaches data from the `timed_data` table where the timed data identifier is 2 and the data times fall into the range defined by the start and end of the segments. Usually, a single row in the resulting table will match several rows in the `timed_data` table. Using an inner join here means that each row in the compound table will be made up of only those rows with values which meet the criteria.

At the top of the subquery, the relevant fields are selected from the compound table: work, note, part and voice identifiers. Additionally, the time of each value from the `timed_data` table is selected. This allows us to order the values by time before passing them to the `addLineGraphUnderNotes` This is needed since we cannot guarantee the order in which the values will be returned by the database.

The last value in the **select** clause is the actual value which will be plotted. It is calculated as the difference between the performed pitch and the target pitch in cents and is calculated according to Equation 6.1.

$$\text{value} = 1200 \log_2 \left(\frac{f_{\text{segment}}}{f_{\text{performed}}} \right) - \texttt{centsdiff} \tag{6.1}$$

where f_{segment} is the calculated frequency for the segment in Hertz, $f_{\text{performed}}$ is the performed frequency, also in Hertz and `centsdiff` is the difference between the target pitch and the segment pitch in cents. The segment frequency is calculated from the averaged frequency for the segment.

Finally, in the outer subquery, the selected fields are combined as the arguments to the `addLineGraphUnderNotes` function. The first argument is the name of the document to which these values are being attached: ‘ash’ in this case. The second is the name for this data set: ‘pitch_contour’ in this case. The final argument is the `spoff_value_type` Since the result of the lower subquery is a set of fields and not the compound `spoff_value_type` we could cast the values to the correct type. It is enough, however, to combine the values in an array, as done here. The `group by` clause on line 36 groups the results by the work, note, part and voice and ensures that the initial state of the `addLineGraphUnderNotes` function is reset for each note. In this way we can ensure that each set of values is only attached to the correct note.

The ‘ash’ document is then extracted in the same way as that described in Section 6.1.2 and processed with the Lilypond music typesetter.

6.2.2 The Result

The Lilypond code resulting from the query is in Appendix B in full and is summarised here in Program 6.4 in its edited form.

Program 6.4: A summary of the Lilypond markup code generated by the query in Program 6.3.

```
\version "2.12.1"

\pointAndClickOff

linegraphboxheight = 10
linegraphboxwidth = 5
linegraphscalefactor = #0.066666666
linegraphbias = 75

#(define (pslines prev_x x_inc y_list)
  (if (> (length y_list) 0)
   (format #f "~$ ~$ ~a~a"
    (exact->inexact (+ prev_x x_inc))
    (* linegraphscalefactor (+ (car y_list) linegraphbias))
    "lineto
"
    (pslines (+ prev_x x_inc) x_inc (cdr y_list)))
   ""
   ))

#(define (generate-ps y_list)
  (format #f "~$ ~$ ~a~a~a"
   '0
   (* linegraphscalefactor (+ (car y_list) linegraphbias))
   "moveto
"
   (pslines '0 (/ linegraphboxwidth (- (length y_list) 1)) (cdr y_list))
   "stroke"))
```

```
#(define-markup-command (linegraphbox layout props y_list) (list?)
  "draws a line graph from a list of values in fixed size box"
   (interpret-markup layout props
    (markup
     (#:rounded-box
      (#:combine (#:combine (#:combine (#:combine
       (#:combine (#:combine (#:combine (#:combine
        #:with-color (x11-color 'white)
         #:filled-box `(0 . ,linegraphboxwidth)
          `(0 . ,linegraphboxheight) 0
        #:translate `(0 . 5)
         #:with-color (x11-color 'Black)
          #:draw-line `(,linegraphboxwidth . 0))
        #:translate `(0 . ,(+ 5 (* 60 linegraphscalefactor)))
         #:with-color (x11-color 'LightGrey)
          #:draw-line `(,linegraphboxwidth . 0))
        #:translate `(0 . ,(+ 5 (* 40 linegraphscalefactor)))
         #:with-color (x11-color 'LightGrey)
          #:draw-line `(,linegraphboxwidth . 0))
        #:translate `(0 . ,(+ 5 (* 20 linegraphscalefactor)))
         #:with-color (x11-color 'LightGrey)
          #:draw-line `(,linegraphboxwidth . 0))
        #:translate `(0 . ,(+ 5 (* -20 linegraphscalefactor)))
         #:with-color (x11-color 'LightGrey)
          #:draw-line `(,linegraphboxwidth . 0))
        #:translate `(0 . ,(+ 5 (* -40 linegraphscalefactor)))
         #:with-color (x11-color 'LightGrey)
          #:draw-line `(,linegraphboxwidth . 0))
        #:translate `(0 . ,(+ 5 (* -60 linegraphscalefactor)))
         #:with-color (x11-color 'LightGrey)
          #:draw-line `(,linegraphboxwidth . 0))
        #:with-color (x11-color 'red)
         #:postscript (generate-ps y_list ))))))

\book {
 \score { <<
  \new Staff = "Staff 1 "
   <<
   {
    \key c \major
    \clef treble
    \time 3/4
    e''8 disis''32 e''32 eis''32 f''32
    fis''16 e''16 cis''8 e''16 cis''16 b'8
```

```
       cis''8 gis'4 ais'4 cis''16 cis''16
75    }
      \addlyrics
      {
       \markup \linegraphbox #'(-27.052078535 -33.0811246357 -37.1979962791 ...
       \markup {}
80     \markup \linegraphbox #'(24.3379305708 21.7113376389 19.2565159557 ...
       \markup \linegraphbox #'(1.32210016935 -0.252991595803 -2.39992054312 ...
       \markup {}
       \markup \linegraphbox #'(2.61163242619 -5.36683190018 -11.1096629582 ...
       \markup \linegraphbox #'(19.1605978728 19.3257335074 17.8560475439 ...
85     \markup \linegraphbox #'(-38.4968622961 -40.2087710756 -41.0455687734 ...
       \markup \linegraphbox #'(9.6387158307 2.60735871978 1.16760594764 ...
       \markup \linegraphbox #'(1.05743256154 4.15050852965 6.35099898724 ...
       \markup \linegraphbox #'(16.3793628642 18.3635650729 18.9910033344 ...
       \markup \linegraphbox #'(-5.81490477254 -7.44611253398 -8.30203904057 ...
90     \markup \linegraphbox #'(5.10151575788 5.65344559157 7.84166776491 ...
       \markup \linegraphbox #'(-15.6358081059 -17.826242782 -19.8901531136 ...
       \markup \linegraphbox #'(9.72374232529 5.57701872602 5.07919097319 ...
       \markup {}
      }
95    >>
    >> }
   }
```

As before, the Lilypond code in Program 6.4 can be considered in 2 parts: the scheme functions to draw the linegraph boxes and the Lilypond markup which describes the score and calls the scheme functions.

There are 4 variables which determine the size of the line graph: `linegraphbox-height` for height, `linegraphboxwidth` for width, `linegraphscalefactor` for scaling the line to fit in the graph and `linegraphbias` to move the graph up (required when the values dip below zero).

There are 3 scheme functions responsible for drawing the line graphs: `pslines` (lines 10-19) which returns postscript commmands to connect the points on the line; `generate-ps` (lines 21-28) which returns postscript commands to move to the origin, insert the points from `pslines` and draw the line; `linegraphbox` draws the box surrounding the line graph and calls `generate-ps` to fill it.

The `linegraphbox` function has additional code to that described in Section 5.7. Lines 40-60 draw several lines on the box. A single black line for the origin and additional grey lines to mark +/− 20, 40 and 60 cents deviation from the target pitch.

The annotated score which results from processing Program 6.4 with the Lilypond typesetter is shown in Figure 6.2.

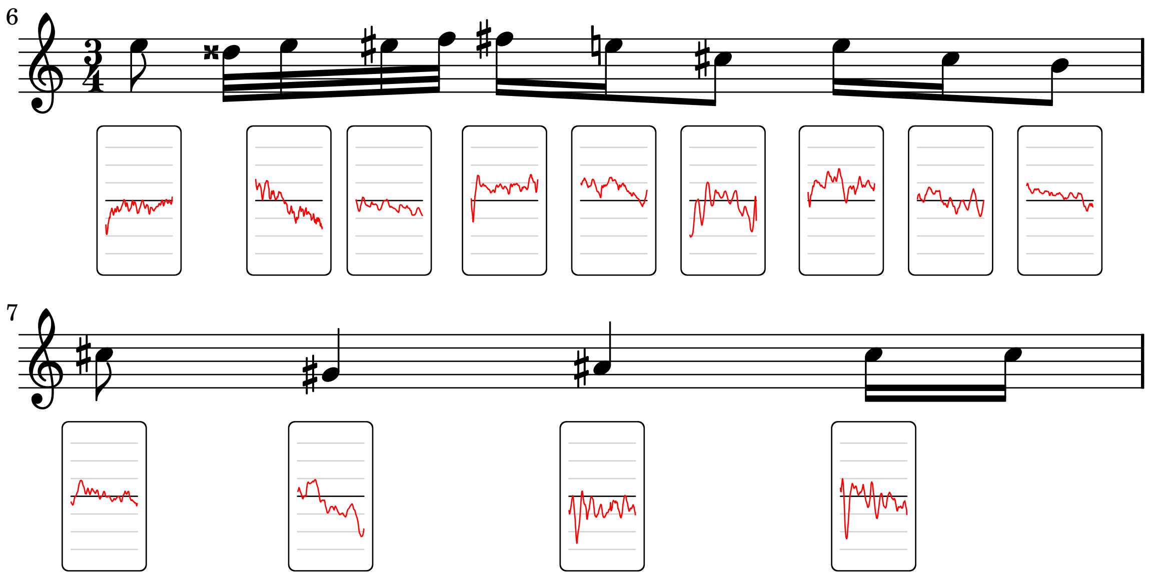


Figure 6.2: An extract from Ash by Graham Hair annotated with the singer's pitch contour. This edited extract is the result of the query in Program 6.4 and processing the extracted file shown in Program 6.3. The graphs have a heavier line for the origin and fainter lines at +/− 20, 40 and 60 cents. Plots are shown centre-aligned, underneath the note to which they belong. Where the matcher failed to match a note to a segment of the performance data, a note will have no plot. In these cases, the typesetter moves surrounding notes closer.

Figure 6.2 shows the successful annotation of a score in 19-tone equal temperament with the singer's pitch trajectory. We can clearly see which notes were sung sharp and which were sung flat. We can also see the pitch trajectory over the course of the note and amount and depth of vibrato applied. This figure shows the application of the database system to the analysis of music in non-standard tuning systems and shows the ability of the system to display useful continuous data alongside a score.

6.3 Combining Musical Queries and Performance Data

In this section we will show the combination of queries of musical information and performance data. The piece we will use to illustrate this is Bach's Two-Part Invention No. 1 for solo piano (BWV772) performed by a student at Glasgow University. The piece was recorded using the Piano Bar according to the procedure outlined in Chapter 5. The presentation will annotate the musical intervals and highlight those considered dissonant. In addition, the inter-onset interval will be displayed so that one can clearly see how the presence of a dissonance is reflected in the performance. The IOIs will be normalised (divided by the score note's duration in crotchets) to more clearly show any deviation from a regular tempo.

6.3.1 The Query

The query to create the document calls the `populateDocument` function in a similar manner to that described above in Section 6.1.2, creating a document called 'inv1'.

The query to attach note intervals and inter-onset intervals to the score consists of three parts: a query to attach the interval text to the top staff (the right hand); a similar query to attach interval text to the bottom staff (the left hand); and a query to attach the bar graphs showing the IOIs.

The first part of the query to attach intervals to the top staff uses a temporary table to store the intervals between the top staff and the bottom staff. Then the `addBarGraphUnderNotes` function is called twice: once for the dissonant intervals to be highlighted in red text; and once for the other intervals. The full query is shown in Program 6.5.

Program 6.5: A query in SQL to attach text to each note in the top staff of a score with dissonant intervals highlighted with red text.

```
begin;
```

```
create temp table bi as
(
        select  top.work_id,
                top.note_id,
                top.voice,
                top.part_id,
                getinterval(top.pitch, bottom.pitch) as value
                from score_notes as top
                inner join score_notes as bottom
                on (bottom.work_id = top.work_id
                and bottom.part_id = 'Staff 1 '
                and bottom.type = 'pitch '
                and (bottom.onset = top.onset
                or (bottom.onset < top.onset and bottom.onset + bottom.duration > top.onset)))

        where top.part_id = 'Staff 2 '
        and top.type = 'pitch '
        and top.work_id = 0
        order by top.onset, bottom.onset
        )
;

select addtextundernotes('inv1', 'intervs',
        cast(
                (bi.work_id,
                bi.note_id,
                bi.voice,
                bi.part_id,
                ' \\with-color #red ' || interval2text(bi.value)
        ) as spoff_text_type))
from bi
where equateIntervalClass(bi.value, text2interval('M2'))
or equateIntervalClass(bi.value, text2interval('P4'))
or equateIntervalClass(bi.value, text2interval('M7'))
;

select addtextundernotes('inv1', 'intervs',
        cast(
                (bi.work_id,
                bi.note_id,
                bi.voice,
                bi.part_id,
                interval2text(bi.value)
        ) as spoff_text_type))
```

```
   from bi
   where not equateIntervalClass(bi.value, text2interval('M2'))
   and not equateIntervalClass(bi.value, text2interval('P4'))
50 and not equateIntervalClass(bi.value, text2interval('M7'))
   ;

   drop table bi;

55 commit;
```

The first part (lines 3 to 23) of the query shown in Program 6.5 creates a temporary table called 'bi' to hold the intervals for further processing. The table has columns for work, note, part and voice identifiers of the notes from the top staff along with the interval between the top staff and the bottom staff pitch. The interval is calculated using the `getInterval` function which takes two values of `spoff_pitch_type` and returns an interval as a `spoff_interval_type`

Notes from the `score_notes` table are included if they fulfil the criteria on lines 18-20: the work identifier is 0; the part identifier is 'Staff 2' and the note type is 'pitch.'

The result is joined with `score_notes` using an inner join to attach the simultaneous notes from the bottom staff. An inner join is used to ensure that only those notes with a simultaneous note are copied to the temporary table. The criteria for the join are: the work identifier must match; the part identifier must be 'Staff 1'; and the note type must be 'pitch.' The final criteria select those notes whose onset is simultaneous with the top note or those whose onset is before the top note and whose offset is after the top note's onset.

It is worth noting here that these criteria do not account for those overlapping notes whose onset occurs after the onset of the top note but before the offset of the top note. These notes will be covered by the later query.

The next subquery (lines 25-37) attaches the interval names in red text for those intervals from the 'bi' table which are dissonant by calling the `addtextundernotes` function. Here, a dissonant interval is defined as any kind of 2nd, 4th or 7th. The query collects together the work, note, voice and part identifiers and the interval text from the ‘bi’ table and casts them as a `spoff_text_type` The interval text is produced by calling the `interval2text` function (line 31), which takes an interval as its argument and returns text, and prepending it with the Lilypond code for red text.

The dissonant intervals are selected by calling the `equateIntervalClass` function. This function takes two intervals as its arguments and returns true if the first interval has the same class as the second. For example, if the first interval was a minor 2nd and the second was an augmented 2nd, the function would return true as a minor 2nd is a kind of 2nd.

The call to `equateIntervalClass` makes use of the `text2interval` function to convert the string ‘M2’ to a major 2nd represented as a `spoff_interval_type` In this way, only those intervals which are a kind of 2nd, 4th or 7th are attached to the document in red text.

The next subquery is very similar to the previous only it selects those intervals that are neither a 2nd, 4th or 7th without the red text.

Finally the ‘bi’ temporary tabled is deleted (‘dropped’).

The query to attach interval text to the bottom staff (the left hand) is very similar to that shown in Program 6.5. The query is shown in Program 6.6.

Program 6.6: A query in SQL to attach text to each note in the bottom staff of a score with dissonant intervals highlighted with red text.

```
begin;

create temp table bi as
(
    select  bottom.work_id,
            bottom.note_id,
            bottom.voice,
            bottom.part_id,
            getinterval(top.pitch, bottom.pitch) as value
            from score_notes as top
            inner join score_notes as bottom
            on (bottom.work_id = top.work_id
            and bottom.part_id = 'Staff 1 '
            and bottom.type = 'pitch '
            and (bottom.onset >= top.onset and bottom.onset < top.onset + top.duration)
            )
    where top.part_id = 'Staff 2 '
    and top.type = 'pitch '
    and top.work_id = 0
    order by top.onset, bottom.onset
);

select addtextundernotes('inv1', 'intervs',
    cast(
            (bi.work_id,
            bi.note_id,
            bi.voice,
            bi.part_id,
            ' \\with-color #red ' || interval2text(bi.value))
    as spoff_text_type))
from bi
where equateIntervalClass(bi.value, text2interval('M2'))
or equateIntervalClass(bi.value, text2interval('P4'))
or equateIntervalClass(bi.value, text2interval('M7'))
;

select addtextundernotes('inv1', 'intervs',
    cast(
            (bi.work_id,
            bi.note_id,
            bi.voice,
            bi.part_id,
            interval2text(bi.value))
    as spoff_text_type))
from bi
where not equateIntervalClass(bi.value, text2interval('M2'))
and not equateIntervalClass(bi.value, text2interval('P4'))
and not equateIntervalClass(bi.value, text2interval('M7'))
;

drop table bi;

commit;
```

The query to attach interval text to the bottom staff proceeds in much the same way as the query to attach text to the top staff except for the obvious substitutions of the top staff with the bottom staff and vice versa. A notable difference is the criteria for selecting those notes in the bottom staff which overlap with the top staff. In this second query we select only those note in the bottom staff whose onset occurs at or after the onset of the top note and whose onset occurs before the top note's offset.

In this way we have covered all the possible ways in which the two notes can overlap. One might ask why we couldn't attach all of the intervals to the top staff. We could but the intervals would not be aligned with the notes they represent. We have two separate queries to create a clearer presentation of the intervals.

The final query calculates the inter-onset interval and attaches a bar graph to each note to represent it. It is shown in Program 6.7.

Program 6.7: A query in SQL to attach a bar graph representing the inter-onset interval.

```
\set performance_id 0
\set work 0

BEGIN;

select addBarGraphUnderNotes('inv1', 'ioi', data) from (
        select  first.work_id,
                first.note_id,
                first.voice,
                first.part_id,
                (seg_fol.start_time − seg_first.start_time)
                 /
                 (cast (((first).duration).crotchet_numerator as numeric)
                  /
                 (cast (((first).duration).crotchet_denominator as numeric)))
                as value

                from score_notes first
                inner join score_notes following
                        on (first.onset + first.duration = following.onset
                        and first.work_id = following.work_id
                        and following.type = 'pitch'
```

```
                    and first.voice = following.voice
                    and first.part_id = following.part_id)
            left join segments seg_first
                    on (seg_first.matched_work_id=first.work_id
                    and seg_first.perf_id=:performance_id
                    and seg_first.matched_note_id = first.note_id
                    and seg_first.align = 'correct')
            left join segments seg_fol
                    on (seg_fol.matched_work_id=first.work_id
                    and seg_fol.perf_id=:performance_id
                    and seg_fol.matched_note_id = following.note_id
                    and seg_first.align = 'correct')

            where first.work_id = :work
            and first.type = 'pitch'
            order by seg_first.start_time)
        as data;

COMMIT;
```

The query in Program 6.7 collects the work, note, part and voice identifiers and calculates the normalised inter-onset interval. This is calculated from the difference between the following note's performance time onset and the first's divided by the first note's score time duration. This calculation differs from the previous IOI calculation in that it is normalised to the score note duration. Whereas previously the IOI had been displayed as an absolute value, here we show the IOI as a deviation from a constant tempo. If the entire performance had been played at a constant tempo, the normalised IOI would give a constant value — the reciprocal of the beats per minute which is the duration of one beat. This would be the result if we analysed a computer-generated MIDI performance with the system. When we show the normalised IOI of a real performance, we can see the deviations from a constant tempo.

Since the score note durations are stored as a compound type which represents the numerator and denominator of a fraction, we divide one by the other to get a result that we can use in a calculation. First, though, each value must be cast from an integer to a numeric type to ensure that the calculation is performed with sufficient accuracy.

The values are taken from a compound table created by joining the `score_notes` with itself to extract first notes and following notes. This is then joined with the segments table twice to extract the start times of the first notes and following notes from the performance.

As before the Lilypond markup is extracted using the `getLilypond` function.

6.3.2 The Result

The Lilypond code resulting from the query is in Appendix B in full and is summarised here in Program 6.8 in its edited form.

Program 6.8: A summary of the Lilypond markup code generated by the query in Programs 6.5, 6.6 and 6.7.

```
\version "2.12.1"

\pointAndClickOff
valueboxheight = 10
valueboxwidth = 1
#(define-markup-command (valuebox layout props val) (number?)
 "Draws 2 boxes - one containing val as text, one containing val as a line graph - in a column"
  (interpret-markup layout props
   (markup
    #:center-column
     (#:override '(box-padding . 0.1)
     #:rounded-box
      (markup #:override '(font-size . -5)
       (format #f "~$" val))
        #:override '(box-padding . 0.1)
         #:rounded-box
          (#:combine
           (#:combine
            #:with-color (x11-color 'white) #:filled-box `(0 . ,valueboxwidth) `(0 . ,valueboxheight) 0
             #:with-color (x11-color 'blue) #:filled-box `(0 . ,valueboxwidth) `(0 . ,(* 8 val)) 0 )
              #:translate `(-0.5 . 0) #:draw-line `(,(+ 1 valueboxwidth) . 0))
     ))))

\book {
 \score { <<
```

```
\new Staff = "Staff 1 "
 <<
  {
   \time 4/4 \key c \major \clef treble
   e''16 c''16 d''16 e''16
   f''16 d''16 e''16 c''16
   d''16 e''16 f''16 g''16
   a''16 f''16 g''16 e''16
  }
  \addlyrics
  {
   \markup {} \markup {\tiny 1+m3} \markup {\tiny 1+M3}
   \markup {\tiny \with-color #red 1+A4} \markup {\tiny 1+m6}
   \markup {\tiny \with-color #red 1+P4} \markup {\tiny 1+M6}
   \markup {\tiny \with-color #red 1+P4} \markup {\tiny 1+M6}
   \markup {\tiny \with-color #red 1+M7} \markup {\tiny 1+m3}
   \markup {\tiny \with-color #red 1+P4} \markup {\tiny 1+M6}
   \markup {\tiny \with-color #red 1+P4} \markup {\tiny 1+M6}
   \markup {\tiny \with-color #red 1+A4}
  }
  \addlyrics
  {
   \markup {} \markup \valuebox #0.8288 \markup \valuebox #0.692 \markup \valuebox #0.8872
   \markup \valuebox #0.6376 \markup \valuebox #0.8208 \markup \valuebox #0.7544
   \markup \valuebox #0.808 \markup \valuebox #0.7336 \markup \valuebox #0.8248
   \markup \valuebox #0.8252 \markup \valuebox #0.6668 \markup \valuebox #0.754
   \markup \valuebox #0.7624 \markup \valuebox #0.6792 \markup \valuebox #0.8044
  }
 >>
\new Staff = "Staff 2 "
 <<
  {
   \clef bass \key c \major
   a8 bes8 a8 g8
   f8 d'8 c'8 bes8
  }
  \addlyrics
  {
   \markup {\tiny 1+P5} \markup {\tiny 1+M3} \markup {\tiny 1+m6} \markup {\tiny 1+M6}
   \markup {\tiny 1+M6} \markup {\tiny 1+m3} \markup {\tiny 1+M6} \markup {\tiny 1+M6}
  }
  \addlyrics
  {
   \markup \valuebox #0.679 \markup \valuebox #0.8668 \markup \valuebox #0.7874
```

```
      \markup \valuebox #0.7542 \markup \valuebox #0.8 \markup \valuebox #0.7542
      \markup \valuebox #0.6042 \markup \valuebox #0.8916
     }
    >>
  >>
}
}
```

The Lilypond code in Program 6.8 has the same structure as previous examples: a scheme section followed by a Lilypond markup section. Each staff of music is followed by two lyric lines: one holding the interval text and one holding the calls to the valuebox function which draws the bar graph of the normalised inter-onset interval. In lines 36-45, we can see the interval text highlighted in red for the dissonant intervals and un-highlighted for the consonant intervals.

The annotated score resulting from the Lilypond code in Program 6.8 is shown in Figure 6.3.

The score in Figure 6.3 shows the dissonant intervals and the normalised inter-onset interval. We can clearly see a difference in emphasis between certain notes with the notes of the dissonant intervals nearly always showing a slowing relative to their neighbours. This figure shows the application of the musical functions of the database in annotating the score. It also shows how presenting performance data and musical analysis combined can provide new insights into a piece of music and its performance.

6.4 Summary

This chapter has demonstrated the system analysing several different performances and annotating scores with the results. We have seen the system dealing with different types of performance data and music written for different tunings systems.

This chapter shows the system fulfilling all of the requirements outlined in Chapter 1.

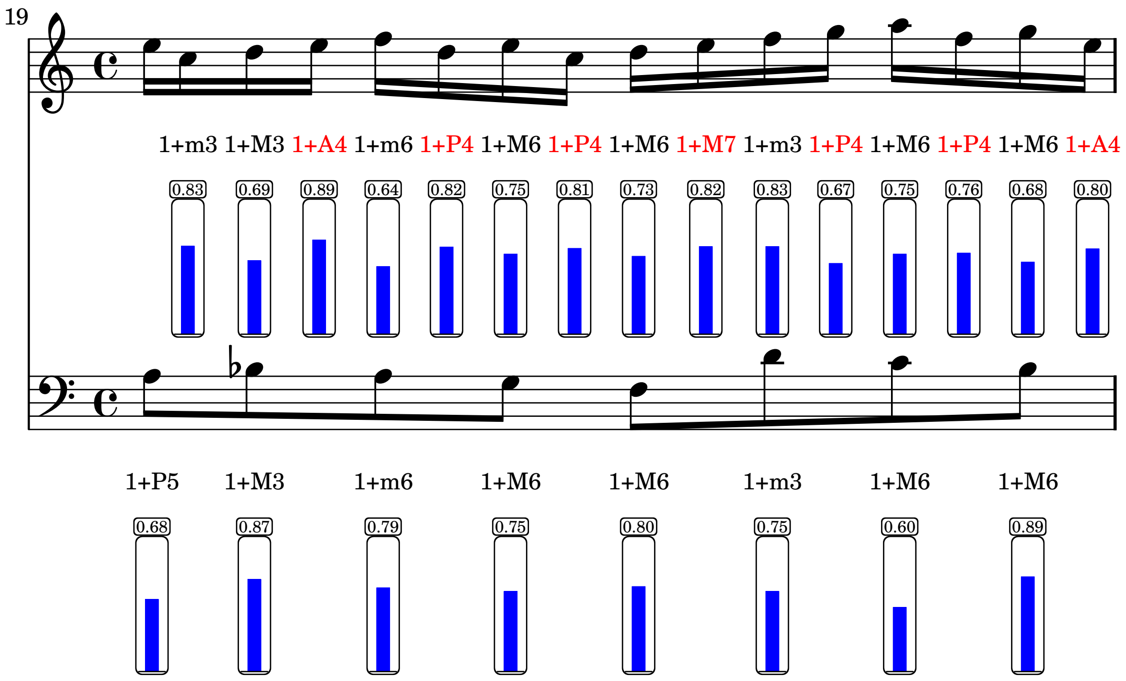


Figure 6.3: An extract of Bar 19 from Bach's Two-Part Invention No. 1 (BWV772) annotated with normalised inter-onset intervals shown in bar graphs and intervals. Dissonant intervals are shown in red. Whereas previously the IOI has been displayed as an absolute value, here we show the IOI as a deviation from a constant tempo. If the entire performance had been played at a constant tempo, the normalised IOI would give a constant value. Since we are analysing a real performance, with fluctuations in tempo, the normalised IOIs change. A semiquaver with a longer line line than a quaver here means that the semiquaver's normalised IOI was longer than the quaver's — the semiquaver's IOI was longer relative to the expected semiquaver IOI for a constant tempo than the quaver's expected IOI.

- It shows that the system has successfully stored a score in a form which allows it to be analysed and later extracted for presentation.
- It shows that the system has successfully stored performance data in a form which allows it to be analysed
- It demonstrates the system manipulating music written in different tuning systems.
- It illustrates the system being used for analysis of both musical score and performance data.
- It shows that the system is able to accept many diverse queries.

- It shows that the system is capable of displaying performance data alongside a musical score.

This chapter demonstrates the successful implementation of the representations for pitch and time (described in Chapters 2 and 3) and the functions for the manipulation of these types (described in Chapter 4). It shows that the system described in Chapter 5 can be used for useful analyses of musical performance. It demostrates that a system for the analysis of music and performance data has been created which is capable of providing new insights into musical performance.

Chapter 7

Discussion

In Chapter 1, we looked at computer systems used in music research. We found computer systems being used in 5 domains: logical, visual, gestural, analytical and phonological. The systems tend to be specialised to one domain for answering a specific research question. Few systems exist which can accept general musical queries. Looking at performance musicology, we found few systems capable of combining results from different musical domains and none which could display results alongside the score.

We made clear that the aim of the current work is to create a system which is capable of both music and performance analysis. Since no such system exists we formulated a set of requirements that a system would need to fulfill in order to be useful for combined music and performance analysis. We will look again at the requirements here and assess the extent to which the current system meets these requirements.

In [77, 80], MacRitchie *et al* investigated the correlation between movement and musical performance — in particular, how the performer's movement indicated the underlying structure of the piece. The studies recorded 9 pianists performing 2 pieces in a Vicon motion capture system as well as recording audio, MIDI and video. In [77] the performers' movements were then segmented and matched to their performance and analysed using Principal Component analysis. The study demonstrates the need for systems (such as the one decribed here) capable of analysing both scores and performance data and doing so across several performances and several pieces. In [80], the audio and MIDI data were analysed, demonstrating the need for systems which can manipulate data from multiple sources, such as the current system.

The current system is described in [118] and the proposed file format is described in [119].

7.1 Represent the score

The first requirement in Section refsec:requirements was to represent the score used in the performance without loss of data. This requirement is needed since the current system aims to enable the analysis of the relationship between the score and performance, therefore we must store the score. To enable varied analyses, we must store the score without loss of data. This is because we cannot predict what analyses will be done on the data. This contrasts some other systems, such as the OMRAS2 system [59], which only needs MIDI data so only stores MIDI data, but loses a great deal of information by so doing. The distinction is between systems designed to receive general queries (which must therefore store all the data) and those designed to perform a specific function which need only store the data relevant to that function.

The software created for the current system allows simple scores to be converted from MusicXML to the database representation and from the database to Lilypond without loss of data or accuracy. It is extensible to allow a more complete representation of music notation to be built up in time.

The representation approximates the Abstract Data Type outlined by Wiggins *et al* [123] with the exception of the timbre element. The timbre element, intended to describe the timbre and intensity of a note, could be represented as part of a more generalised structure.

The note groups in the current representation approximate Wiggins' collections. An important difference is that the note groups do not allow groups of groups only groups of notes. This has the effect of preventing the creation of hierarchical structures. The addition of a type column in the `note_groups__score_notes` table (and the renaming of it) would allow for hierarchical structures to be represented.

```
group_members [ group_id integer, member_type char[10],
work_id integer, member_id integer ]
```

where the group id is the identifier of the group, the member type will be ‘note’ or ‘group’, the work id is the identifier of the work to which the member belongs and the member id is either the note identifier or the group identifier for the member.

Whilst the database is capable of representing collections of notes in note groups, it does not have a method for representing the many articulation instructions such as staccato or fermata which might apply to a single note. The problem could be solved by creating note groups with only a single member but this is not conceptually consistent with the purpose of note groups to collect many notes. Alternatively, a new table could be created to hold note marks. (Only one table would be required as this would be a one to many relationship between one note and many marks or articulations). The new table could be as simple as:

```
note_marks [ work_id integer, note_id integer,
mark_type char[10], value[] integer ]
```

The mark type would hold a short string to describe the type of articulation mark e.g., ‘fermata’ or ‘staccato’. The value array would be available for any values specific to that type, just as it is used in note groups.

The database does not have any native representation for chords. All the necessary primitives and functions to manipulate chords are already present, however. As seen in Section 6.3.2, it is possible to establish simultaneities and name intervals using existing functions. New functions would need to be written to convert a collection of intervals into a chord name.

The music representation used in the current work successfully stores simple musical scores without loss of meaning. It stores pitch without losing important information and stores time without losing important information. But is not complete —

there are still plenty of score marks and annotations that are not currently included when a file is uploaded to the database. However, the note group facility is able to provide that function. Work remains to be done in writing the code to upload and download each score mark as a note group.

7.2 Represent different tuning systems

In [48], Blackwood suggested that the representation of microtonal music be used as a test for any music representation. This test was therefore added as a requirement to test the chosen pitch representation's validity. In the Results chapter, we saw that the database can load, store and process music written in 19-tones to the octave showing the success of the approach in the current work.

The pitch system described here is valid — it functions as a pitch representation, it is interval invariant and allows manipulation of musical data. An alternative pitch representation described in the Introduction is the Base-N representation [71]. It is valid for most cases and is capable of representing microtonal scales. It is problematic when an interval lands on a 'hole' in the representation as these gaps have ambiguous pitch.

Another pitch representation described in the Introduction was the binomial representation. It is valid — it functions as a pitch representation. However, it is probably more accurate to call the binomial system a method for creating a pitch representations — a family of pitch representations. For each number of divisions of the octave there is a separate representation of pitch with different representations for all the intervals. In the spoff representation there is a single representation for all pitches and intervals – the fraction – which decouples the pitch representation from the tuning system used. For example, a C♮ is always a $\frac{1}{1}$ or $\frac{2}{2}$ or $\frac{3}{3}$ etc. whether the piece is tuned in 12-tone, 19-tone or 24-tone. In the binomial system a C♮ is `<0, 0>` in 12-tone, `<1, 0>` in 19-tone etc.

Despite this, the spoff representation is a little esoteric. It does not match the way that we think about pitch: as an ascending sequence in pitch. Only theorists really use the spiral of fifths so this makes the representation more difficult to use.

To address this convenience functions were made to convert to and from the more familiar textual representation.

The system has the potential to represent music from non-Western traditions. The Arabic *maqām* system used in Persian music is mapped onto a 24-tone scale so could, in theory, be represented by the current system. The Persian scales use different note names to the 7 letters used in the Western tradition. Additionally, each note has a different name in different octaves. New functions would have to be written to convert between text representations and the Spoff representation.

There are scales that the current system cannot represent: any scale which does not use the 7 tones and some accidentals cannot be represented. For example, Harry Partch created a scale based on small integer ratios with 43 tones to the octave [99]. The system also cannot represent scales which are not octave based. For example, the Bohlen-Pierce scale uses the compass of an octave and a fifth and divides it into 13 tones.

So the current pitch representation system can represent a large family of scales but not all scales. It can represent a larger family of scales, and represent them more succinctly, than other representations in use elsewhere. Although it may not be as easy to learn as other representations, conversion functions are provided so that a user should never have to use the representation directly if they do not wish to.

7.3 Represent the performance

The third requirement in Chapter 1 was to store the performance. Since the current system aims to enable analysis of music and performance data, a representation of the performance must be stored.

The system stores two types of performance data: sampled data in the `timed-_data` table and intervallic data in the `segments` table.

In Chapter 1, the Gesture and Music Signal (GMS [88]) file format was introduced. It stores performance data in a number of hierarchical streams made up of channels, units and scenes. Storing GMS data in the `timed_data` table, loses this structure.

A system of similar to the note groups used with the score notes might allow the representation of the GMS structure and others in the performance data.

The system uses Performance Markup Language (PML) as the principal method of loading intervallic/segmented data into the database. Although other formats exist for storing scores and other formats exist for storing performance data, there are no formats which store score performance matches together other than PML. Also, it is the format outputted by the software used to accomplish score-performance matching. The system can load scores in MusicXML format and the database can load performance data in Comma Separated Value (csv) text files.

Once the data is in the database, only a small amount of additional information about each performance is stored. This is sufficient for the small number (< 50) of performances currently stored in the database as there are not many cross-performance comparisons that can be made. For a larger collection of performances, it would be advantageous to store more data about the data (metadata) to enable stronger distinctions to be made. Data such as the performance location, date, time, instrument and equipment used could all be added to the performance table as extra columns.

7.4 Combined Analysis

The fourth requirement was that the system must enable analysis of musical scores, performance data and scores and performance data combined. Since the system is intended to enable analysis of the process of musical performance, including how musical structure is reflected in performance and how performance indicates musical structure, the system must enable the analysis of both datasets together.

Other computer systems for computer music analysis tend to be specialised toward one type of analysis rather than their combination. For example, the HumDrum [74] music analysis system does not provide the ability to analyse and display performance data; the EyesWeb [35] system displays performance data but cannot analyse or display music notation. By providing the tools for the analysis of musical scores and performance data along with the means to present results, the current system aims

Program 7.1: An SQL query to obtain all the notes of the piece with work identifier 6 and order them by onset time within each part (horizontal time-slicing).

```
select * from score_notes as note
where note.work_id = 6
order by note.part_id, note.voice, note.onset;
```

to enable investigation into the process of performance which would not be possible using other systems.

The fourth requirement outlined several specific criteria which would enable the system to analyse music and performance data together. The criteria included the ability to represent musical structure such as phrases. This is provided by the note groups and could be extended to enable hierarchical structures with the alterations described above. The second and third criteria were to represent segmentation of performance data and match these segments to notes in the score. This is achieved using the matching software written by Douglas McGilvray [58] which creates PML files containing segmented and matched performances which are uploaded to the database. The fourth and fifth criteria were to enable the manipulation of the data by the computer and make the data queryable — that is enable the creation of many different queries rather than only a pre-programmed set. These criteria are satisfied by the creation of musical functions which extend the database's SQL query language.

One of the advantages of using a database with a query language over other representations is that it is simple to change between different methods of time slicing (time interlacing). Most music contains several parts consisting of many notes at different times. For some applications, such as melodic analysis, it is preferable to order the musical notes sequentially in each part. This is called horizontal time-slicing and can be achieved with the SQL query in Program 7.1.

The `order by` SQL command is not available on composite types (such as the `spoff_time` type used for the note onset) unless the complete set of ordinal operators has been defined (<, >, <=, >=, =). Once this has been done, a comparison function is created which returns -1 if a the second argument is less than the first, $+1$ if it is greater and 0 if it is equal. The functions are combined into an operator class. The operator class enables the `order by` command and allows B-tree indices to be created, increasing performance of the database.

For some applications, such as harmonic analysis, it is preferable to order musical notes by time across parts — to group all the notes which occur simultaneously. This is called vertical time-slicing and can be achieved with the SQL query in Program 7.2.

Program 7.2: An SQL query to obtain all the notes of the piece with work identifier 6 and order them by onset time across parts (vertical time-slicing).

```
select * from score_notes as note
where note.work_id = 6
order by note.onset, note.voice, note.part_id;
```

Using a database for the current application has allowed us to slice time horizontally and vertically with ease — an operation which is not easy to achieve using other representations — enabling different analyses to be performed on the same data.

The musical functions described in Chapter 4 implement all the functions for manipulating the Abstract Data Type — the ordinal operators for pitch and time, the `getInterval` and `addInterval` functions and addition and subtraction of time.

Additionally, most of the functionality of the CHARM specification [70] is implemented: CHARM *constituents* are equivalent to note groups, specific functions for getting and setting values are not needed; similarly, functions for defining *streams* (horizontal time-slices) and *slices* (vertical time-slices) are not needed. The ability to define amplitude is not present though this could be implemented through the note-mark method described above.

The musical functions could be used to encode the $\mathcal{SPP}$ (Structured Polyphonic Patterns) grammar outlined in [91]. $\mathcal{SPP}$ is a grammar for describing sequences and simultaneities in polyphonic music. An interesting application of the current system would be to implement the $\mathcal{SPP}$ grammar in SQL.

The final criterion is to present the results in the context of the score. Lilypond is used in the current system to provide high-quality typeset music and to draw graphs of performance data. Whilst there are several programs available which are capable of typesetting music and several which are capable of drawing graphs there are few (if any) other than Lilypond that are capable of doing both.

The problem is usually that the output format for most typesetting programs (including Lilypond) is a purely graphical — that is, all musical meaning is lost in the conversion from logical score to graphical score. The output formats consist only of instruction as to where to draw lines or pixels, not the musical pitch of the notehead being drawn. This makes it almost impossible to trace back which notehead belongs to which note and, therefore, which graph should be drawn where.

Early in the current work, partially successful attempts were made to change Lilypond's output routines to include some metadata in it's output that would allow the tracing of output glyphs back to the originating items in the score. The routines of Lilypond's Scalable Vector Graphics (SVG) output were altered to include metadata (in this case the note identifier from the database) read from the source file. SVG is an XML-based language that allows for custom metadata to be included inside all graphics entities. The SVG file was then processed by a Python script which drew the performance data graphs on the score aligned with the correct notes.

This approach, though successful, suffered where the graph boxes clashed with the music notation. A better solution would be to specify the performance data graphs in advance of the typesetting and let the typesetting software resolve any clashes. This is the approach used in the current system.

It is clear from the results in Chapter 6 that Lilypond is capable of high-quality musical output. It is licenced under the GNU GPL [11] which has several advantages in the current work: it allows us access to the source code to extend the functionality of the software; and it helps us to fulfil Requirement 5 — that the system should be accessible on as many different platforms as possible.

By fulfilling these criteria, the system has been created to enable the analysis of music and performance data combined. By representing musical structure we enable music analysis. By representing segmentation of performance data we enable performance analysis. By representing the correspondances between score and performance and by including functions to the score and making the whole system queryable we enable the analysis of the score and performance data combined. Finally, by presenting the results in the context of the score we allow results to reach the widest audience.

7.5 Accessibility

The fifth requirement states that the current system should run on, and be accessible from, as many different operating systems as possible. This requirement was motivated by Lamere's study [33] which showed no single preference for an operating system among the MIR community. In order that the system is of most use to as many researchers as possible, this requirement for accessibility was formulated.

The database chosen for the current system (PostgreSQL) has a long history (over 20 years) and a wide user and developer base. It therefore represents a stable and mature platform on which to develop the current system.

The PostgreSQL database is free software and is released under a BSD-style licence. It is available for all major operating systems (Windows, Macintosh, Linux, BSD), as is the Python programming language used in the current system. PostgreSQL can be accessed and extended in many different languages. All these make it a suitable choice to fulfil Requirement 5 to allow access from different operating systems.

The MySQL database is arguably more popular than the PostgreSQL database and also fulfils the requirements above. However, at the time that the PostgreSQL database was chosen for the current project, MySQL did not have the capability to be extended through database functions.

So the software written for the current system runs on the major operating systems in use by the MIR community and is available under a liberal licence. The system utilises the matching software of Douglas McGilvray which is currently only available for the Linux operating system. For the complete system to satisfy this criteria, the matching software would need to be available on all platforms.

7.6 Interoperability

The sixth requirement was for the system to interoperate with existing software. We have seen from Lamere's study [33] that there are a diverse range of tools in use by the MIR community. To be of most use to the MIR community, the current system should attempt to interoperate with them where possible.

The system supports taking score input from MusicXML files. MusicXML was chosen, among other reasons, because it is well supported among the most popular music typesetting and sequencing programs in use in the MIR community [33]. The segmented and matched performance data is taken from PML files. Whilst these are not widely supported, there is no other interchange medium for matched performances and, since PML is an extension of MusicXML, it is relatively easy to add PML support to other software.

The system is capable of outputting presentations to Lilypond format. Once this file is processed, the presentation is available in a number of different formats: PDF, PNG and SVG would be the most popular. Using the SQL `COPY` command, one can output query results to formatted text files including the popular CSV file format. The system does not currently output to PML which would be the only format capable of transporting the score and performance data to another application.

As part of investigations into appropriate target formats for the current system, a new format was proposed which could combine the PML file with the original performance data and graphical score whilst retaining compatibility with existing software. The format is called Music and Gesture File (MGF) and is introduced in [119].

So the current system supports widely used formats for entering scores to the system and producing results. The format used for entering segmented and matched performances is not widely used but has been designed so that it is relatively easy to support.

7.7 Understandable

The final requirement was that the interface to the current system be understandable to programmers and musicologists alike. Since the system aims to enable analysis of musical performance, and this analysis is of most use to musicologists, it is preferable that the system could be used by musicologists directly rather than relying on a programmer to interpret their wishes.

7.7.1 Complexity of Query Language

The SQL query language is the most widely used language to query databases. However, it is not widely used in musicology and music information retrieval — two communities who would most benefit from the current work. Language bindings to the PostgreSQL database are provided for a large number of languages which enable users who do not wish to use SQL directly to use the language of their choice.

The current system uses a relational database to represent music and performance. Others [54] have argued that an object-oriented approach is more appropriate to music representation and that an object database would, therefore, be a more appropriate choice for the current application. So why choose a relational database?

One only has to visit their high-street book store to see that relational databases and SQL are the most popular database technologies. Choosing this technology makes the system more accessible to more people and better fulfils Requirement 7 to use a data representation which can be understood by as many people as possible than an object database.

7.7.2 Database Design

Much of the detail of the musical score is stored in the note groups such as bar numbers, key signature and time signature. Currently, the mechanism for querying this data involves two joins and is quite verbose. An example query to add bar numbers to notes is shown in Program 7.3.

Program 7.3: An SQL query to find all the notes in the piece with work identifier 6 and to append the bar number of each note to its row.

```
select  note.work_id,
        note.note_id,
        note.voice,
        note.part_id,
        note.type,
        note.pitch,
        note.onset,
        note.duration,
        note_groups.value[1] as bar_number
        from score_notes as note

left join note_groups__score_notes as ngsc
on (note.work_id = ngsc.score_note_work_id
and note.note_id = ngsc.score_note_note_id)

inner join note_groups
on (note_groups.id = ngsc.note_group_id
and note_groups.type = 'measure')

where note.work_id = 6;
```

The query in Program 7.3 first performs a left join to attach all the note groups to which the score note belongs. The second join is an inner join and leaves only those rows which have a note group type 'measure.' Finally, the appropriate fields are selected.

The query could be greatly simplified if a few extra functions were defined to allow searching for or testing a note group from the note. For example, a function such as:
`getNoteGroupValue(note score_notes, group_id integer, index)`
which returns a value from the note group's value array, could be combine with one such as:
`getNoteGroupID(note score_notes, note_group_type char[10])`
which returns a note group's identifier, to create a simplified query as in Program 7.4.

Program 7.4: An SQL query to find all the notes in the piece with work identifier 6 and to append the bar number of each note to its row using the proposed new functions.

```
select  note.work_id,
        note.note_id,
        note.voice,
        note.part_id,
        note.type,
        note.pitch,
        note.onset,
        note.duration,
        getNoteGroupValue(note, getNoteGroupID(note, 'measure'), 1) as bar_number
        from score_notes as note

where note.work_id = 6;
```

A query such as that suggested in Program 7.4, which replaces two joins with two function calls would probably be far less efficient to execute than the equivalent query created with joins. The existence of the new functions would not, however, prevent the user from forming a query using joins. It would have to be a decision for the user to trade off ease of use against query performance.

The inclusion of an array to hold data values pertaining to note groups is a compromise to reduce the number of tables in the database. It is not optimal because it violates one of Honing's criteria that data should be explicit. There is nothing explicit which defines what the second value of a measure group array should represent — the user just has to know. Ideally a note group entry should point to a separate table, one for each type, which holds the required data in separate, well-defined fields. However, this would require many additional tables in the database which the current solution avoids. The current solution also places the restriction that note group values must be integers. In future, the note groups should be separated into their own tables.

The current database design has separate tables for composers and performers. This is satisfactory for the queries described in Chapter 6 but could lead to problems later. If a composer was to perform their own work, there would be two entries for them in the database — one in the composer table and one in the performer table. This does not correctly reflect the real world: in the real world there is one person who fulfils 2 roles. This relationship would be better represented with a single table to store people and separate tables to store the roles they carry out.

So the current system uses a relational database and the SQL language to allow easy access to the data using well understood, popular technology. The addition of extra convenience functions would make the process of writing queries easier but users still have to be computer literate. The interface to the data is probably not understandable by all but the most computer literate musicologists. It was beyond the scope of this work to create a graphical user interface to the system — such an interface would require a great deal of work. It is left to future work to define and build a graphical interface.

7.8 Further Work

The system outlined in the previous chapters succeeds in fulfilling nearly all of the requirements in Section 1.4. The only requirement which is partially fulfilled is the requirement to provide an interface which is understandable to both musicologists and programmers. The current interface is suitable for programmers and computer literate musicologists. The underlying representation used and the interface presented to the user satisfies Honing's criteria. It is explicit in its representation of time, able to represent all the types of data structure and balances the ease of comprehension of dedicated primitives with the flexibility of a generalised system.

There remain areas where the design of the system could be improved. This section will outline some of those areas and suggest solutions which could be implemented by others.

7.8.1 Database Design

The database aims to represent the important parts of the musical score and associated data without too much complexity. The representation used is sufficient to represent simple scores as can be seen from the results in Chapter 6. More types types could be added to properly represent a more complete set of musical notations. More tables could be added for other applications such as the storage of audio and video data.

The segments table holds a great deal of information with many of its fields optional. For example, when storing a vocal performance, it is not relevant to use the `midi_velocity` field. Likewise, when storing a MIDI performance, it makes no sense to use the `centsDiff` field to store the difference in intonation. There are, however, some fields which the two performances have in common: both need a start time and duration and matched note and work identifiers. It is therefore felt that it would make more sense to split the table into several smaller tables. This could be achieved through the PostgreSQL table inheritance mechanism. A common table could be defined with the basic fields required by all segment tables. Each additional segment table would then inherit this set of basic fields and add their own. This configuration would protect the segment table from becoming bloated with extra fields for each new type of performance data.

7.8.2 Performance Time

The use of milliseconds for the representation of performance time is the most obvious choice as it is easily converted to from most of the popular sample-based performance representations and is the most easily understood representation of time for the system user.

The database type used to store the performance time is `numeric` which has the highest accuracy possible in the database — able to store numbers with up to 1000 digits of precision. There is a database type intended for storing time — the `interval` type. This has 6 digits of precision so the `numeric` type was chosen for its increased precision.

No amount of fractional precision, however, is able to represent a number with a recurring fraction such as $\frac{1}{3}$. This limitation could create problems when the `numeric` type is used to represent performances at some sample rates. For example, a sample rate of 48kHz has a sample interval of $0.0000208\bar{3}$. A fraction data type (which stored

Program 7.5: An SQL query to find notes in a piece located between bars 1 and 6 using the proposed getBarNumber function.

```
select * from score_notes as note
where note.work_id = 6
and getBarNumber(note) >= 1
and getBarNumber(note) <= 6;
```

a numerator and denominator) would be a solution to this problem. PostgreSQL does not have such a type and its implementation would require some work in redefining all the mathematical and logical operators for the new type. Such work would be advantageous to the current system for the performance time representation, the score time representation and the pitch representation.

7.8.3 Score Time

The score time representation and its associated functions were used successfully to locate notes and to compare durations. The representation is successful in retaining the original meaning of the musical representation and its associated logic.

In Section 6.1, two methods of locating a music by the bar number were shown: one where the location of the bar was known in the spoff time representation and one where the location was computed from a subquery. It would be rare for a user to know the exact location of a bar in the spoff time representation, so this method does not demonstrate the normal usage of the database. The other method (using a subquery) should be considered the proper method.

The subquery method is rather long for a method of locating notes that is second nature to a person used to using musical scores. It is a strong candidate for conversion into a database function:

```
getBarNumber(note score_notes)
```

The function takes one argument: the note to query and returns the bar number for that note. The function could then be used in a query such as the one shown in Program 7.5.

An alternative to the above scheme which is, perhaps, obvious, is to include the bar number as an extra field in the `score_notes` table. This would simplify finding bar numbers over the solution above. It would not, however, be consistent with the data model. A bar number is not a property of a note like pitch or duration; rather, a note belongs to a bar. Thus representing a bar as a note group to which a note belongs is consistent with the data model.

The bar number was deliberately excluded from the representation of time to prevent problems when locating notes which sound together in music where two parts are in different or changing time signatures. The current time model allows notes to be located regardless of the time signature of the current or previous bars. This model does, however, assume that the two parts have the same tempo.

To locate notes which sound together in music where two parts have different tempi would require additional information. If the two parts had different but constant tempi, the tempo of a part could be represented with a note group. A new function could be created to calculate a tempo-adjusted note location from a tempo represented in beats-per-minute (BPM). Equation 7.1 shows the calculation this function would implement.

$$t_{\mathrm{tempo}} \leftarrow b_{\mathrm{BPM}} \frac{t_{\mathrm{n}}}{t_{\mathrm{d}}} \tag{7.1}$$

where t_{tempo} is the tempo-adjusted location, b_{BPM} is the tempo in beats-per-minute and $\frac{t_{\mathrm{n}}}{t_{\mathrm{d}}}$ is the time location as a fraction.

Such a system would require a mapping to be made between the more common musical tempo instructions, such as *Lento* or *Allegro*, and BPM values. Such a mapping is not a trivial undertaking as the values will depend on many factors and will be different for different styles and different historical periods.

Further complication would be introduced by changing tempi in several parts such as can be found in many works by the composer Steve Reich. A method of defining start and end tempi or start tempo and rate of change for a tempo group would be required to represent tempo instructions such as *Accelerando* and *Rallentando*. This would further complicate the above equation for calculating tempo.

Although the current system does not represent tempo accurately, it enables the necessary research that would be needed to investigate different performers' playing speeds in response to different score instructions and establish a mapping between tempo instructions and beats per minute.

7.8.4 MGF

The current system outputs presentations to widely used graphical formats such as PDF or SVG. It has been noted that the conversion of a score to a graphical format is a lossy one. The true meaning of the notes is lost from the digital file when they are converted to a glyph with a coordinate. A solution to this problem has been proposed which is the Music and Gesture File format [119].

MGF aims to integrate data in formats which are already in use and which applications already support. By combining PML and other data sources into a widely used compression format, we encapsulate the multifarious data into an easily transported form whilst preserving the accessibility of that data.

The proposed container format extends the MusicXML 2.0 specification [21] to include data from other sources inside a compressed archive. (It is worth noting that while additional data is added to the file, it remains a valid MusicXML 2.0 file). The data is integrated through the inclusion of a Performance Markup Language (PML) file which relates notes in the score to locations in the other files.

The MusicXML 2.0 specification [21] extends the more widely used MusicXML 1.1 specification introduced here [20]. Of particular interest to the current work is the compressed format which stores data in a JAR (JAVA archive) file (compatible with the popular zip format) with the addition of an index file called container.xml under the META.INF/ folder. The JAR container allows for the inclusion of files which are not MusicXML files.

The use of zip/JAR as the container format is particularly attractive since it is widely supported in file browsers and operating systems. This enables access to the contained data even where there is no software which supports MusicXML 2.0 or MGF. It also simplifies the extension of software to support the file format.

A basic MGF file consists of a MusicXML file, a Performance Markup Language (PML) file which references it and a META.INF folder containing a “container.xml” file. These files are compressed into a ZIP archive. For music with no score the MusicXML file can be empty. An overview of the structure of an MGF file is given in Figure 7.1.

The MGF file can store different representations of the score, audio and video. Formats were chosen for their openness and the ability to store extra metadata in them — providing locations for the PML files to point to.

The MGF file is a valid ZIP file, a valid JAR file and a valid MusicXML 2.0 file at the same time providing access to the data it contains from many different applications. A user would not have to have an application which could read all of an MGF file to have access to some of the data it contains.

7.8.5 OSF

After the publication of [119], a consortium of organisations including Yamaha released a specification for the Open Score Format (OSF [24]) which has similar aims. The format uses the MusicXML 2.0 specification to provide “A package format for combining digital scores with other media assets such as HTML, video, audio and MIDI into a single distribution package.”

The two formats could coexist, with MGF being built on top of and extending OSF. This would provide another set of applications which MGF and the current system could interact with fulfilling Requirement 6 for interoperability.

So the current system has broadly fulfilled the requirements of Chapter 1, but work remains to be done in improving the database design and the output format of the system.

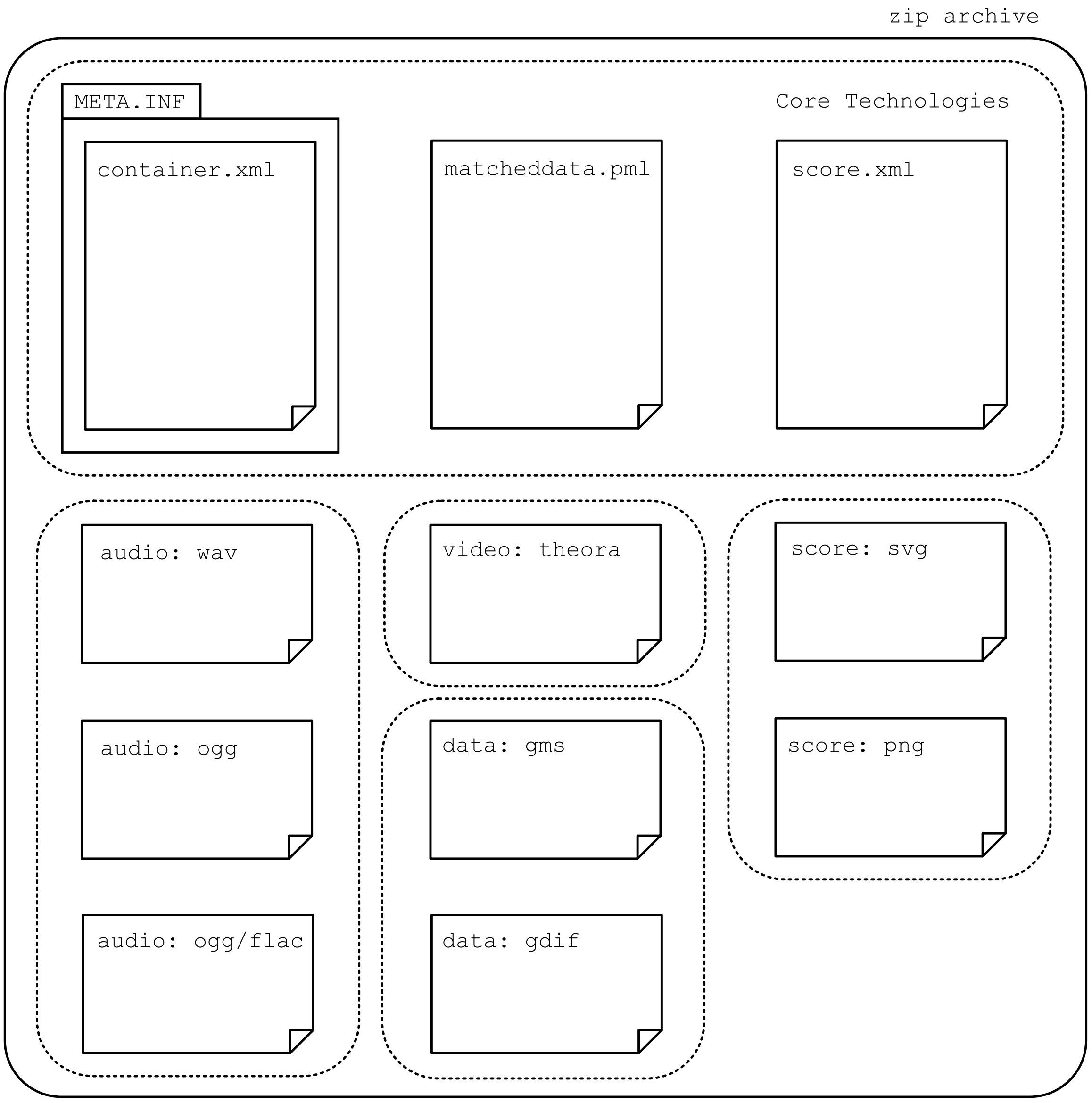


Figure 7.1: The structure of an MGF file showing the core technologies and preferred representations.

7.9 Summary

In this section we have looked at the extent to which the current system meets the requirements outlined in Chapter 1. In doing so we have assessed the music representation; the suitability of the pitch representation to microtonal music; the representation of performance data; the capability to combine music and performance analysis and display results; the accessibility of the software from different computing environments; the ability of the current system to interoperate with other systems; and the extent to which a musicologist could use the system.

In all assessments except one we have found the current system to be able to meet the requirement, often surpassing the capability of other systems. The only requirement which the current system does not fully meet is to be understandable to all musicologists. The interface to the system is more appropriate to programmers and computer literate musicologists. Further work would be needed to create a graphical interface which is easier to use.

Finally we looked at improvements that could be made to the database design and the output format of the system.

Chapter 8

Conclusion

The current work has described a system which is capable of combining musical and performance queries and displaying results annotated on the score. The need for such a system is clear if we are to enlighten the process of performance — that is the method used by a performer to turn a page of printed music into a musical performance.

The current work has presented a new extended pitch representation capable of representing microtonal scales and has explained the necessary methods to perform music analysis using the representation. A database system has been described which stores and analyses music and performance data and is capable of annotating scores with performance data.

Finally the system was tested using several different analyses demonstrating the ability of the system to display single values and continuous data alongside the score; to manipulate tonal and microtonal music; and to calculate analyses from score data and performance data.

The system has been shown to be capable of fulfilling all the requirements of a system for musical performance analysis and been demonstrated creating novel analyses of musical performance.

Appendices

Appendix A

Recording Equipment Inventory

Equipment	**Manufacturer** **Model/Version**	**Description**
Microphone	Beyerdynamic MCE 82 N(C)	Stereo, condenser microphone with balanced XLR connection
Piano Bar	Moog Music Piano Bar	MIDI augmentation for an acoustic piano
Audio/MIDI interface	Tascam US122	USB 1.1 Audio and MIDI interface with mic pre-amps
Laptop Computer	Toshiba Satellite Pro L10	1.6GHz Celeron processor, 1Gb RAM
Operating System	Kubuntu Linux 8.04/Hoary Hedgehog	Kernel: 2.6.24, KDE: 3.5.10
MIDI sequencer	Rosegarden 1.6.1	MIDI sequencer
Audio software	Ardour 2.3	
Audio sync	JACK 0.109.2-1ubuntu1	
Digital Camera	AVT Guppy F-046C	High frame rate video camera; ROI; 50 fps; requires separate PSU
Firewire interface	Belkin P81800 / (F5U513)	PCMCIA Firewire interface
Video recorder	Coriander 1.0.1-3.2build1	Software to control and record from IEEE1394(Firewire) camera

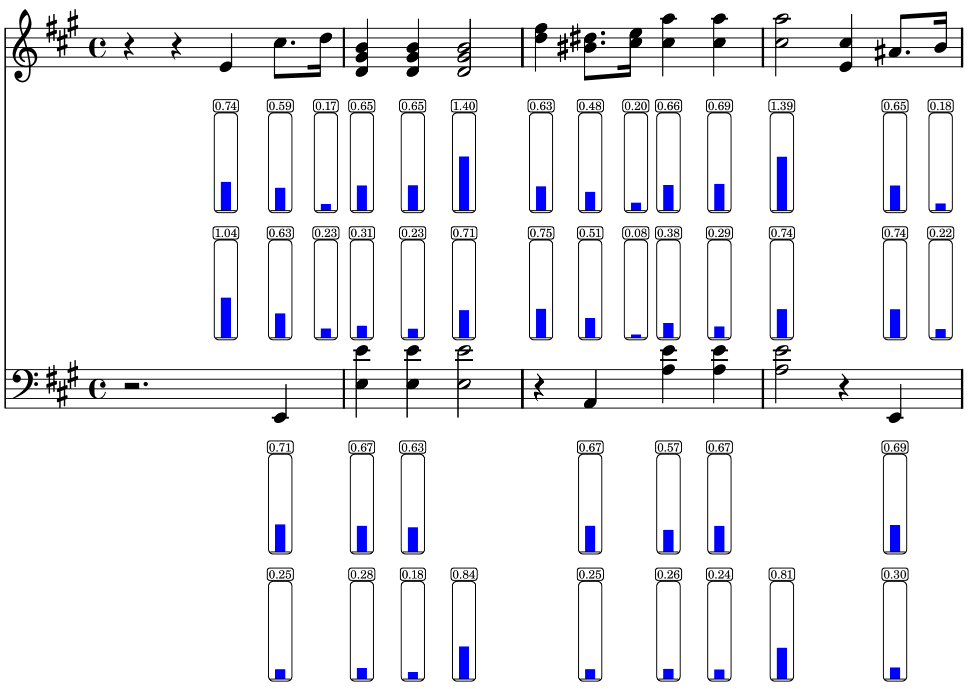
Table A.1: Inventory of equipment used in a typical recording setup

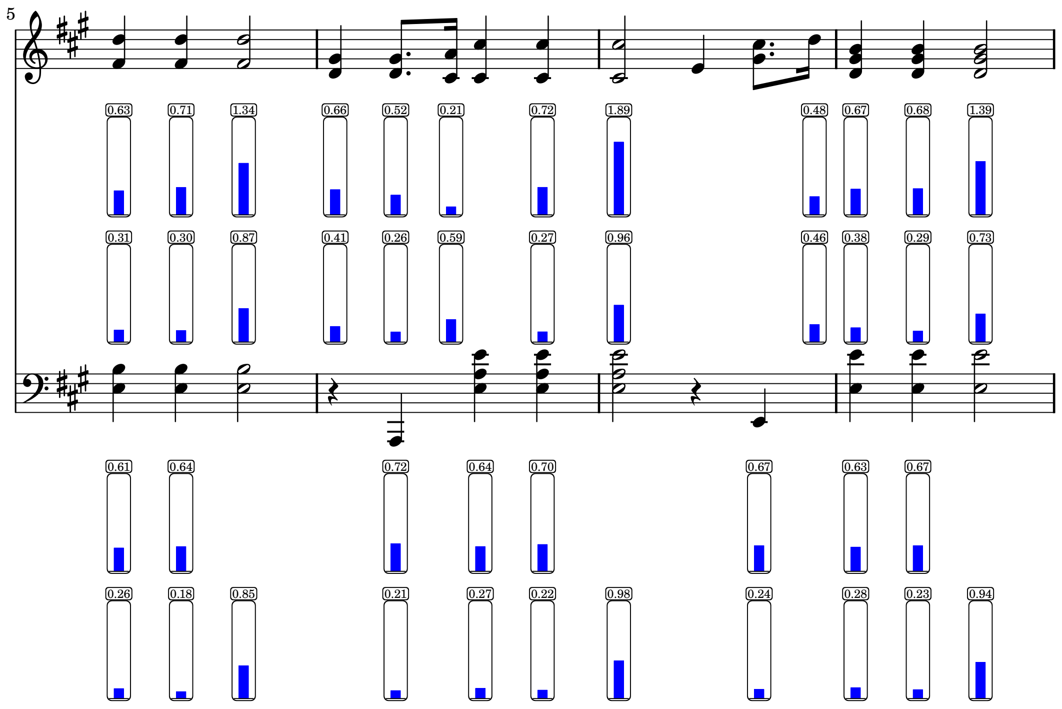
Appendix B

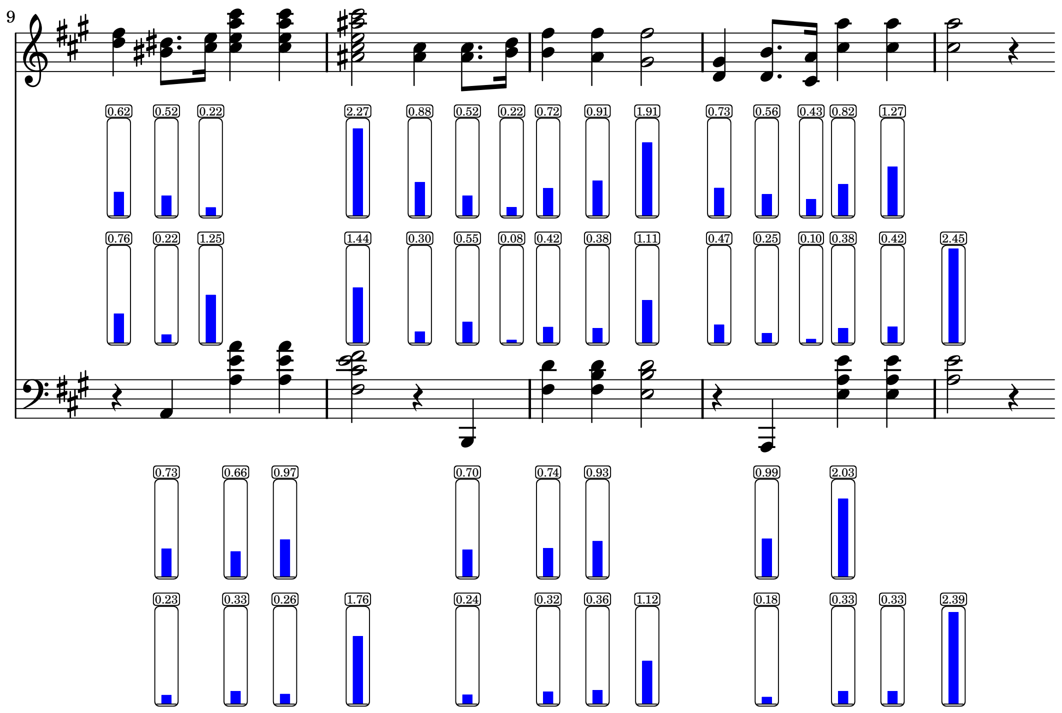
Lilypond Results

B.1 Displaying Performance Data - single values

A presentation of a performance of Chopin's Prelude No.7 by Martin Jones. The score is annotated with 2 lines of graphs of performance data: the top line shows the inter-onset interval and the bottom line shows the keypress duration.


0.74 0.59 0.17 0.65 0.65 1.40 0.63 0.48 0.20 0.66 0.69 1.39 0.65 0.18
1.04 0.63 0.23 0.31 0.23 0.71 0.75 0.51 0.08 0.38 0.29 0.74 0.74 0.22
0.71 0.67 0.63 0.67 0.57 0.67 0.69
0.25 0.28 0.18 0.84 0.25 0.26 0.24 0.81 0.30

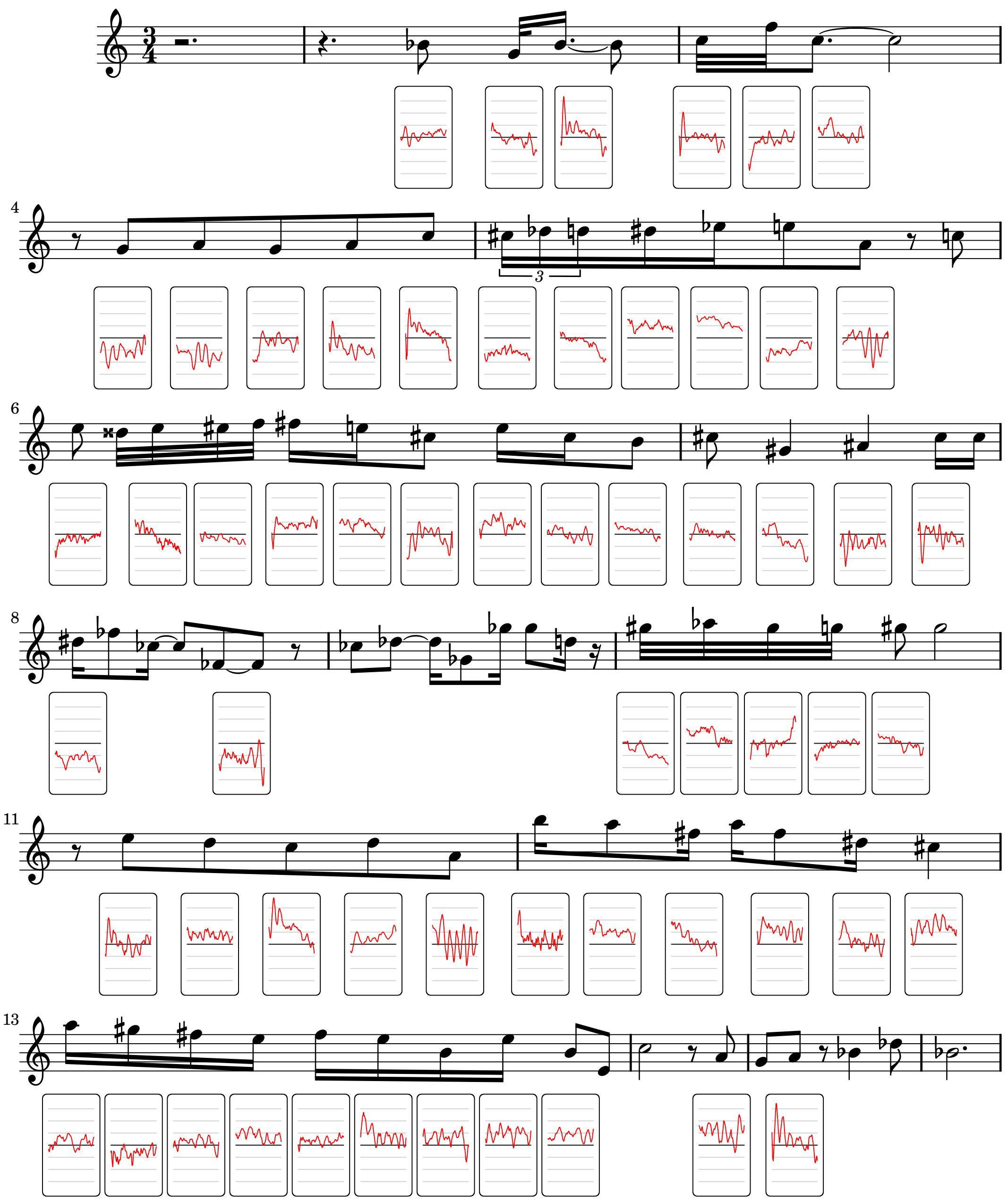

5
0.63 0.71 1.34 0.66 0.52 0.21 0.72 1.89 0.48 0.67 0.68 1.39
0.31 0.30 0.87 0.41 0.26 0.59 0.27 0.96 0.46 0.38 0.29 0.73
0.61 0.64 0.72 0.64 0.70 0.67 0.63 0.67
0.26 0.18 0.85 0.21 0.27 0.22 0.98 0.24 0.28 0.23 0.94



Music engraving by LilyPond 2.12.1—www.lilypond.org

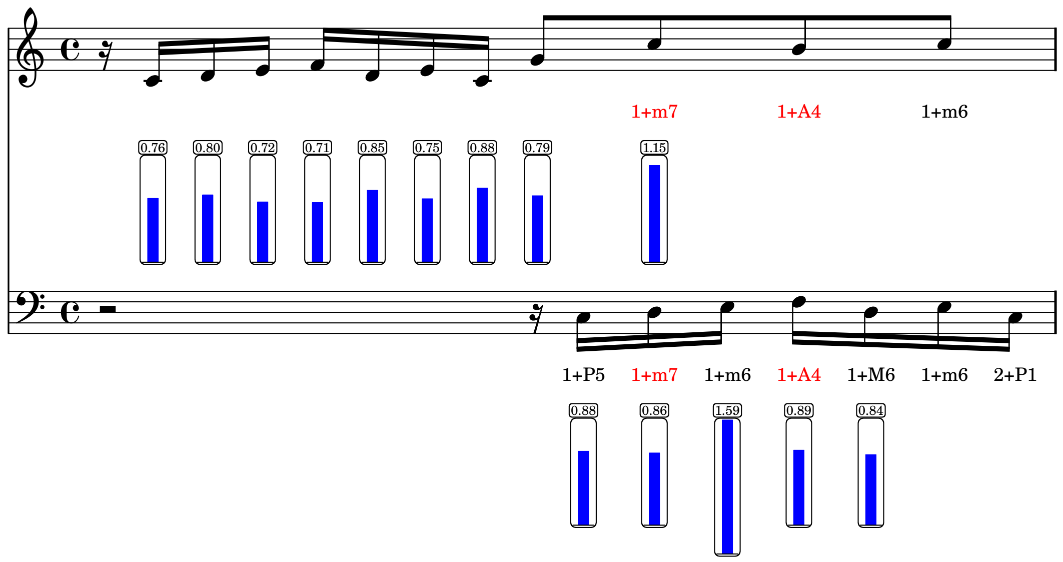
B.2 Displaying Performance Data - continuous values

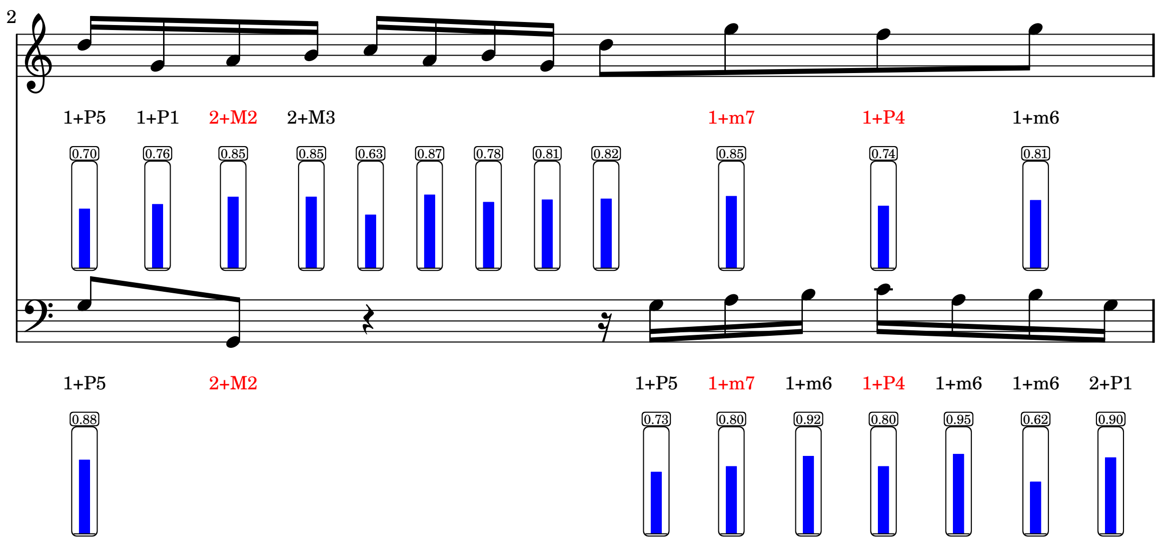
A presentation of a performance of Graham Hair's 'Ash' by the soprano Amanda Morrison. The score is annotated with bar graphs showing the pitch contour for the duration of that note. The graphs have a heavier line for the origin and fainter lines at $+/-$ 20, 40 and 60 cents. Plots are shown centre-aligned, underneath the note to which they belong. Where the matcher failed to match a note to a segment of the performance data, a note will have no plot. In these cases, the typesetter moves surrounding notes closer.

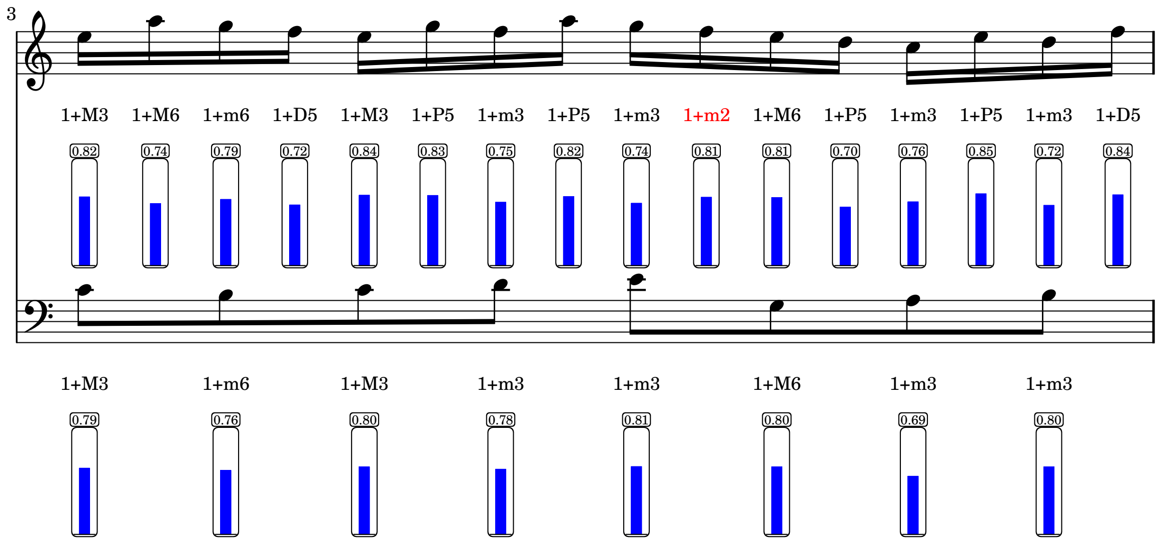


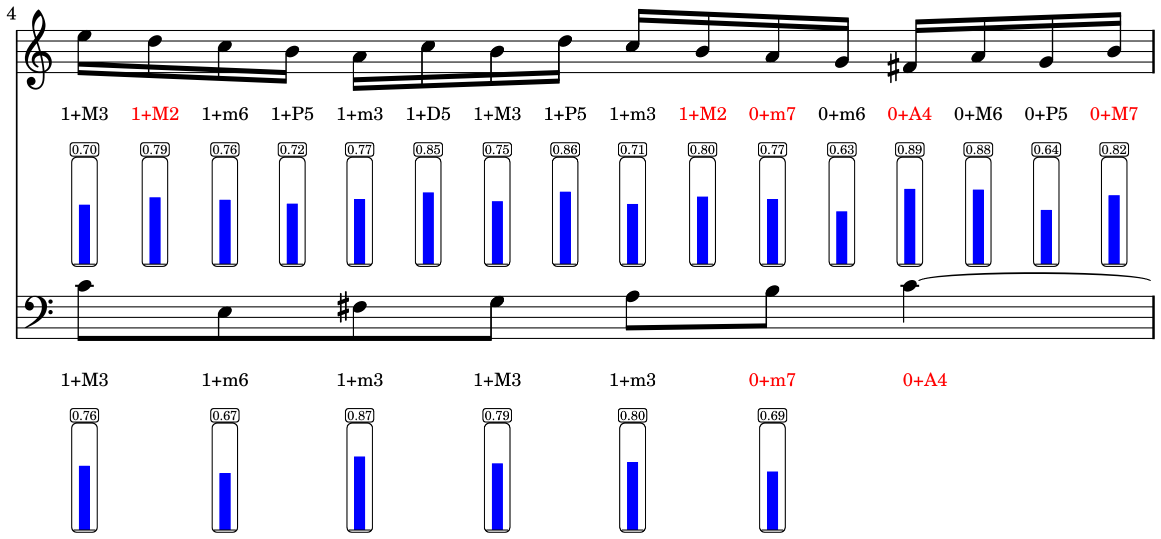
B.3 Combining Musical Queries and Performance Data

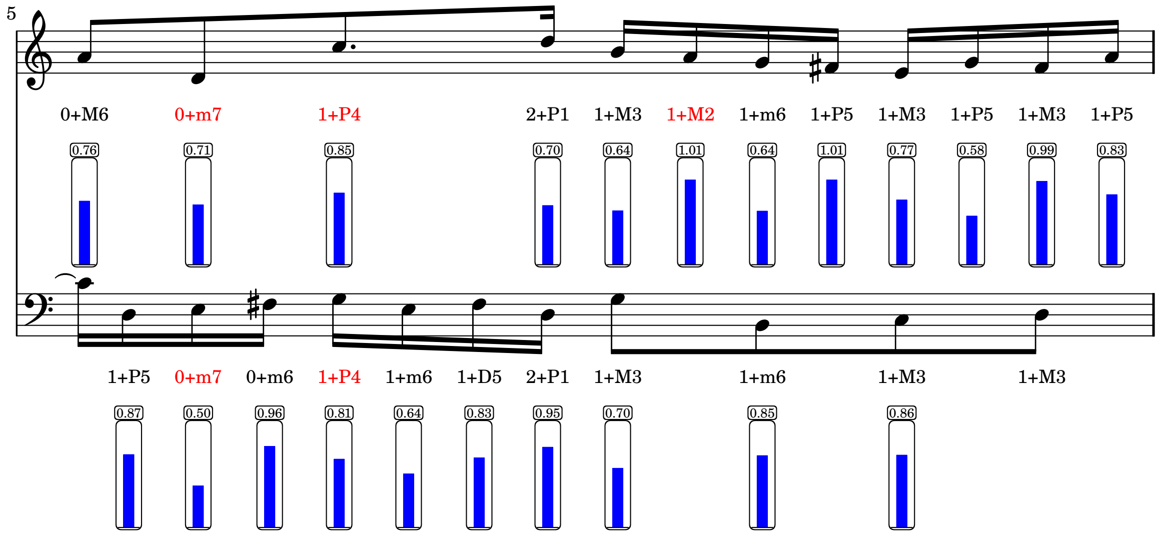
A presentation of a performance of Bach's Two Part Invention No.1 by a student at Glasgow University. The score is annotated with bar graphs showing the normalised inter-onset interval. Musical intervals are marked, with those intervals considered dissonant indicated in red. Whereas previously the IOI has been displayed as an absolute value, here we show the IOI as a deviation from a constant tempo. If the entire performance had been played at a constant tempo, the normalised IOI would give a constant value. Since we are analysing a real performance, with fluctuations in tempo, the normalised IOIs change. A semiquaver with a longer line line than a quaver here means that the semiquaver's normalised IOI was longer than the quaver's — the semiquaver's IOI was longer relative to the expected semiquaver IOI for a constant tempo than the quaver's expected IOI.

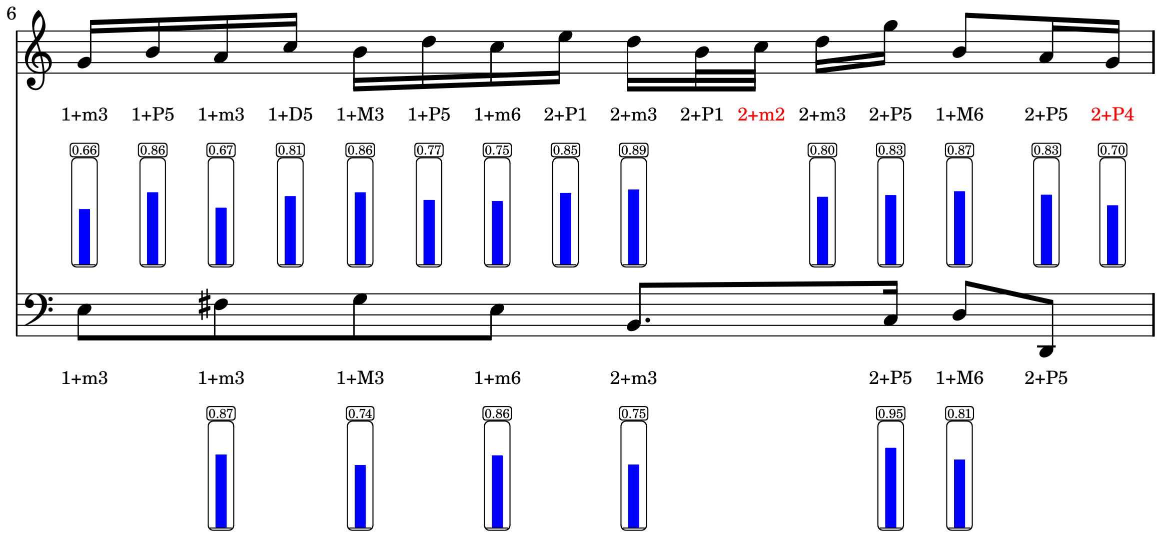

1+m7
1+A4
1+m6
0.76
0.80
0.72
0.71
0.85
0.75
0.88
0.79
1.15
1+P5
1+m7
1+m6
1+A4
1+M6
1+m6
2+P1
0.88
0.86
1.59
0.89
0.84

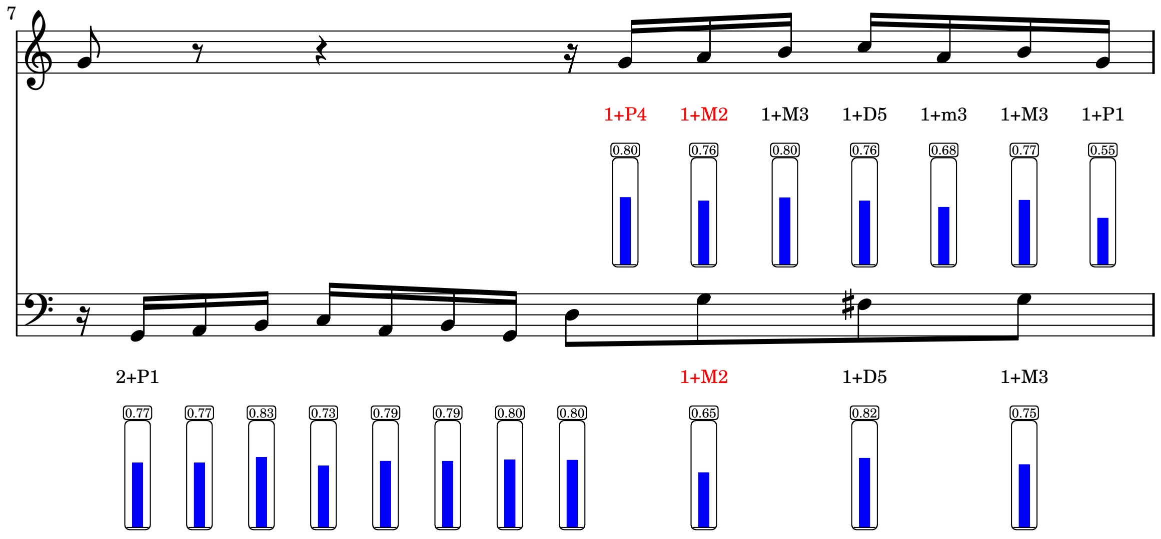

2
1+P5
1+P1
2+M2
2+M3
1+m7
1+P4
1+m6
0.70
0.76
0.85
0.85
0.63
0.87
0.78
0.81
0.82
0.85
0.74
0.81
1+P5
2+M2
1+P5
1+m7
1+m6
1+P4
1+m6
1+m6
2+P1
0.88
0.73
0.80
0.92
0.80
0.95
0.62
0.90

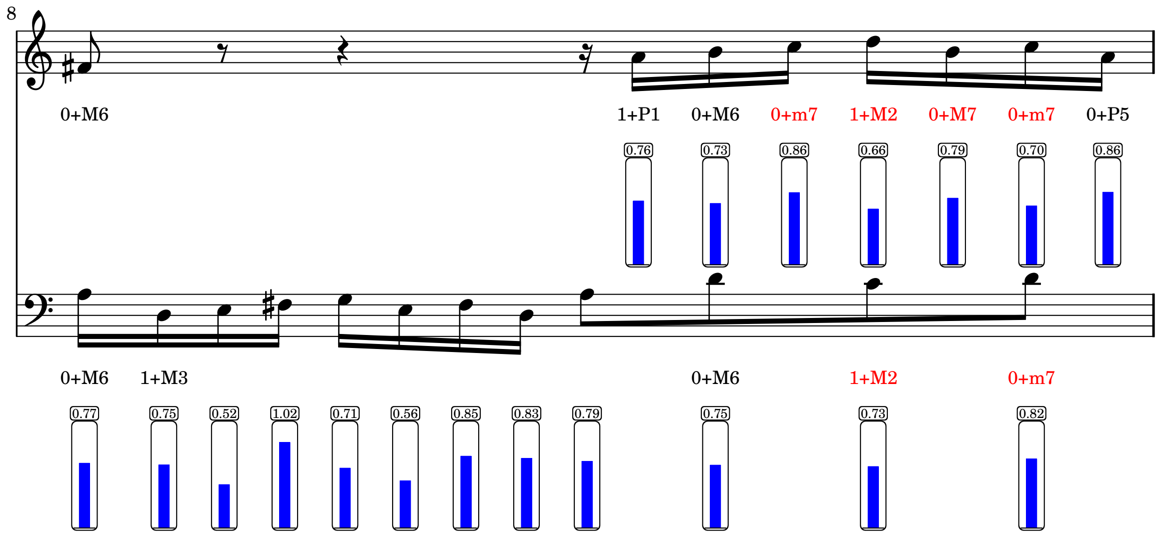

3
1+M3 1+M6 1+m6 1+D5 1+M3 1+P5 1+m3 1+P5 1+m3 1+m2 1+M6 1+P5 1+m3 1+P5 1+m3 1+D5
0.82 0.74 0.79 0.72 0.84 0.83 0.75 0.82 0.74 0.81 0.81 0.70 0.76 0.85 0.72 0.84
1+M3 1+m6 1+M3 1+m3 1+m3 1+M6 1+m3 1+m3
0.79 0.76 0.80 0.78 0.81 0.80 0.69 0.80

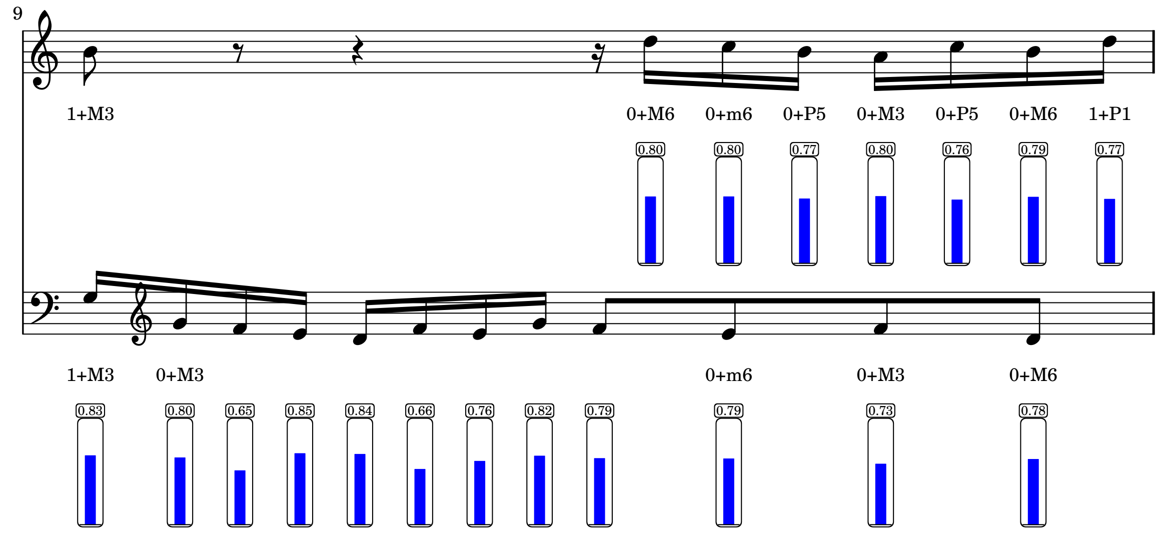

4
1+M3 1+M2 1+m6 1+P5 1+m3 1+D5 1+M3 1+P5 1+m3 1+M2 0+m7 0+m6 0+A4 0+M6 0+P5 0+M7
0.70 0.79 0.76 0.72 0.77 0.85 0.75 0.86 0.71 0.80 0.77 0.63 0.89 0.88 0.64 0.82
1+M3 1+m6 1+m3 1+M3 1+m3 0+m7 0+A4
0.76 0.67 0.87 0.79 0.80 0.69

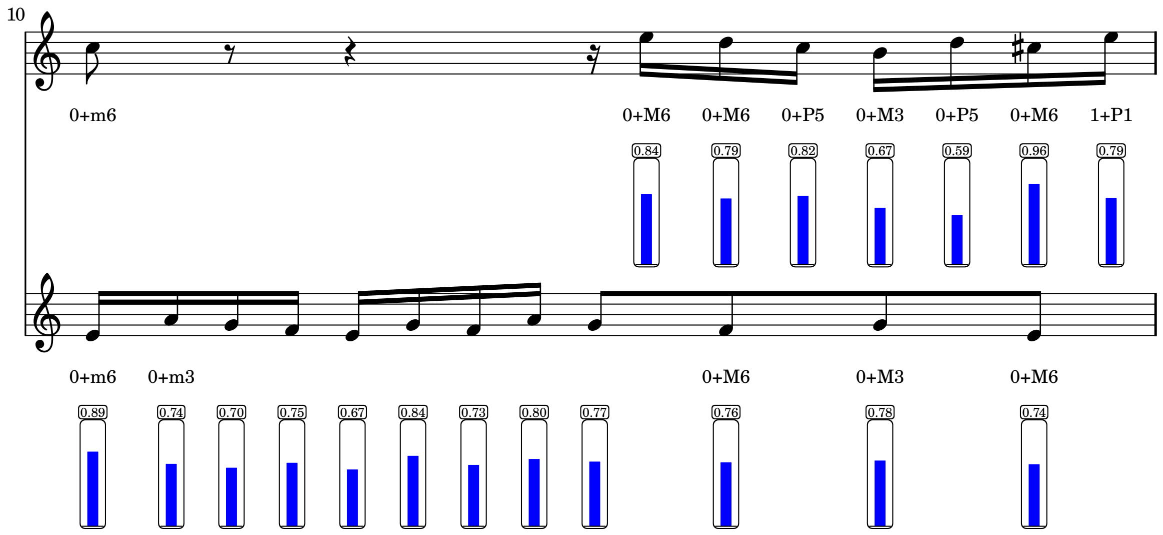

5
0+M6
0+m7
1+P4
2+P1
1+M3
1+M2
1+m6
1+P5
1+M3
1+P5
1+M3
1+P5
0.76
0.71
0.85
0.70
0.64
1.01
0.64
1.01
0.77
0.58
0.99
0.83
1+P5
0+m7
0+m6
1+P4
1+m6
1+D5
2+P1
1+M3
1+m6
1+M3
1+M3
0.87
0.50
0.96
0.81
0.64
0.83
0.95
0.70
0.85
0.86

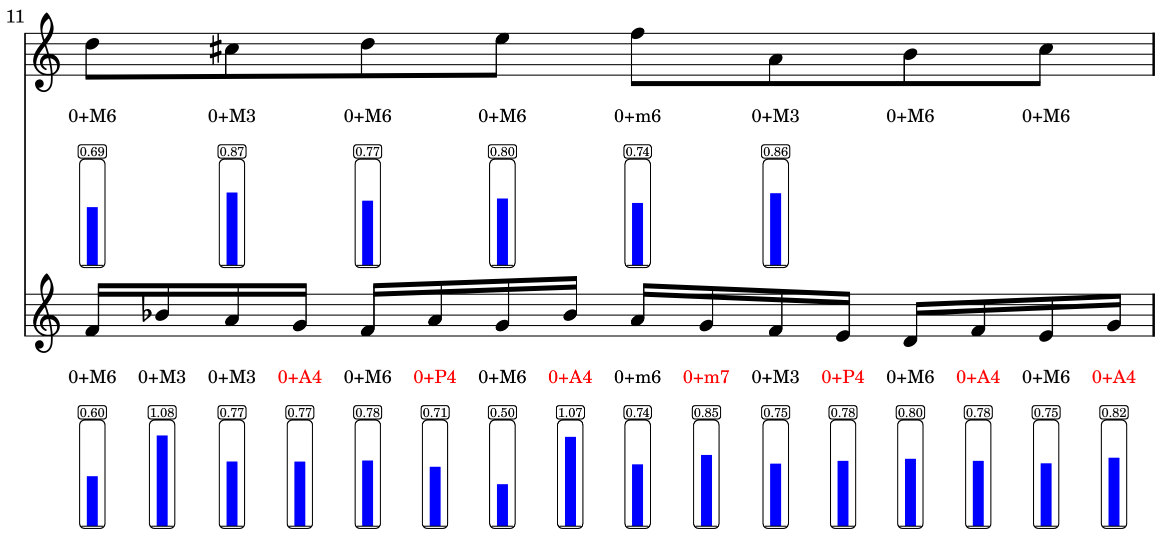

6
1+m3
1+P5
1+m3
1+D5
1+M3
1+P5
1+m6
2+P1
2+m3
2+P1
2+m2
2+m3
2+P5
1+M6
2+P5
2+P4
0.66
0.86
0.67
0.81
0.86
0.77
0.75
0.85
0.89
0.80
0.83
0.87
0.83
0.70
1+m3
1+m3
1+M3
1+m6
2+m3
2+P5
1+M6
2+P5
0.87
0.74
0.86
0.75
0.95
0.81

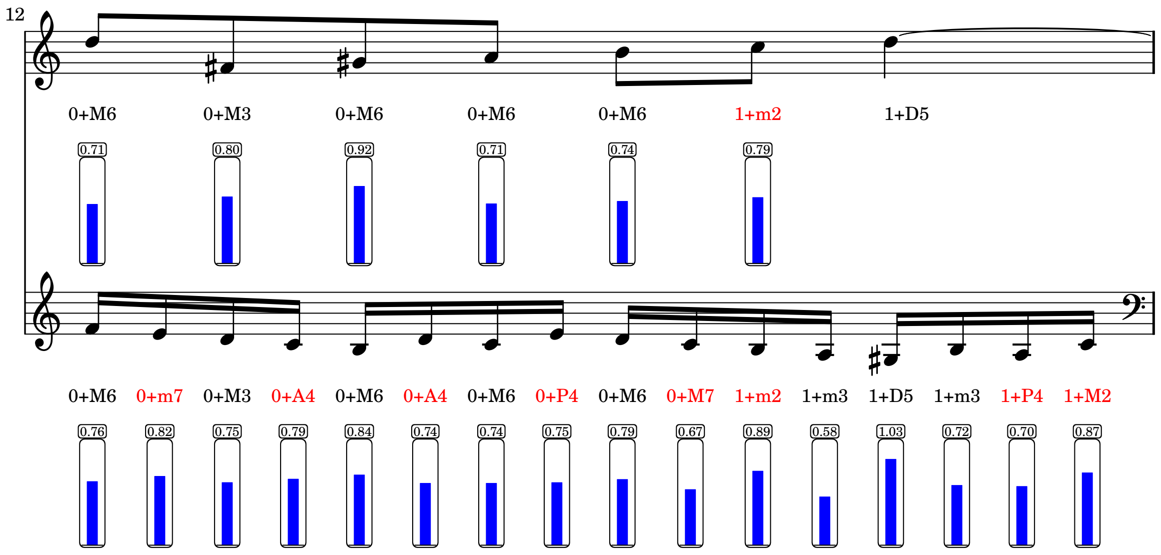

7
1+P4
1+M2
1+M3
1+D5
1+m3
1+M3
1+P1
0.80
0.76
0.80
0.76
0.68
0.77
0.55
2+P1
1+M2
1+D5
1+M3
0.77
0.77
0.83
0.73
0.79
0.79
0.80
0.80
0.65
0.82
0.75

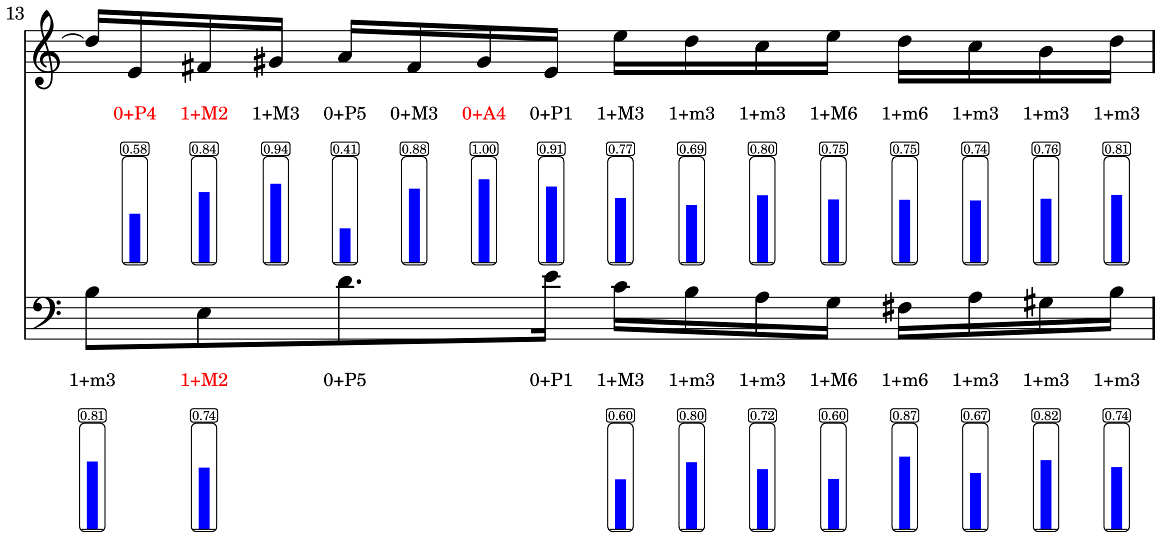

8
0+M6
1+P1
0+M6
0+m7
1+M2
0+M7
0+m7
0+P5
0.76
0.73
0.86
0.66
0.79
0.70
0.86
0+M6
1+M3
0+M6
1+M2
0+m7
0.77
0.75
0.52
1.02
0.71
0.56
0.85
0.83
0.79
0.75
0.73
0.82

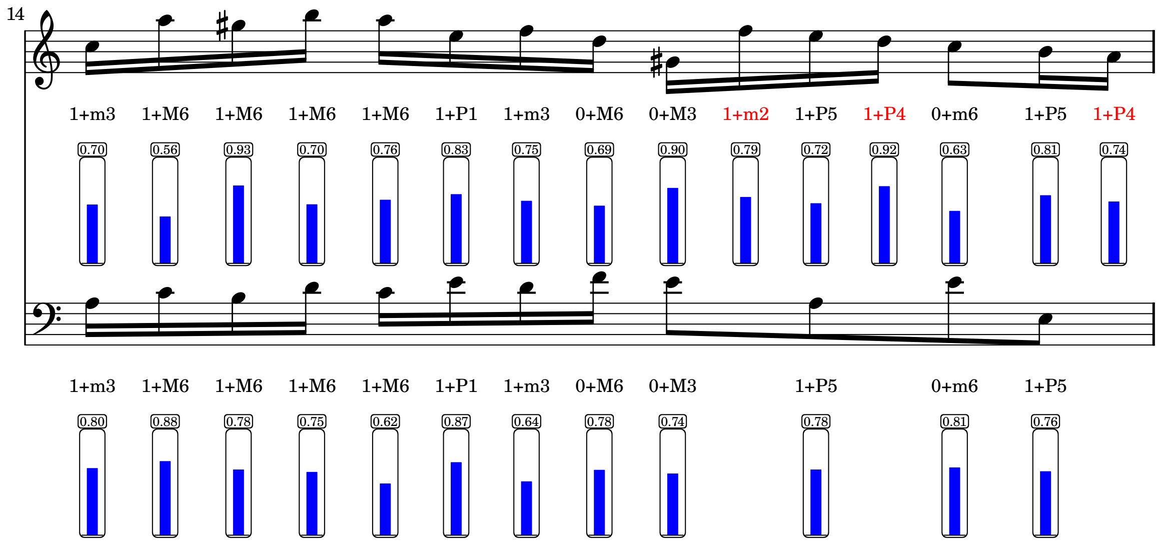

9
1+M3
0+M6
0+m6
0+P5
0+M3
0+P5
0+M6
1+P1
0.80
0.80
0.77
0.80
0.76
0.79
0.77
1+M3
0+M3
0+m6
0+M3
0+M6
0.83
0.80
0.65
0.85
0.84
0.66
0.76
0.82
0.79
0.79
0.73
0.78

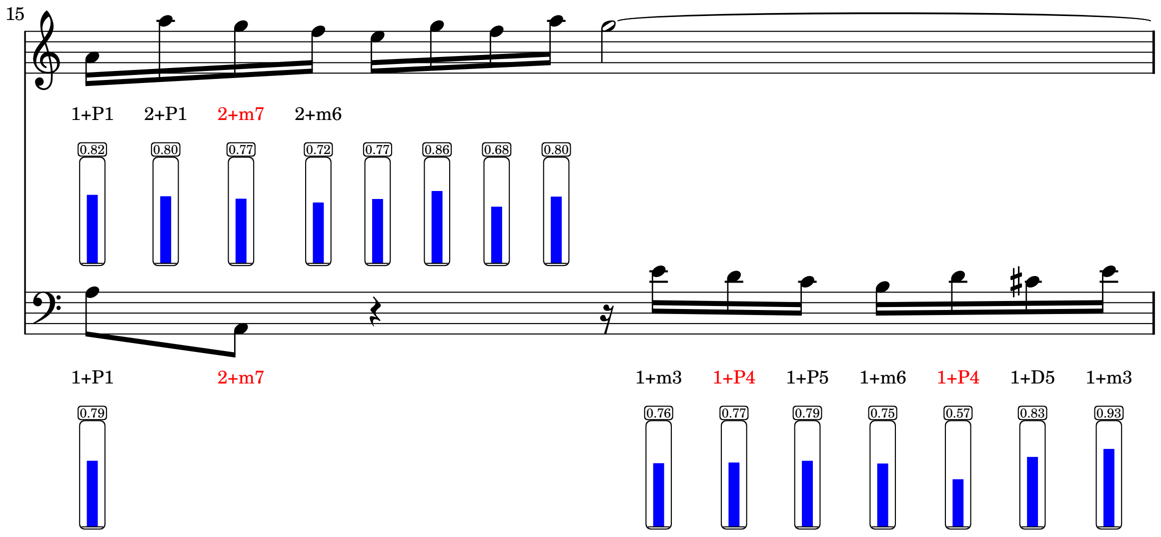

10
0+m6
0+M6
0+M6
0+P5
0+M3
0+P5
0+M6
1+P1
0.84
0.79
0.82
0.67
0.59
0.96
0.79
0+m6
0+m3
0+M6
0+M3
0+M6
0.89
0.74
0.70
0.75
0.67
0.84
0.73
0.80
0.77
0.76
0.78
0.74

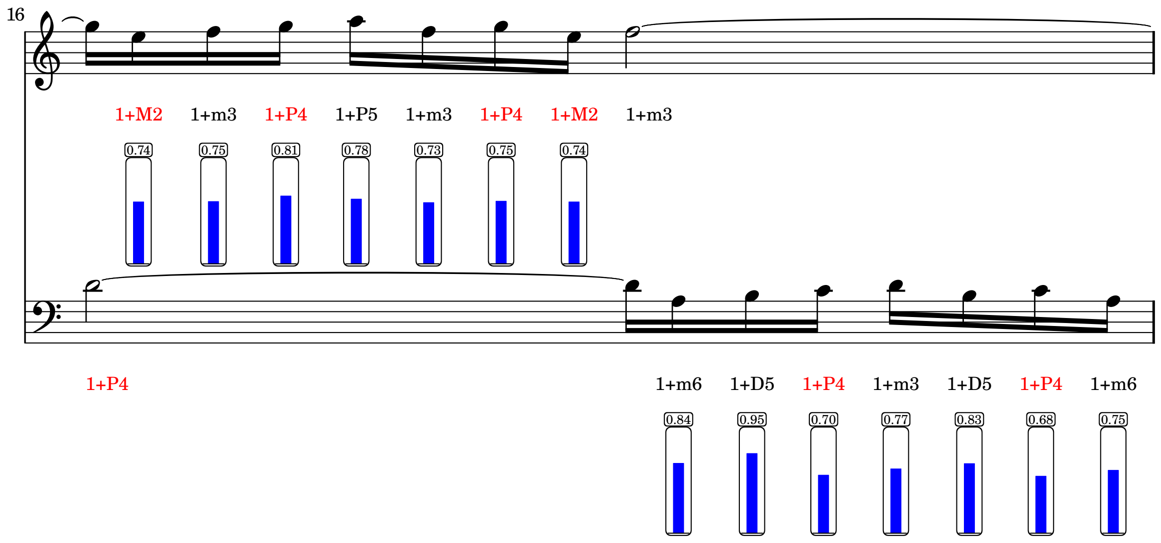

11
0+M6 0+M3 0+M6 0+M6 0+m6 0+M3 0+M6 0+M6
0.69 0.87 0.77 0.80 0.74 0.86
0+M6 0+M3 0+M3 0+A4 0+M6 0+P4 0+M6 0+A4 0+m6 0+m7 0+M3 0+P4 0+M6 0+A4 0+M6 0+A4
0.60 1.08 0.77 0.77 0.78 0.71 0.50 1.07 0.74 0.85 0.75 0.78 0.80 0.78 0.75 0.82

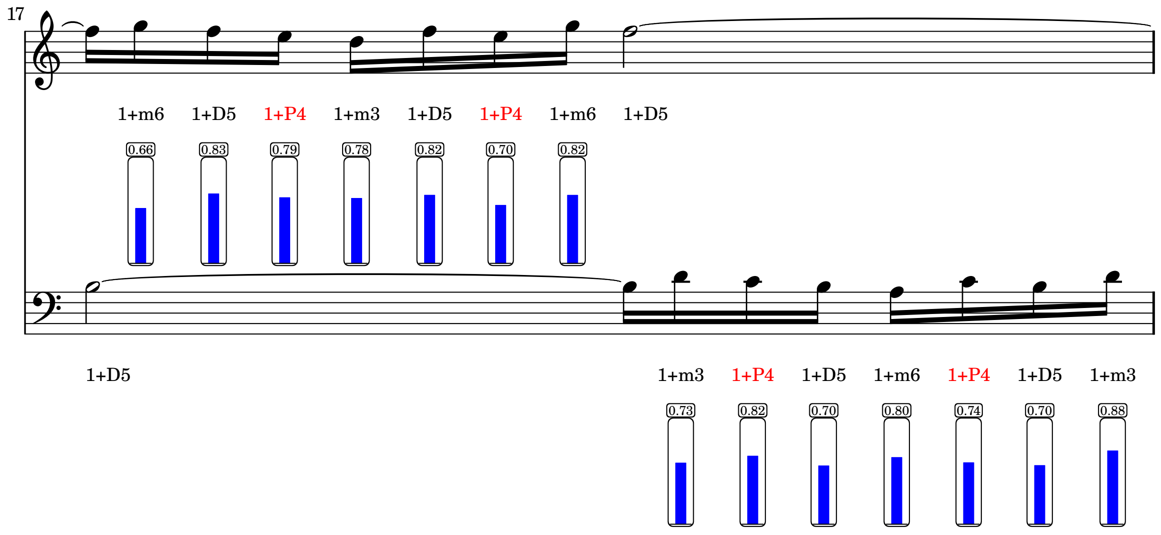

12
0+M6 0+M3 0+M6 0+M6 0+M6 1+m2 1+D5
0.71 0.80 0.92 0.71 0.74 0.79
0+M6 0+m7 0+M3 0+A4 0+M6 0+A4 0+M6 0+P4 0+M6 0+M7 1+m2 1+m3 1+D5 1+m3 1+P4 1+M2
0.76 0.82 0.75 0.79 0.84 0.74 0.74 0.75 0.79 0.67 0.89 0.58 1.03 0.72 0.70 0.87

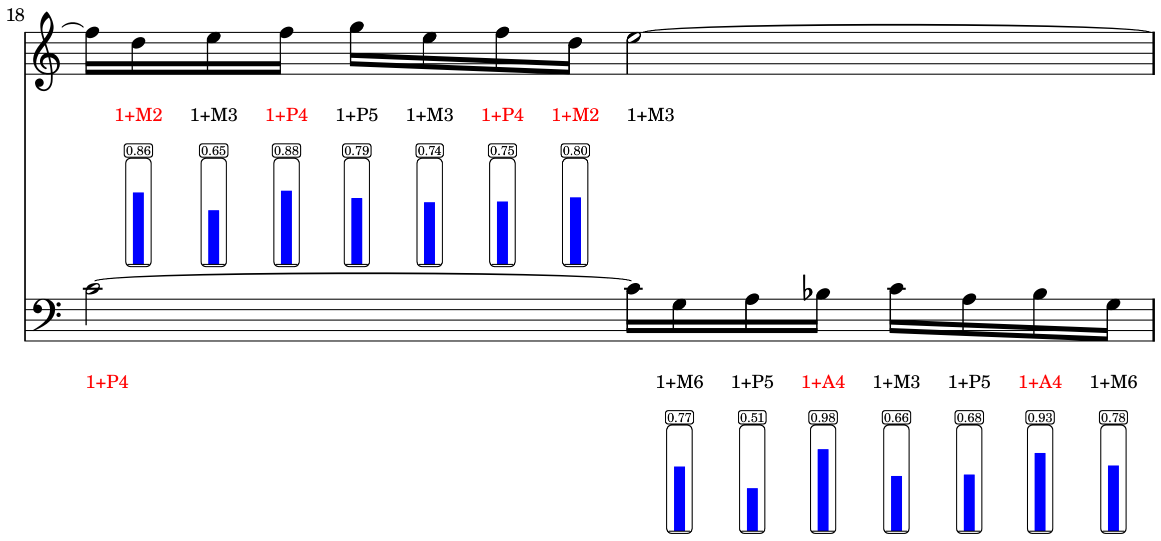

13
0+P4 1+M2 1+M3 0+P5 0+M3 0+A4 0+P1 1+M3 1+m3 1+m3 1+M6 1+m6 1+m3 1+m3 1+m3
0.58 0.84 0.94 0.41 0.88 1.00 0.91 0.77 0.69 0.80 0.75 0.75 0.74 0.76 0.81
1+m3 1+M2 0+P5 0+P1 1+M3 1+m3 1+m3 1+M6 1+m6 1+m3 1+m3 1+m3
0.81 0.74 0.60 0.80 0.72 0.60 0.87 0.67 0.82 0.74

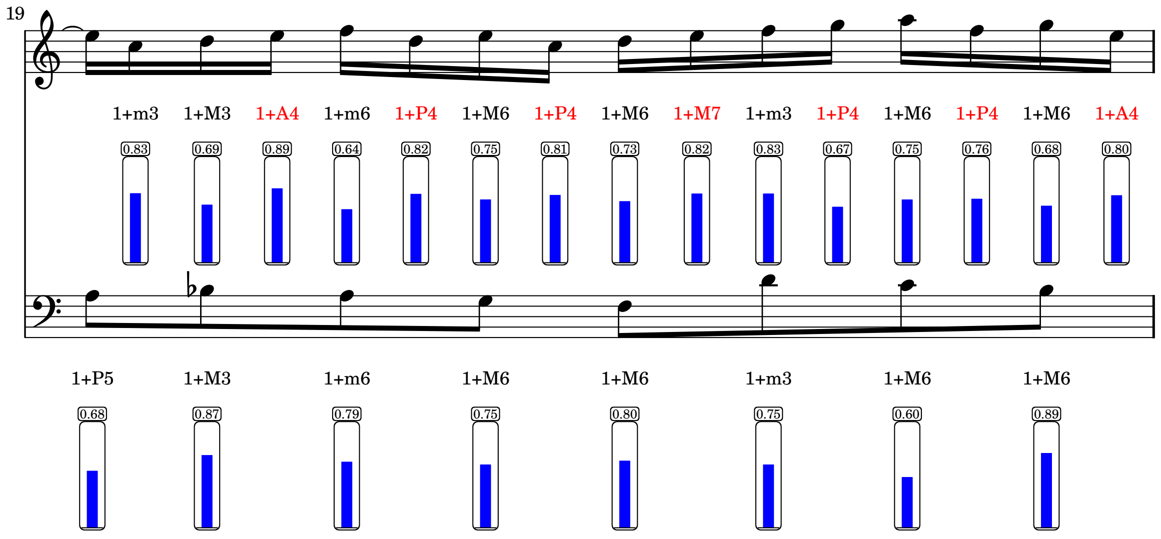

14
1+m3 1+M6 1+M6 1+M6 1+M6 1+P1 1+m3 0+M6 0+M3 1+m2 1+P5 1+P4 0+m6 1+P5 1+P4
0.70 0.56 0.93 0.70 0.76 0.83 0.75 0.69 0.90 0.79 0.72 0.92 0.63 0.81 0.74
1+m3 1+M6 1+M6 1+M6 1+M6 1+P1 1+m3 0+M6 0+M3 1+P5 0+m6 1+P5
0.80 0.88 0.78 0.75 0.62 0.87 0.64 0.78 0.74 0.78 0.81 0.76

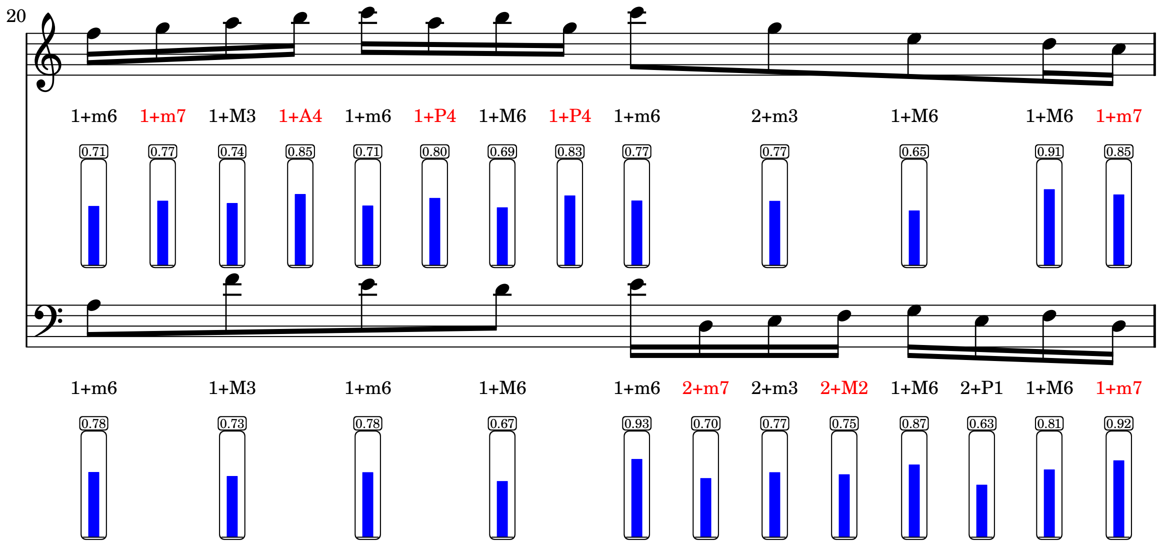

15
1+P1
2+P1
2+m7
2+m6
0.82
0.80
0.77
0.72
0.77
0.86
0.68
0.80
1+P1
2+m7
1+m3
1+P4
1+P5
1+m6
1+P4
1+D5
1+m3
0.79
0.76
0.77
0.79
0.75
0.57
0.83
0.93

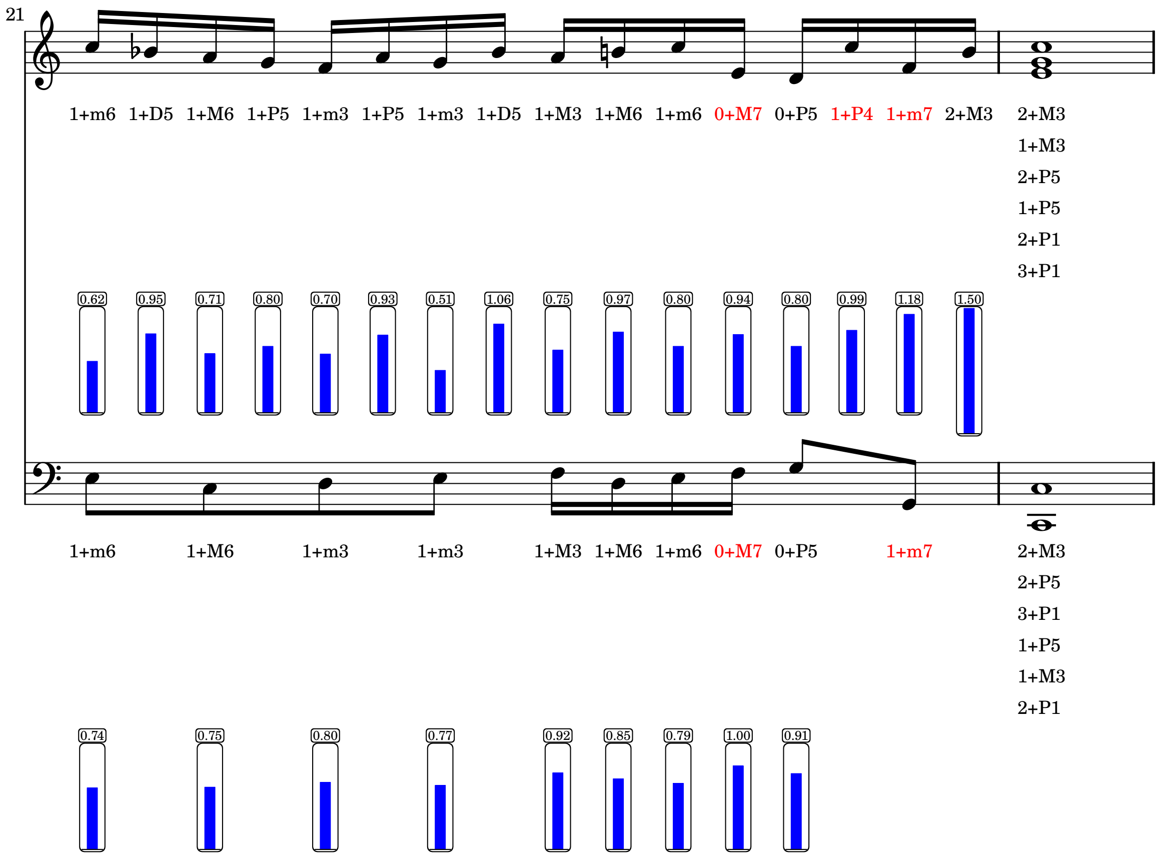

16
1+M2
1+m3
1+P4
1+P5
1+m3
1+P4
1+M2
1+m3
0.74
0.75
0.81
0.78
0.73
0.75
0.74
1+P4
1+m6
1+D5
1+P4
1+m3
1+D5
1+P4
1+m6
0.84
0.95
0.70
0.77
0.83
0.68
0.75


17
1+m6
1+D5
1+P4
1+m3
1+D5
1+P4
1+m6
1+D5
0.66
0.83
0.79
0.78
0.82
0.70
0.82
1+D5
1+m3
1+P4
1+D5
1+m6
1+P4
1+D5
1+m3
0.73
0.82
0.70
0.80
0.74
0.70
0.88


18
1+M2
1+M3
1+P4
1+P5
1+M3
1+P4
1+M2
1+M3
0.86
0.65
0.88
0.79
0.74
0.75
0.80
1+P4
1+M6
1+P5
1+A4
1+M3
1+P5
1+A4
1+M6
0.77
0.51
0.98
0.66
0.68
0.93
0.78


19
1+m3 1+M3 1+A4 1+m6 1+P4 1+M6 1+P4 1+M6 1+M7 1+m3 1+P4 1+M6 1+P4 1+M6 1+A4
0.83 0.69 0.89 0.64 0.82 0.75 0.81 0.73 0.82 0.83 0.67 0.75 0.76 0.68 0.80
1+P5 1+M3 1+m6 1+M6 1+M6 1+m3 1+M6 1+M6
0.68 0.87 0.79 0.75 0.80 0.75 0.60 0.89


20
1+m6 1+m7 1+M3 1+A4 1+m6 1+P4 1+M6 1+P4 1+m6 2+m3 1+M6 1+M6 1+m7
0.71 0.77 0.74 0.85 0.71 0.80 0.69 0.83 0.77 0.77 0.65 0.91 0.85
1+m6 1+M3 1+m6 1+M6 1+m6 2+m7 2+m3 2+M2 1+M6 2+P1 1+M6 1+m7
0.78 0.73 0.78 0.67 0.93 0.70 0.77 0.75 0.87 0.63 0.81 0.92



Music engraving by LilyPond 2.12.1—www.lilypond.org

Bibliography

[1] abc music notation. http://www.walshaw.plus.com/abc/.

[2] Ardour - the new digital audio workstation. http://www.ardour.org/.

[3] Chuck : Strongly-timed, concurrent, and on-the-fly audio programming language. http://chuck.cs.princeton.edu/.

[4] Clam is a full-fledged software framework for research and application development in the audio and music domain. http://clam-project.org/.

[5] Cubase 5 – advanced music production system. http://www.steinberg.net/en/products/musicproduction/cubase5_product.html.

[6] Everything you need to know about the humdrum "**kern" representation. http://www.music-cog.ohio-state.edu/Humdrum/representations/kern.html.

[7] Extensible markup language (xml). http://www.w3.org/XML/.

[8] Finale - music notation software. http://www.finalemusic.com/.

[9] The free, cross-platform sound editor. http://audacity.sourceforge.net/.

[10] Free music composition & notation software. http://musescore.org/en.

[11] Gnu general public license. http://www.gnu.org/licenses/licenses.html.

[12] Gnu octave is a high-level language, primarily intended for numerical computations. http://www.gnu.org/software/octave/.

[13] Gnuplot is a portable command-line driven graphing utility. http://www.gnuplot.info/.

[14] Interactive visual programming environment for music, audio, and media. http://cycling74.com/products/maxmspjitter/.

[15] Lilypond documentation. http://lilypond.org/web/documentation.

[16] Lilypond typesetter software. http://lilypond.org.

[17] Logic studio is a complete set of professional applications that lets you write, record, edit, mix, and perform. http://www.apple.com/logicstudio/.

[18] Mathematics software. http://www.mathworks.com/products/matlab/.

[19] Muse is a midi/audio sequencer with recording and editing capabilities. http://muse-sequencer.org/index.php/Main_Page.

[20] Musicxml 1.1 dtd. http://www.recordare.com/dtds/1.1/index.html.

[21] Musicxml 2.0 dtd. http://www.recordare.com/dtds/index.html.

[22] Musicxml™ document type definition public license version 2.0. http://www.recordare.com/dtds/license.html.

[23] Noteedit - a score editor. http://noteedit.berlios.de/.

[24] Open score format is an open and non-proprietary distribution, interchange and archive file format for digital scores (sheet music). http://openscoreformat.sourceforge.net/.

[25] Performance markup language. http://www.n-ism.org/Projects/pml.php.

[26] Pro tools is the most popular sound creation and production system in the world. http://www.digidesign.com/protools/.

[27] Python speed wiki page. http://wiki.python.org/moin/PythonSpeed.

[28] Recording, mixing, editing, and mastering — adobe audition. http://www.adobe.com/products/audition/.

[29] Rosegarden: music software for linux. http://www.rosegardenmusic.com/.

[30] Sibelius - software for writing, playing, printing and publishing music notation. http://www.sibelius.com/.

[31] Sonic visualiser is an application for viewing and analysing the contents of music audio files. http://www.sonicvisualiser.org/.

[32] Supercollider is an environment and programming language for real time audio synthesis and algorithmic composition. `http://supercollider.sourceforge.net/`.

[33] The tools we use. `http://www.music-ir.org/evaluation/tools.html`.

[34] *Cambridge Advanced Learner's Dictionary*. Cambridge University Press, 2007.

[35] M. Ricchetti A. Camurri, S. Hashimoto. EyesWeb - Toward Gesture and Affect Recognition in Interactive Dance and Music Systems. *Computer Music Journal*, 24(1):57–69, 2000.

[36] R. Trocca A. Camurri, M. Ricchetti. EyesWeb - toward gesture and affect recognition in dance/music interactive systems. *Proceedings of the IEEE Multimedia Systems '99, Firenze, Italy*, June 1999.

[37] A. X. Rodet and P. Cointe. Formes: composition and scheduling of processes. *Computer Music Journal*, 8(3):32–50, 1984.

[38] Perry R. Cook Ajay Kapur, Philip Davidson. Digitizing North Indian Performance. In *Proceedings of the International Computer Music Conference*, 2004.

[39] Alan Marsden. MTT - A Music Theory Tool. In *Proceedings of the Journées d'Informatique Musicale (JIM '97)*, 6 June 1997.

[40] Alan Marsden. *Representing Musical Time: A Temporal-Logic Approach*. Swets & Zeitlinger, 2000.

[41] Alexander R. Brinkman. Representing Musical Scores for Computer Analysis. *Journal of Music Theory*, 30(2):225–275, 1986.

[42] Stefan Kersten Alfonso Perez Carillo, Esteban Maestre. Expressive Irish Fiddle Performance Model Informed With Bowing. In *Proceedings ICMC 2008*, 2008.

[43] B. M. Eaglestone. Composition tools integration with a music database system. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, number 978 in Lecture Notes in Computer Science, pages 459–468. Springer-Verlag, 1995.

[44] B. M. Eaglestone, G. L. Davies, M. Ridley, and N. Hulley. Implementation of an artists version model using extended relational database technology. In *Advances in Databases. 11th British National Conference on Databases (BNCOD-11)*, number 696 in Lecture Notes in Computer Science, pages 258–276, Keele, July 1993. Springer Verlag.

[45] Niall Moody; Dr. Nick Fells; Dr. Nicholas Bailey. Ashitaka: an audiovisual instrument. In *Proceedings of the New Interfaces for Musical Expression Conference*, 2007.

[46] Pierfrancesco Bellini and Paolo Nesi. Wedelmusic format: an xml music notation format for emerging applications. In *Proceedings of the First International Conference on WEB Delivering of Music*, 2001.

[47] Easley Blackwood. *Twelve Microtonal Etudes for Electronic Music Media, Op. 28*. G. Schirmer, Inc., 1982.

[48] Easley Blackwood. *The Structure of Recognizable Diatonic Tunings*. Princeton University Press, 1985.

[49] Jered Bolton. *Gestural Extraction from Musical Audio Signals*. PhD thesis, Department of Electronics and Electrical Engineering University of Glasgow, 2004.

[50] Richard Boulanger, editor. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT Press, Cambridge, Mass., 2000.

[51] Friberg Bresin, R. Director musices: The KTH performance rules system. In *Proceedings of SIGMUS-46*, pages 43–48, 2002.

[52] Bresin R.; Umberto Battel G. Articulation Strategies in Expressive Piano Performance: Analysis of Legato, Staccato, and Repeated Notes in Performances of the Andante Movement of Mozart's Sonata in G Major (K 545). *Journal of New Music Research*, 29(3):211–224, 2000.

[53] Alexander R. Brinkman. A binomial representation of pitch for computer processing of musical data. *Music Theory Spectrum*, 8:44–57, 1986.

[54] Carola Boehm. *Methodologies for the Design and Development of System and Data Architectures for Music Information*. PhD thesis, University of Glasgow, 2005.

[55] David Meredith. The ps13 pitch spelling algorithm. *Journal of New Music Research*, 35(2):121–159, 2006.

[56] F. Delalande. *La gestique de Gould*, page 85–111. Quebec, 1988.

[57] Diana Young. The Hyperbow: A Precision Violin Interface. In *Proceedings of the International Computer Music Conference (ICMC2002)*, 2002.

[58] Douglas McGilvray. *On the Analysis of Music by Computer*. PhD thesis, University of Glasgow, September 2007.

[59] M. Dovey. Overview of the omras project: Online music retrieval and searching. *Journal of the American Society for Information Science and Technology*, 2002.

[60] E. W. Large. Dynamic programming for the analysis of serial behaviors. *Behavior Research Methods, Instruments, & Computers*, 25(2):238–241, 1993.

[61] B.M. Eaglestone, B. Desai, R. Holton, and E. Gulatee. Temporal database support for cooperative creative work. In *Proceedings of the 2nd International Database Engineering and Applications Symposium (IDEAS'98)*, pages 266–275, Cardiff, July 98.

[62] F. Deliège and T. Pederson. Music warehouses: Challenges for the next generation of music search engines. In *Proceedings of the LSAS*, 2006.

[63] F. Lerdahl and R. Jackendoff. *A Generative Theory of Tonal Music*. MIT Press, Cambridge, Mass., 1983.

[64] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 2nd edition, 2002.

[65] Gareth Loy. Musicians Make a Standard: The Midi Phenomenon. *Computer Music Journal*, 9(4):8–26, 1985.

[66] Alexander R. Jensenius; Tellef Kvifte; Rolf Inge Godoy. Towards a gesture description interchange format. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, page 176–179, 2006.

[67] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[68] Michael Good. *The Virtual Score*, volume 12 of *Computing in Musicology*, chapter 8, page 113–124. The MIT Press, 2001.

[69] Cindy Grande. The notation interchange file format. In Eleanor Selfridge-Field, editor, *Beyond MIDI*, pages 491–512, Cambridge, Mass., 1997. The MIT Press.

[70] Smaill Harris, M. Representing music symbolically. In C. Canepa A. Camurri, editor, *IX colloquio di Informatica Musicale*, Genova, 1991. Universita di Genova.

[71] Walter B. Hewlett. A base-40 number-line representation of musical pitch notation. *Musikometrika*, 4:1–14, 1992.

[72] Henkjan Honing. Poco: an environment for analysing, modifying, and generating expression in music. In *Proceedings of the 1990 International Computer Music Conference*, 1990.

[73] Henkjan Honing. Issues in the representation of time and structure in music. *Contemporary Music Review*, 9:221–239, 1993.

[74] David Huron. Music information processing using the humdrum toolkit: Concepts, examples and lessons. *Computer Music Journal*, 26(2):11–26, 2000.

[75] J. Lane and W. Punch. A relational database approach to polyphonic music search systems using regular expressions. Technical report, Department of Computer Science and Engineering, Michigan State University, Michigan, USA, 2002.

[76] Jamie Forth. Personal communication with the author.

[77] Bryony Buck Jennifer MacRitchie and Nicholas J Bailey. Visualising Musical Structure through Performance Gesture. In *Proceedings of the 10th International Society for Music Information Retrieval Conference*, September 2009.

[78] Nicholas J. Bailey Jennifer MacRitchie, Stuart Pullinger and Graham Hair. Communicating Phrasing Structure with Multi-Modal Expressive Techniques in Piano Performance. In *The Second International Conference on Music Communication Science*, Sydney, Australia, 3 December 2009.

[79] Jesus L. Alvaro, Eduardo R. Miranda, and Beatriz Barros. EV: Multilevel Music Knowledge Representation and Programming. In *Proceedings of the 10th Brazilian Symposium of Musical Computation (SBCM)*, Belo Horizonte (Brazil), 2005.

[80] B.Buck J.MacRitchie and Nicholas Bailey. Gestural Communication: Linking multi-modal Analysis of Performance to Perception of Musical Structure. In *Proceedings of the International Symposium of Performance Science*, Auckland, New Zealand, December 2009.

[81] Joachim Ganseman, Paul Scheunders, and Wim D'haes. Using XQuery on Musicxml Databases for Musicological Analysis. In *Proceedings of ISMIR 2008*, 2008.

[82] Joshua J. Bloch and Roger B. Dannenberg. Real-time computer accompaniment of keyboard performances. *Proceedings of the ICMC*, pages 279–289, 1985.

[83] Karl Aberer and Wolfgang Klas. Supporting Temporal Multimedia Operations in Object-Oriented Database Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, USA, May 1994.

[84] Keiji Hirata and Tatsuya Aoyagi. Computational Music Representation Based on the Generative Theory of Tonal Music and the Deductive Object-Oriented Database. *Computer Music Journal*, 27(3):73–89, 2003.

[85] Tillman Weyde Kia Ng and Paolo Nesi. I-MAESTRO: Technology-Enhanced Learning for Music. In *Proceedings of the International Computer Music Conference (ICMC) 2008*, 2008.

[86] J. Timothy Kolosick. A machine-independent data-structure for the representation of musical pitch relationships: Computer-generated musical examples for cbi. *Journal of Computer-Based Instruction*, 13/i:9–13, 1986.

[87] Müller Kurth, F. SyncPlayer - An Advanced System for Multimodal Music Access. In *Proceedings of the 6th International Conference on Music Information Retrieval (ISMIR 2005)*, London, UK, 2005.

[88] Annie Luciani, Matthieu Evrard, Damien Courousse, Nicolas Castagne, Claude Cadoz, and Jean-Loup Florens. A basic gesture and motion format for virtual reality multisensory applications. In *International Conference on Computer Graphics Theory and Applications*, 2006.

[89] M. Balaban. *Understanding Music with AI: Perspectives on Music Cognition*, chapter Music structures: Interleaving the temporal and hierarchical aspects in music, pages 11–13. MIT-AAAI Press, 1992.

[90] M. V. Mathews and J. R. Pierce. *Current Directions in Computer Music Research*, chapter The Bohlen-Pierce Scale. MIT Press, 1989.

[91] Mathieu Bergeron and Darrell Conklin. Structured Polyphonic Patterns. In *Proceedings of the International Conference on Music Information Retrieval*, pages 69–74, 2008.

[92] David Meredith. The ps13 pitch spelling algorithm. *Journal of New Music Research*, 35:2:121–159, 2006.

[93] Mira Balaban. The Musical Structures Approach to Knowledge Representation for Music Processing. *Computer Music Journal*, 20(2):96–111, 1996.

[94] F. Richard Moore. The dysfunctions of midi. *Computer Music Journal*, 12(1):19–28, Spring 1988.

[95] Jerry Morrison. Ea iff 85 - standard for interchange format files, 1985.

[96] Will Mowat. Bob moog piano bar. *Sound On Sound*, March 2005.

[97] S. R. Newcomb. Standard music description language complies with hypermedia standard. *Computer*, 24(7):76–79, 1991.

[98] Douglas McGilvray Nicholas Bailey and Graham Hair. Musically Significant, Automatic Localisation of Note Boundaries for the Performance Analysis of Vocal Music. In *Proceedings of the Conference on Interdisciplinary Musicology*, Thessaloniki, 2008.

[99] Harry Partch. *Genesis of a Music*. University of Wisconsin Press, 1979.

[100] Pedro Kröger and Alexandre Passos. Rameau: A System for Automatic Harmonic Analysis. In *Proceedings of the International Computer Music Conference (ICMC) 2008*, 2008.

[101] Perry Roland. XML4MIR: Extensible Markup Language for Music Information Retrieval. In *Proceedings of the International Conference on Music Information Retrieval (ISMIR 00)*, 2000.

[102] Peter Chen. The Entity Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[103] Philip D. Morehead. *Bloomsbury Dictionary of Music*. Bloomsbury, 1992.

[104] Miller Puckette. Pure data. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1997.

[105] Richard Parncutt and Malcolm Troup. *Science and psychology of music performance: Creative strategies for teaching and learning*, chapter Piano, pages 285–302. Oxford University Press, New York, 2002.

[106] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, Inc., July 1999.

[107] Richard T. Snodgrass (chair), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[108] Roger B. Dannenberg. An on-line algorithm for real-time accompaniment. In *Proceedings of the ICMC*, 1984.

[109] Perry Roland. The music encoding initiative (mei). In *XML (MAX)*, 2002.

[110] S. T. Pope. The interim dynapiano: An integrated computer tool and instrument for composers. *Computer Music Journal*, 16(3):73–91, 1992.

[111] Heinrich Schenker. *Harmony*. University of Chicago Press, 1956.

[112] Carl Seashore. *Psychology of Music*. McGraw-Hill Book Company, 1938.

[113] Eleanor Selfridge-Field, editor. *Beyond MIDI*. Center for Computer Assisted Research in the Humanities. The MIT Press, 1997.

[114] Eleanor Selfridge-Field. *Beyond MIDI*, chapter DARMS, Its Dialects, and Its Uses. Center for Computer Assisted Research in the Humanities. The MIT Press, 1997.

[115] Donald Sloan. Hytime and standard music description language. In Eleanor Selfridge-Field, editor, *Beyond MIDI*. The MIT Press, 1997.

[116] Alan Smail, Geraint Wiggins, and M Harris. *Hierarchical music representation for composition and analysis*. Computers and the Humanities. Springer, 1993.

[117] Stephen Smoliar. SCHENKER: a computer aid for analysing tonal music. *ACM SIGLASH Newsletter*, 10(1–2):30–61, 1976.

[118] Jennifer MacRitchie Stuart Pullinger, Nicholas Bailey. Computer Assisted Analysis and Display of Musical and Performance Data. In *Proceedings of the International Symposium of Performance Science*, Auckland, New Zealand, December 2009.

[119] Nicholas Bailey Stuart Pullinger, Douglas McGilvray. Music and Gesture File: Performance Visualisation, Analysis, Storage and Exchange. *Proceedings of the International Computer Music Conference*, 2008.

[120] Takayuki Hoshishiba and Susumu Horioguchi. Improved DP matching between a musical score and its performance using interpolation. *Acoustical Science and Technology*, 22(1):13–19, 2001.

[121] W. Bradley Rubenstein. A database design for musical information. In *Proceedings of the ACM SIGMOD*, pages 479–490, 1987.

[122] Marcelo M. Wanderley, Bradley W. Vines, Neil Middleton, Cory McKay, and Wesley Hatch. The musical significance of clarinetists' ancillary gestures: An exploration of the field. *Journal of New Music Research*, 34(1):97–113, 2005.

[123] Harris Wiggins, G. A. Representing music for analysis and composition. In O. Laske M. Balaban, K. Ebcioglu, editor, *Proceedings of the Second IJCAI Workshop on Artificial Intelligence and Music*, pages 63–71, Detroit, US-MI, 1989.

[124] R. W. Young. Terminology for Logarithmic Frequency Units. *The Journal of the Acoustical Society of America*, 11(1):134–139, July 1939. Source of the number 4 for the octave starting on Middle-C.