Gestural Extraction from Musical Audio Signals

Jered Bolton

30th September 2004

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

Department of Electronics and Electrical Engineering

University of Glasgow

Abstract

Conventional exploration of gestures normally associated with musical instruments can be a costly and intrusive process. This thesis presents a novel approach to gestural extraction which overcomes these problems. The motivation behind this research is that the result of gestural input can be heard and therefore extracted from the acoustic signal produced by a musical instrument. Therefore, the guiding principles of this work are taken from the human auditory system.

The concept of temporal grouping, and the fact that any sound which reaches the inner ear is conveyed to the brain, are two features of the auditory system that are mimicked by the presented system. Pertinent definitions are proposed for the sections of the note envelope and musical instrument gestures are classified according to those responsible for excitation or control.

The extraction of gestural information is dependent upon successful identification of note events. A note tracking system is presented which exploits the structure of a note in order to perform preliminary note onset detection. A backtracking function is employed to regress through auditory data, providing a means of assigning individual start points to each note harmonic. The note tracking system also records the end point of each note harmonic. Note information is validated by a bespoke musical comparison system which provides a means of comparing and evaluating different note detection methods.

Information provided by the note tracking system is used to extract gestural information regarding oboe key presses and excitation (articulation) methods of string instruments. System tests show that it is possible to correctly distinguish between bowed and plucked notes with an 89% success rate, using only three discriminators associated with the onset of a note.

In this thesis the foundations of a multifacetted gestural extraction system are presented with useful potential for further development.

"And even things without life giving sound, whether pipe or harp, except they give a distinction in the sounds, how shall it be known what is piped or harped? For if the trumpet give an uncertain sound, who shall prepare himself to the battle?"

1 Corinthians 14:7,8

Acknowledgements

First and foremost I would like to thank Dr N.J. Bailey and Prof. J Arnold for their help and guidance. I am also greatly indebted to the following professional Musicians who gave freely of their time to record their instruments in an anechoic chamber: Miss Rosie Staniforth, Mr Donald Gillan and Miss Kate Dunnho.

Thanks also to Mr. Tom O'Hara the lab technician for help, assistance and encouragement.

I am also indebted to my younger sister Keri Bolton (an oboist) for providing valuable advice on the mechanics of the oboe.

Finally I would like to thank my wife Katie for all her help and support.

Contents

List of Figures

List of Tables

1 Introduction

Johan Sundberg begins his introduction to a special "Motion and Music" issue of the Journal of New Music Reasearch[88] with the statement:

"Music and motion are closely related."

Sound is produced as a result of a movement providing an input to a musical instrument. Whilst the movements required to play an instrument are quite basic, they take years of practice to master. Risset and Wessel[25, p135] comment that "*Acoustic instruments are controlled by carefully learned gestures*". Learning the wrong technique can ultimately hamper progress in musical proficiency. Researchers have therefore long sought facile methods that will allow them to study the gestures and techniques associated with an instrument and gain a detailed understanding of how movements or gestures influence the sound produced.

This thesis presents strong evidence that a note tracking system can be used to successfully and reproducibly identify specific gestures that control sound. Information about these gestures can be extracted from audio signals and could be utilised as a tool for assessment of musical technique.

The rationale for the design of this synthetic "listening" system is presented in chapter two, exploiting aspects of the human auditory system. The following chapter explores the link between music and motion. A taxonomy of gestures is presented and the gestures associated with the oboe and violin are identified.

In chapter four the nomenclature of the traditional sections of a note is adjusted for gestural analysis purposes and published note onset detection methods are reviewed.

The specifications of a tailor-made comparator which can validate the output from a note detection system are presented in chapter five. The performance of a published pitch tracker is rigorously tested for later comparison.

A novel note tracking system designed for the purpose of extracting information pertinent to gesture and the tests that validate its output are described in detail in chapters six and seven, respectively.

In chapter eight, excitation gestures common to string instruments are successfully extracted from note information and oboe key clicks, which can be detrimental to sound quality, are identified.

Finally, conclusions are drawn and further work on various aspects of the system is recommended.

2 The Human Auditory System

The human auditory system consists of the combination of the ear and brain. The ear serves as a means of gathering sound which it converts into information which the brain can process. The role of the brain is to interpret, and in some cases store, the information it receives.

Advances in the performance of personal computers have resulted in computers that can perform complex operations on large amounts of data. Desktop computers can now carry out signal processing tasks that were previously the sole domain of super computers. Whilst the speed of computer data processing has increased, the ability to automatically interpret data has not continued apace. Any attempt to make a computer interpret data is essentially an attempt at emulating a given human sense. In order to acheive this the gap between computer objectiveness and human subjectiveness must be bridged. Helpful analogies can be drawn between different senses, particularly hearing and vision. Phenomena such as the persistence of vision and optical illusion can help to describe comparable phenomena in the auditory domain.

This chapter considers different aspects of the human auditory system as a means to establish the foundations of a system which will enable a computer to "hear" music and interpret it.

2.1 The anatomy of the ear

The hearing system in humans shares common features with that of most mammals. There are usually three physically identifiable stages of the hearing system, namely: the outer ear, the middle ear and the inner ear. These three stages result in a remarkable device that can detect sound intensities ranging from a barely audible whisper to the noise of a jet taking off. The three linked stages of the ear are considered in the following sections.

2.1.1 The outer ear

Some mammals have a greater degree of control over the outer ear than others. In humans the outer ear faces forward such that the predominant field of hearing is broadly matched by the field of vision. A simple experiment, whereby cupped hands are placed over the ears so that sound is captured from behind the head, serves to highlight the relationship between what we see and what hear. Thus the brain has a twofold supply of information; a conclusion reached from one sensory source can be reinforced with information from another.

Shaw[85] uses data from twelve different studies to show the transformation of sound pressure to the ear drum, azimuthal dependence and interaural differences as functions of frequency. Algazi *et al.*[2] show that in humans, the shape of the outer ear (pinna) modifies high frequency components of incoming sound. They also show that the filtering effect of the pinna is dependent on sound source location and is therefore used as a localising aid.

2.1.2 The middle ear

It is suggested that the middle ear has two major functions, summarised by Moore[62, p23]:

1. The combination of the ossicles[1], the eardrum and the oval window of the cochlea results in an acoustic impedence matcher. This functionality is required as a result of the larger area of the eardrum relative to that of the cochlea's oval window. This in turn ensures that more sound is transmitted to the cochlea than if the oval window was directly exposed to sound waves.

2. It isolates the inner ear from internally transmitted sounds. If the inner ear was not isolated then certain actions (e.g. eating) would cause virtual deafness due to the masking of external sounds by internal sounds. Moore notes that birds and reptiles, whose inner ears are not isolated, swallow their food whole whereas mammals chew their food.

2.1.3 The inner ear

The inner ear is the main processing centre where the movements of the ossicles cause the basilar membrane to vibrate. The position of the vibration on the basilar membrane is determined by the frequency of the sound wave entering the ear. Vibrations of the basilar membrane cause hair cells, of which there are over 15,000, to be displaced resulting in neural activity. It is commonly agreed that the basilar membrane effectively acts as a frequency analyzer[70][16, p52][62, p66].

[1]These are the three bones between the eardrum and the cochlea: the malleus, incus and stapes, colloquially known as the hammer, anvil and stirrup respectively.

2.2 The sensitivity of the ear

The ear is a remarkably precise and sensitive device. Rasch and Plomp[25, p90] comment that it can respond to changes in sound pressure of less than 1dB and changes in frequency of less than 1Hz. The following aspects of the ear's sensitivity have been studied:

- **Pitch**. It is generally agreed that the ear is sensitive to frequencies within the range 20-20,000Hz, though with the onset of old age, the upper limit diminishes[35]. Studies have shown that due to the critical bandwidth effect, it is only possible to distinguish between the lower 5 to 8 harmonics of a complex tone[71]. Partials above the 6th to 8th harmonic usually fall within one bandwidth. The obvious implication of this in terms of computational analysis would be to restrict analysis to harmonics 1 to 8. However, such an approach is flawed. Whilst it has been shown that the ear cannot resolve higher harmonics into separate frequencies, it still hears the effect the presence of higher harmonics have on a note.

- **Intensity**. As has already been mentioned, the human auditory system can detect a remarkable range of sound intensities. Other studies have examined how sensitive the ear is to changes in sound intensity levels. The threshold of the smallest difference of signal intensity detectable by the ear is known as the Just Noticable Difference[4]. Moore *et al.*[61] investigated the effect the intensity of individual partials have on the perceived pitch of complex tones.

- **Temporal**. Research shows that the ear is highly sensitive to the timing of musical events. Krumbholz *et al.*[52] report that the human auditory system can detect peaks in basilar-membrane motion with a resolution as little as 1-20 microseconds. The word timing can also refer to the order in which musical events are perceived. Ronken[77] investigated and showed that the human auditory system is able to detect phase and power differences between pairs of clicks. Darwin and Ciocca[23] investigated the effect the timing of a mistuned component had on the perception of a harmonic complex tone. They showed that if the mistuned harmonic started too early it made no contribution to the perceived pitch of the complex tone, even thought it was present throughout the complex tone.

These different measures relate to the concept of grouping (also known as stream segregation) and are discussed in this context in Section 2.3.2.

2.3 Hearing and Recognition

Humans respond to auditory stimulus whilst still in the womb[49]. Saldanha and Corso[80] show that the identification of a sound source improves with practice. In terms of sound source identification, an adult human typically has access to years of learning experience (or practice). A computer only has the information it has been programmed with. Thus any computerised system must at some stage rely on a database of information that relates to the the auditory task it is trying to emulate.

However, the human auditory system is subject to error. Deutsch[24, p.127] and Bregman[10, pp.21–29] both report on experiments which show that it is possible to deceive the brain. They both cite the example of Dannenbring's experiment (published in 1976, Canadian Journal of Psychology, pp99-114) in which sections of a sine wave tone that changed in frequency were replaced by bursts of noise. The experiment demonstrated that the brain was deceived in that it perceived a continuous tone. Deutsch[24, pp108–114] also reports other illusionary effects where the brain was unable to detect the reversal of a pair of headphones used for listening to a sequence of tones.

Mêtois[65] states "*our flexible and somewhat puzzling perception of time tends to mislead us in expecting much more from a computer than we can achieve ourselves.*" He comments that when the brain makes a mistake it is able to recover "*in a manner that makes it imperceptible to us.*" Any developed system should therefore be able to recover from errors in a graceful manner.

2.3.1 The importance of context

The context within which music is listened to plays a significant role in reducing the scope of what the listener can expect to hear. This context can be imposed in a number of ways. One example is attendance at a concert where the type of performance is known and expected. The scope of musical context provided by the phrase "string quartet" is considerably smaller than that provided by "symphony orchestra". An attendee of a performance given by a conventional string quartet can expect to hear only three different types of string instrument. Thus *a priori* knowledge of a given situation allows a human to visualise and anticipate what they are going to hear. This anticipation is realised when the actual performer and instrument can be seen and confirmed audibly when the performance commences.

Visualisation (real or imaginary) of a sound source aids the identification of

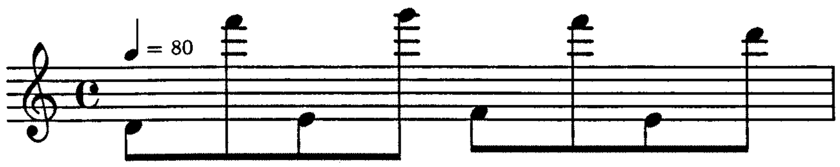


Figure 1: A series of notes.

the source. Risset and Wessel[26, p132] argue that listeners were more able to identify synthesised instruments when they could ascribe something they heard to a physical action - a gesture - despite the fact that no such movement was used to generate the sound. Thus visualisation of the gesture and therefore the instrument associated with it aids sound source identification.

2.3.1.1 Muscial context Context also plays an important role at note level. The note to note transients which occur when a series of notes are played on the same instrument are the temporal characteristics of music itself. Grey[44], in his milestone work on timbre, states that:

> "*The various components of timbre recognition which exist between a set of notes have not been given attention in the literature of timbre perception, in that experiments have universally been done with single, context-less musical tones.*"

Grey's observation highlights the relationship between timbral and gestural analysis. A product of this relationship is the "screech" like sound produced by a steel strung guitar when the guitarist moves between chords.

2.3.2 Stream Segregation

Stream segregation is a phrase promoted by Bregman[10] and is a means of describing how the ear-brain combination channels auditory information. The supporting hypothesis is that when presented with audio stimulus the brain makes use of up to nine different methods of stream grouping[24, p.118–127]. Apart from grouping by amplitude these different methods can be reduced into two main areas:

- **Temporal.** Reference has already been made to research concerning the effect timing has on the human perception of the onset of a note. Thus the

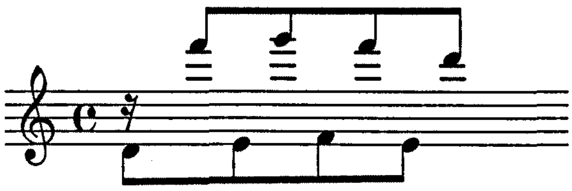


Figure 2: The notes from Figure 1 in a different layout.

timing of musical events is taken into account when determining whether fusion takes place.

- **Pitch**. Research[10, p.58–67] has shown that the perceptual separation of notes in Figure 1 depend on both frequency and time. If the notes are played at a slow tempo they will be heard as one single fused stream of sound. At a certain threshold tempo, the high notes will separate from the low notes. The overall perception would be that of two sound streams, even if the notes were played on the same instrument. This perception is visualised in Figure 2, which shows the same notes in a different layout. The separation point also depends on the pitch interval between the high and low streams.
  Other research into the grouping of pitch has been carried out at the harmonic level. Darwin[22] reports on the effect the presence of one sound has on another in terms of perception and the way the sounds are subsequently grouped. Darwin showed that when a tone starts or stops at a different time from other tones it forms its own perceptual stream.

The conclusion drawn from the above research is that in terms of the human auditory system temporal information takes precedence over pitch.

Bregman[10] suggests that our auditory systems employs two methods to build mental descriptions of events in a current (sound) environment. The first method is described as a primitive process of auditory grouping, where the incoming sound energy is initially broken down into a large number of separate analyses. The auditory system then has to decide how to group the results of the analyses so that each group is derived from the same environmental event.

The second method is described as governing the listening process by schemas that incorporate our knowledge of familiar sounds. Each schema incorporates information regarding a specific event in the auditory environment. It is thought that when schemas become active they detect their particular patterns.

Cooke and Ellis[17] provide a useful summary of streaming/grouping cues in their report on the auditory organization of speech in listeners and machines.

2.4 Chapter Summary

The following aspects of the human auditory system are considered to be pertinent to the design of a computerised listening system.

- Apart from filtering at high frequencies, the ear relays all sound that it receives to the brain.
- The human auditory system rejects noise or extraneous sound at a conscious level. All sound is heard, but a choice is made whether it should be ignored or acted upon.
- The concept of grouping is fundamental to understanding how the human auditory system organises and perceives sound.
- Temporal information is the predominant factor when grouping sound.
- Error recovery should be graceful.

3 Gesture Taxonomy

3.1 Defining Gestures

The purpose of this chapter is to rigorously define (within a musical context) the scope and meaning of the word "gesture".

The request "Please play that passage staccato" is readily understood by musicians, but is subject to interpretation because every musician plays staccato notes slightly differently[2]. Very little research has been done in the field of gesture taxonomy, particularily within the context of auditory analysis. Before meaningful analysis can take place, gestures need to be identified and formally defined.

3.1.1 What is a Gesture?

> **gesture** *n* **1** a motion of the hands, head or body to express or emphasize an idea or emotion. **2** something said or done as a formality or as an indication of intention.

This dictionary[28] definition encapsulates three concepts that are key to considering gestures in a musical context:

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

Figure 3 shows how the above concepts are related. Each item is discussed, in reverse order, in the following sections.

3.1.1.1 Ideas and Emotion With reference to Figure 3, the composer records their ideas on a score which is performed according to the interpretation of the musician. Much has been written on the relationship between the performance of a score and the score itself. Seashore[84] developed a means of recording physical aspects (e.g. a measure of how long a pianist held down a particular key) of a performance in addition to the sound. This allowed him to not only compare a performance with its score, but also allowed for comparison between different performances of a score. The data produced by Seashore

[2]This is further complicated by the fact that it is virtually impossible for the same musician to produce identical staccato notes [16].

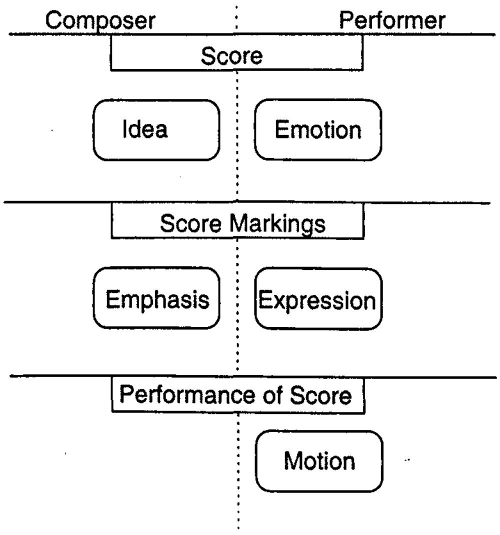


Figure 3: Showing the relationship between composer and performer in terms of a musial score.

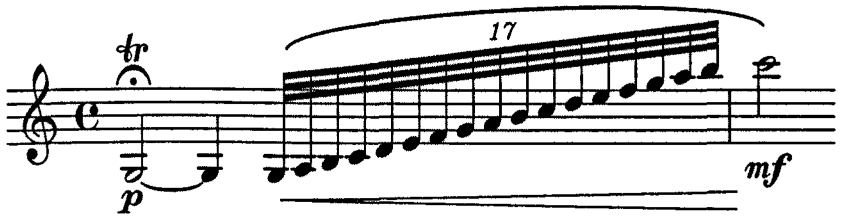
is essentially the same as that now produced by MIDI keyboards. Bresin and Battel used MIDI data to analyse different performances of Mozart's Sonata in G Major[11]. Rather than relating the results of their analyses to the actual gestures used to produce the data, they instead used the analogy of locomotion. Different playing styles were judged to be the equivalent of walking or running, for example. Similar work was carried out by Friberg *et al.*[40] who examined whether the original motion of different gaits used to produce sound via purpose built transducers, could be perceived by a listener. They concluded that each tone could be catergorised in terms of motion. Gabrielsson[41] highlights four categories of timing (tempo, different classes of duration, articulation and deviations from mechanical regularity) which can be varied within a peformance. He discusses how variations in these catergories within a performance, result in various motional-emotional responses from the listener.

3.1.1.2 Expression and Emphasis Score markings are the means by which a composer can communicate their thoughts beyond that of the constraints of the staff, to give an indication of how the music should actually be played. The level of communication goes from a simple pause sign above a note to a detailed written instruction (e.g. Tchaikovsky's instruction (in Russian) in the percussion score for The Nutcracker, to strike and then choke a suspended cymbal with a soft beater). A limitation[3] of score markings is their subjectiveness:

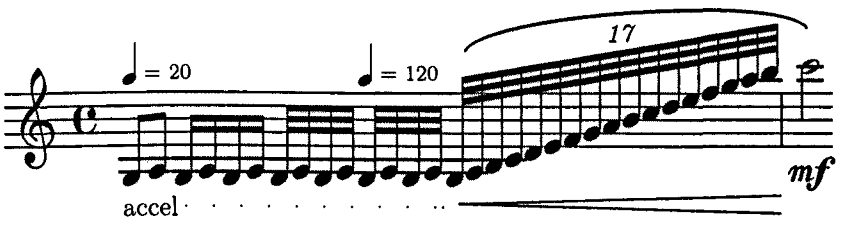
A pianist was playing requests. A viola player enquired:
"Do you know the famous one with the fast trill at the start?"
"I'm not sure that I do." replied the pianist.
"You must do", said the viola player, "it goes..."
- the viola player then hummed the opening of Für Elise.

Miller and Heise[59] show that there is a subjective point, in terms of pitch, where repeated notes are perceived as a trill rather than a series of unrelated, interrupted tones. They used a fixed trill frequency of 5Hz meaning that their *trill threshold* relates solely to pitch. A trill frequency threshold has both an upper and lower limit. The upper limit is ultimately related to the temporal resolution of the ear, of which much has been written [62, pp 163–183][10, pp 59–67], but not in the context of trill resolution.

Rhapsody in Blue by Gershwin starts with an example of the lower trill frequency threshold. The Clarinet's solo opening is scored:



Having never seen the score, the author assumed (when hearing the piece performed) that the solo was scored:



This assumption was influenced by the scale following the trill. This shows the influence that context has on perception: a "fast" piece is likely to have fast trills, wheras a "slow" piece is more likely to have slower trills.

[3]The word "limitation" is used here in an objective sense within a gestural extraction context. In the context of performance, subjectiveness is usually seen as an advantage as it allows freedom of expression.

Therefore, the definition of a given gesture can not be given in black and white; areas of gray have to exist in order to acommodate subjectiveness.

Musicians often vary tempo to make their performances more expressive. Dixon[31] presents a system for the tracking of the beat and therefore the tempo of a performance. Extraction of such data allows the performances of different musicians to be compared. Systems which perform similar comparisons but by different means are discussed in Section 5.2 of Chapter 5.

3.1.1.3 Motion. Music and movement are inextricably linked. There is much research that investigates the relationship between movement and sound. In particular this research is centered on movement that can be seen. The conductor's jacket[55] is a device that was made for recording expressive musical gestures at an orchestral level. It provided a specific means of corrolating a given movement (from the conductor) with the resulting sound produced by the orchestra. Wanderley[9] used Infra-Red markers and video cameras to record movement data. He stated that:

> *"Among the various works on musical performance, few studies have focused on the gestural behaviour of instrumentalists. These have shown that musicians not only perform skilled movements directly related to sound production, but also movements that seem not to have a straight link to the generation of sound."*

Wanderley investigated the effect movement of an instrument (in his case a clarinet) had on the sound it produced.

Dahl[19] presents her observations on the playing of an accent by percussionists. She used motion sensors to track the movement of LEDs attached to the percussionist's right arm and drumstick. Although there were large differences in the way each percussionist interpreted a given task (playing an accent within a sequence of notes), Dahl found that the different approaches resulted in the common outcome of raising the drumstick to a greater height than usual prior to playing the accent.

Askenfelt[3] modified a violin bow so that it contained a length of resistance wire and strain gauges. When used with a suitably modified violin he was able to measure bow motion, force and position, showing how each of these parameters varied according to the style or type of performance.

Other research looks at how gestures not normally associated with (traditional) musical instruments can be used to control electronic or virtual instruments. Trueman and Cook[90] have developed a system that uses the gestures associated with the violin to play a new electronic instrument (BoSSA - Bowed-Sensor-Speaker-Array). Camurri *et al.*, report on their EyesWeb project[14], the aim of which is to develop a system that will perfrom real-time analysis of body movement and gesture. Gestural information captured by video cameras can then be used to both generate and control sound and music. Müller[66], reporting on the development of the *RUBATO* project (an application for producing computer-aided musical performances), describes how support for the gestural control of performance is being incorporated into the *RUBATO* system. Other systems which make use of gestural input for synthesis purposes are *CORDIS-ANIMA[13]* and *MOSAIC*[64].

There are movements (e.g. of the tongue when playing wind and brass instruments) which are very difficult to physically measure and analyse. However, the results of such movements are heard in the sound produced. Sapir[81], in a paper concerning the gestural control of an audio environment, presents a review of current gesture tracking technology. Having described various sensor and MIDI based methods, Sapir states:

> *"It is also possible to indirectly capture gesture by analysing the sound produced by an instrument and deducing information about the gestures applied by a performer to generate it."*

Having made this statement Sapir comments that such information would be provided by "*very sophisticated programs*". No further reference is made to this form of gesture capture. The fact that a given movement can be identified from the effect it has had on the resulting sound is fundamental to this thesis.

In a performance context, a musician with a stiffer posture is usually perceived as a player of "cold" music[35]. This argument relies on visual cues to determine the expressiveness of a performance and will not be discussed here.

This chapter, in the context of auditory analysis, seeks to identify basic movements and therefore define basic gestures.

3.1.2 Sound Movements

The identification of movements which are gesturally relevant is a huge task, mainly due to the large number of orchestral instruments. Each instrument has a set of movements which a musician uses, often skillfully combining several, to produce the desired sound. For the purposes of analysis, relevant movements

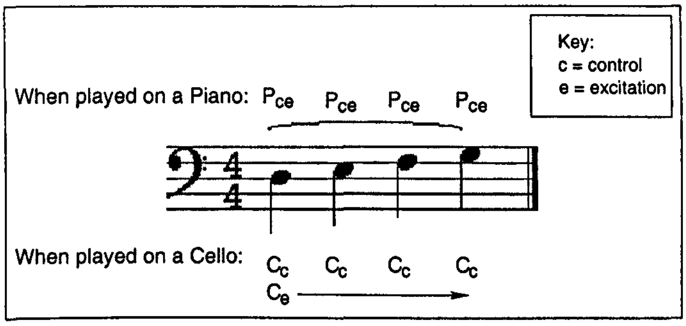

Key:
c = control
e = excitation
When played on a Piano: P_{ce} P_{ce} P_{ce} P_{ce}
When played on a Cello: C_c C_c C_c C_c
C_e

Figure 4: Differences in gestures between a cello and piano playing a slurred passage.

will be identified, defined and grouped according to type. Movements will be classified in terms of the effect they have on the sound produced by the instrument. Using movement as the basis for gestural type definition results in high level generic definitions which are instrument independent. Thus instrument specific gestures, even those which are unique, can be grouped with other gestures under a common type.

When considering the movements associated with playing a cello it is immediately apparent that there are two distinct movements, of the left and right arms/hands, which control the sound. Movement of the right arm i.e. the bowing arm (assuming the right hand is holding a bow resting on a string) directly causes sound to be made. Whilst movement of the left arm does not (with the exception of left hand plucking) directly produce sound, it can fundamentally alter the sound produced. There are clearly two different types of gesture, defined as follows:

1. A gesture which causes sound
2. A gesture which alters the sound which is either playing or about to be played.

Item 1, *causing* sound, is defined as an *excitation gesture.* Item 2 *controls* the sound that already exists as a result of an excitation gesture, and is therefore defined as a *control gesture.* Both aspects (i.e. left and right arm movement) of the gestures associated with a cello are combined into the gesture of striking a selected piano key. This gesture produces a note from the piano, the pitch being determined by the selected key. The difference between the two instruments, in terms of gesture, is illustrated in Figure 4.

The excitation gesture of the cello spans the four control gestures which alter the note being played, whereas each note played on the piano is a combination of control and excitation gestures. The gesture requirements of the piano and cello can be matched by removing the slurred phrasing, as shown in Figure 5.

When the mechanics of playing the cello are considered in more detail it emerges that there is more to playing the cello than first meets the ear. Figure 6 shows a series of notes which can be played either with finger 3 (in fourth position) on the G string, or with finger 1 (in first position) on the D string. Musical sense[4] helps the cellist decide how to play such notes. It is possible to play this sequence sixteen different ways (in terms of fingering), although only a handful would actually be useful.

The choice made by the cellist has an impact on the gestures used to produce the note. For example, if the first "e" is played on the D string, the next two notes played on the G string and the final note played back on the D string, then both the left and right hand have to simultaneously select the correct string. It could be argued that there is a control gesture associated with the third note of this example: a gesture to select the G string for the 2nd note and then another gesture to reselect the same string.

Using only auditory cues, it is very difficult to detect which method, and therefore which string, was used to play the notes in Figure 6. Reselection of the same string can also be defined as not selecting one of the three other strings.

[4]Common sense in a musical context, derived from experience. For reasons of tone consistancy it would usually be desireable to play such notes on the same string [87, p.118].

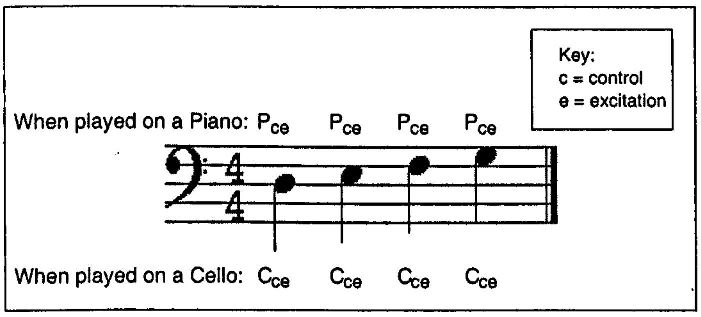

Key:
c = control
e = excitation
When played on a Piano: P_{ce} P_{ce} P_{ce} P_{ce}
When played on a Cello: C_{ce} C_{ce} C_{ce} C_{ce}

Figure 5: Similarities in gestures between a cello and piano playing the same music.

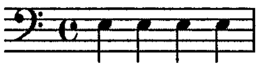


Figure 6: A series of "e"s.

Similarly, reselection of the same note by the left hand cold be defined as not selecting any other possible note on a cello. Failing to select something will not be interpreted as a gesture; lack of change, in terms of note selection, points to a lack of gesture. The influence of a control gesture can continue over similar notes. A change in note pitch indicates the presence of a new control gesture. Therefore in the context of auditory analysis:

- A gesture is an action which causes a definite change in the sound output.

Two types of gesture have been identified:

- An **excitation** gesture - sound causation
- A **control** gesture - alters sound which is either playing or about to be played.

Hence for the example in Figure 6 there is one control gesture when the note is selected by the left hand and four excitation gestures from the right hand.

These two gesture types incorporate and extend the concept of "musical gestures" put forward by Mêtois in his work on Musical Sound information [65]. The use of the phrase "musical gestures" restricts the scope of his work to musical control gestures. The focus of this thesis is gestures that relate to the physical movements used to articulate the sound produced by a musical instrument.

3.1.3 Information Contained Within Audio Signals.

When listening to an instrument the result of gestural input can be heard, therefore it is possible to infer the identity of the gesture used to produce the sound.

Different gestures require the extraction of different information. For example, the gestural difference between a plucked and bowed note is found in data at the beginning of a note, whereas the difference between slurred, legato and staccato bowing is inferred from data at both the beginning and end of notes. The plurality of "notes" in the second example is intentional. It indicates how gesture identification can often depend on more than one note instance. Thus, gesture extraction also depends on the contribution each note makes to the overall musical context.

3.1.3.1 Ambiguous Information.

The group of movements normally associated with playing a piano contain a problem case: the sustain pedal.

If a pianist holds down the sustain pedal whilst playing the music in Figure 4 each note will overlap over a subsequent note. The same effect can be a achieved by holding down each key once it has been pressed. From an auditory perspective, it is virtually impossible[5] to tell the difference between these two methods of playing and therefore virtually impossible to detect whether the sustain pedal has been used. This example proves that metameric gestures do exist[6]. That is, two different means or gestures produce the same perceived end, or sound. Therefore, just as it is impossible for a human to hear all types of gestures it is also impossible for a system to identify all types of gesture.

3.1.3.2 Conventions.

It can be argued that dropping a suitably large and heavy object (usually an anvil) onto a piano from a great height is a gesture that produces a rather unorthodox, piano sound. The scope of "gesture" is therefore restricted to the group of physical movements conventionaly associated with playing an instrument.

Accepted score notations and normal playing techniques are also used to restrict the scope of analysis. Furthermore, the scope of gestures identified and defined in this thesis will be restricted to a small subset of the total number of possible gestures for all instruments.

3.1.4 The General Definition of a Gesture

A gesture is a positive action which provides some form of conventional input to a musical instrument, as a result of either accepted

[5]For the sake of technical completeness, in theory it is possible to tell the difference between these two gestures. When the sustain pedal is held down, all strings are undamped and resonate in sympathy with the strings that are struck. Detection of these quiet sympathetic resonances would in theory allow a system to identify that the pianist is pressing the sustain pedal. The success of sympathetic resonance detection would not only depend on the pitch and loudness of the actual notes played, but also on near perfect recording conditions. In practice the notes played would usually mask any sympathetic resonances.

[6]Bregman [10, p. 122] likens the concept of metameric timbres to metameric colours where different combinations of colour spectra can result in perceptually identical colours.

score notations or normal playing techniques, so that the effect of the gesture is heard in the output of the instrument.

This general definition enables the movements associated with a given instrument to be examined and identified as gesturally relevant. Valid movements can then be explicitly defined in a gestural context. Emphasis on movement distances this definition from that provided by Dubnov[32] who defined gesture in terms of changes of texture over time (where texture relates to psychoacoustic factors that evoke various emotional associations).

3.2 Identification of Gestures

As discussed at the beginning of this chapter the interpretation of score annotations depends on the individual musician. The consequence of this when defining a gesture is that a definition cannot be absolute; it must relate to a given set of boundaries or conditions. Difficulties arise where boundaries overlap e.g. the difference between spiccato or staccato bowing.

Some gestures are common to a number of instrument families and some are not. Stringed instruments, apart from the piano, can be plucked in order to produce sound. However, only certain instruments within this family can be bowed e.g. it is not possible to bow individual strings on a guitar.

This section uses the information presented in the previous section to examine specific instruments and identify movements which can be classified using gestural nomenclature.

3.2.1 The Scope of this Research

Gestural analysis is a vast topic. The scope of research presented in this thesis is therefore restricted to gestures associated with the violin, cello and the oboe. The specific aims and objectives of the research is discussed at an instrument level in the following sub-Sections.

This extraction of gestures will ultimately provide any developed system with the means of distinguishing between instruments at a gestural, rather than timbral level. Instrument identification is an area which gestural analysis could impact heavily upon. The gestural information could be used to determine the instrument family[7] and subtleties of timbre to identify the actual instrument.

[7]The presence of an instrument family specific gesture in an acoustic signal could be used to eliminate other instrument families. For example the presence of bow noise would eliminate woodwind and brass families allowing instrument recognition to concentrate on strings.

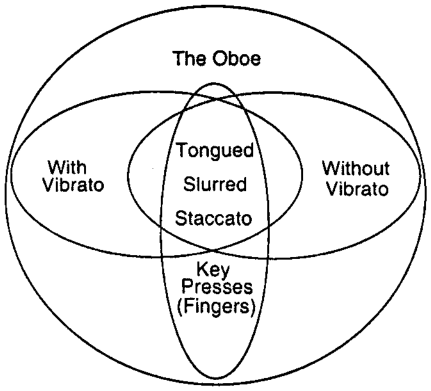


Figure 7: Groupings showing the interaction of oboe gestures.

3.2.2 The Oboe

The oboe is a member of the woodwind family and is usually considered to be a solo instrument. It consists of a reed and then three sections which make up the body. The reed is made up of two seperate reeds bound together onto a tube of metal (known as the staple), the lower half of which is covered in cork to help it mate with the body of the oboe. Producing sound from an oboe is far from easy. It is not simply a matter of blowing air through the instrument. Attempting to do so can lead to the conclusion that it is impossible to get air through the instrument. The opening between the reeds is very small, therefore high air pressure is needed in order to produce sound from an oboe. Only a highly skilled player can generate and maintain the required pressure so that a fine tone can be produced across the entire dynamic range of *ppp* to *fff*.

Movements associated with the oboe are that of embouchure, the tongue and the fingers. Embouchure is the French noun used to describe the position of the lips needed in order to play the oboe. Correct embouchure ensures that the correct air pressure is provided for different notes (the higher the note the greater the air pressure required). Tensing and relaxing the embouchure can sharpen and flatten a note respectively. Such movements however are usually employed in a subconscious manner by the experienced oboist to smooth out the nuances of certain notes on their instrument before they sound the note. Such movements are so subtle that they go beyond the scope of an identifiable gesture.

Breathing also falls into this category. The existence of a note depends on

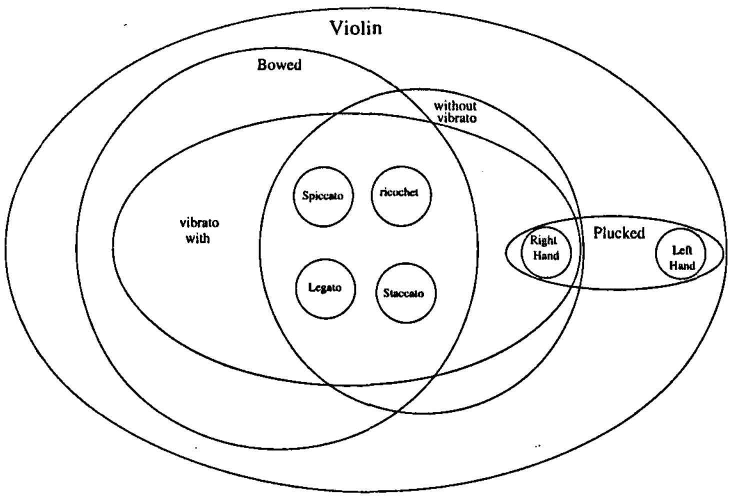


Figure 8: Groupings showing the interaction of violin gestures.

the oboist exhaling, which means they have also had to inhale. The dynamics of a note are varied by changing the speed at which the oboist exhales. Part of the skill of playing the oboe is knowing when to breathe in and out and doing so in a way that is non-obvious to the listener. It is for this reason that this kind of movement falls into the non-gesture category. Figure 7 shows how the remaining oboe movements are grouped. In many ways the tongue acts like the dampers on a piano; it is used to prevent the reed from vibrating and therefore sound eminating from the instrument. Rothwell [78] writes:

> "*Tonguing is the equivalent of bowing on a stringed instrument. when you have real tongue control, be guided by your sense of style and musicianship to use it in the fullest possible way, as a fine string player uses the bow.*"

The ability to articulate notes is obviously a desired ability for any musician. In the context of the oboe this depends on the coordination of breathing and fingering. The gesture to be investigated is therefore finger movement (key presses). That is, to determine from the acoustic signal produced, whether finger movement has occured at the appropriate time.

3.2.3 The Violin

The violin, a member of the string instrument family, is the most common instrument in the modern orchestra. Although not physically part of the instrument itself, a bow is vital if the full range of violin sounds are to be achieved. Thus movements associated with both the violin and bow will be considered. Sound is usually produced by drawing the bow across one of the strings. The bow is used to control the dynamic level and tonal quality of the sound produced, making its proper use a fundamental part of playing the violin. Some state that "the right hand is the soul of the violin"[79, p.72]. Sound can also be produced by plucking the strings with either hand, though this is predominantly done with the right hand.

The violin produces a complex tone often made up of 20 or more partials [79, p.11] which arise as a result of its shape and construction. Figure 8 shows how the movements associated with playing a violin are related and dependant.

Figure 8 is also applicable to the cello. The violin and cello movements to be investigated will be the articulatory gestures of plucking and bowing. There are two levels to this research:

1. The distinction between the method of excitation, i.e. plucked or bowed?
2. In the case of a bowed note: the distinction between different styles of articulation (gesture) e.g. legato or staccato.

At set of discriminators will be identified which will enable a system analyse the acoustic signal produced by the instrument and make a decision as to the gesture type used.

3.3 Extraction of Gestures

The gestures described in this chapter either shape or form musical notes. The extraction of gestural information from audio signals is therefore dependent on the ability of a system to identify note events. The next chapter investigates the definition of a note and then reviews methods of identifying them.

4 Note Detection Methods

In order to extract gestural information form a note event it is imperative to be able to identify its onset point. Before reviewing note onset detection methods, it is first necessary to consider what is meant by the word "note".

4.1 The Definition of a Note

The sections of a note have traditionally been described in terms of an amplitude envelope, shown in Figure 9. The terms "attack", "steady-state" and "decay" are not ideal descriptions of a note in a gestural context. Gordon[43] addresses this problem by drawing a distinction between the perceptual ***attack*** and ***onset*** times. The need for such a distinction arises from the fact that most notes do not follow the traditional shape of Figure 9, for example:

- The majority of notes produced by plucking or striking do not have a traditional "steady-state"[73, p70].
- Even bowed notes do note have a "steady-state"[3].
- A bowed note has no "decay" when the bow is brought to a sudden stop and left in contact with the string.

Consider the violin note shown in Figure 10. It is a note played on the G string of a violin which gets progressively louder.

Traditionally, this note would be described as having a long, slow attack between (points B and D) followed by a fast decay. However, when listened to it is clear that the note onset is at point A (highlighting the difference between

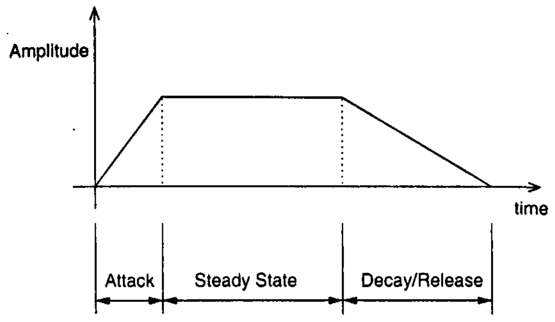


Figure 9: Traditional breakdown of a note into sections.

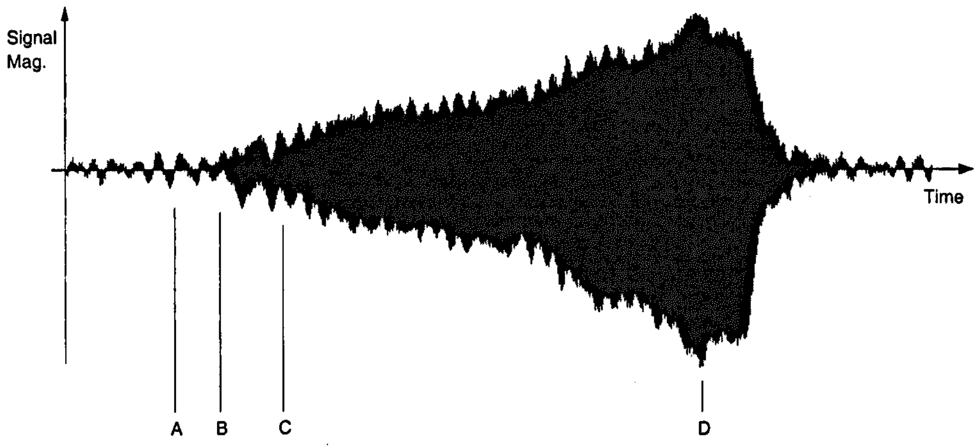


Figure 10: The amplitude waveform of a violin note.

what we can see in the waveform and what we hear). At point C the sound of a violin can clearly be identified. Hence the traditional definition of "attack" is called into question. The section between points C and D contains an established note increasing in volume. The word "attack" is therefore misleading and should not be used to describe this section.

4.1.1 When does a note actually start?

The phrase "the start of a note" is ambiguous. If a wave file contains a single, digitally generated note which occurs after exactly one second of silence, then it is obvious that the note begins one second from the start of the file. It is very difficult however to state when this note will be accurately perceived by a human listener. Whilst humans can determine whether an ensemble of instruments begin to play at the same time, the point at which we become aware of individual notes will always lag the actual physical start of each note. This is partly due to the fact that humans hear everything retrospectively and partly because it takes some time for the brain to process the sound information presented by the ears[96].

4.1.1.1 The ambiguity of the start of a note Consider the case of a violin where the beginning of a note is a mess of noise as the bow is scraped across the string; sound can be heard, but cannot be identified as the note itself. It is from this mess of noise that the harmonics of the note grow and establish

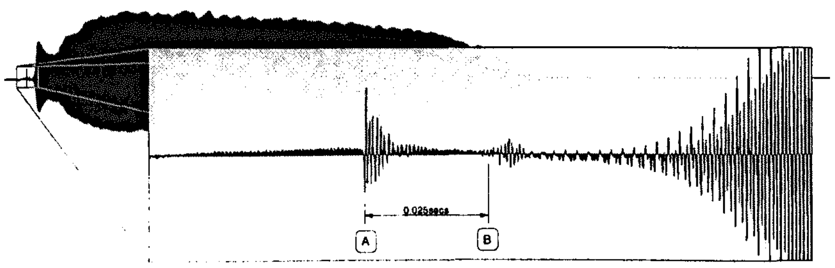


Figure 11: The beginnings of a note on an oboe.

themselves. This situation is made worse when an ensemble of instruments begin playing at the same time. It is impossible for a human listener to immediately state which instruments are playing.

The question of where a note starts is subtly different again in the case of the oboe. Instead of a bow exciting a string, air is forced through a double reed. Figure 11 shows that at the beginning of this particular oboe note there are two separate clicks (labelled A and B) before the note starts. When the clicks are isolated and heavily amplified, a distinct pitch can be heard.

Such clicks are caused by key presses[8] which are usually masked by the sound of other instruments or even other notes played on the same instrument. Does the note in Figure 11 start at point A, or at an unspecified point after point B? To complicate things further, in some cases key clicks can be oberved in unexpected places.

Figure 12 illustrates that a key press for the second note in a scale occurs immediately after the first (previous) note. This key press is heard at the pitch of the second note, giving the first note a clipped ending. This event is very short and therefore extremely difficult to detect. The position of the occurrence of the key press contributes to the perceived quality or purity of the note. Key press events do not occur exclusively at the end of notes, in some instances they can clearly be indentified during the interval between two notes. Such gestural information when extracted could help a performer to consistently produce notes free from key press interference.

[8]The note in Figure 11 is a Bb which is made by holding down the first two fingers of the left hand.

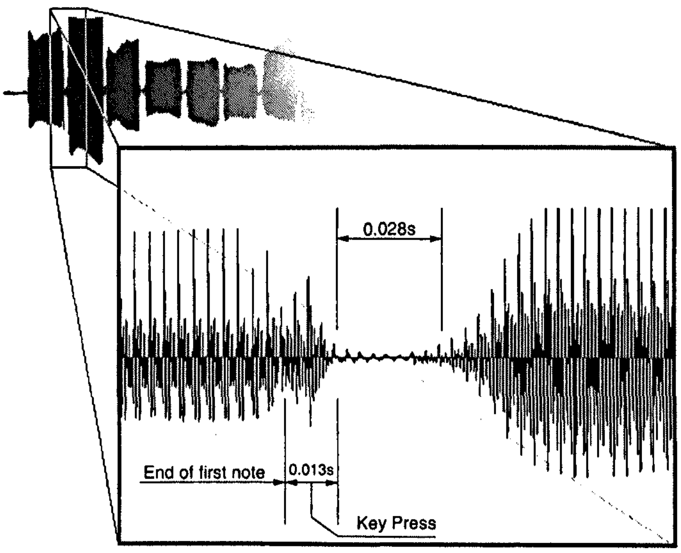


Figure 12: A key press at the end of a note within a scale of tongued notes.

4.1.2 When does a note actually end?

Even in the context of an isolated note, the actual physical end of a note can be very different from its perceived end point. A cymbal continues to vibrate for a long time after the point at which sound can no longer be heard. Non-isolated notes present a number of difficulties:

- Some notes fade imperceptively away, masked by the presence of other sounds.
- The presence of a repeated note of the same pitch produced by the same instrument causes a note to be replaced, rather than to end.

4.1.3 The general description of a note

For the purposes of this research:

- The start of a note is to be taken as the point at which it becomes possible to distinguish note information from noise. The scope of "note information" includes inharmonic partials and onset transients, e.g. the sound heard at the beginning of a bowed note.

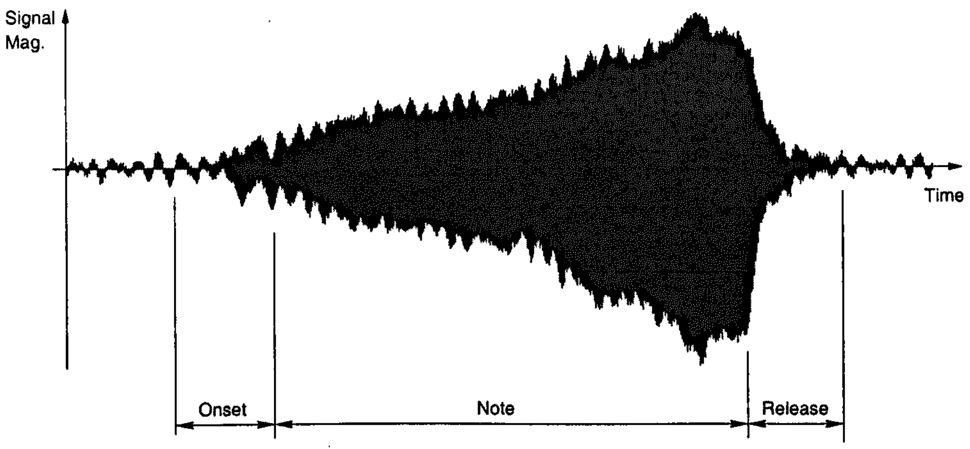


Figure 13: The new descriptions to be used for various parts of a note.

- The end of a note is to be taken as the point at which it is no longer possible to identify the separate harmonics which formed the note. Repeated notes of the same pitch are treated as a special case. If a note is repeated on the same instrument then the start point of the repeated note is also the end point of the previous note.

The criteria for what constitutes the start and end of a note is subtly different. The definition of the start of a note accomodates the need to take onset events into account, the importance of which is discussed in Section 4.2.

A new naming convention for an individual note is proposed, shown in Figure13.

- The "onset" section of the note is defined as the part of the note during which each harmonic of the note establishes itself. The noise associated with this is also included as a vital part of the onset information. The end point of the "onset" section is currently indeterminate. It is best defined as the moment when no further information is needed to ascertain that a note exists.

- The "note" section is defined as the part of the wave during which the note is fully established. In this section the dynamics of the note can increase or diminish. This section ends after the traditional end point of the attack portion (point D in Figure 10).

- The "release" section of the wave is defined as the part during which the wave diminishes to nothing. This part of the wave provides useful information towards gesture identification.

In many ways this note description is a tailoured version of the traditional description illustrated in Figure 10. The new boundaries of each section depend on auditory rather than visual information. It should be noted that the boundaries of each section are fluid rather than fixed. If a note is played so that it gradually diminishes in volume to nothing, it contains no defined "release" section. Systems which identify instruments or timbre are not usually concerned with the release of a note. For gestural analysis purposes, each section of a note is important. Using the violin as an example:

- The onset section contains information as to whether a note was bowed or plucked.
- The note section can be used to detect the presence of vibrato.
- The release section will allow the type of bowing to be inferred (e.g. staccato, legato, spiccato etc).

Having established the importance of each part of a note, the remainder of this chapter considers existing approaches to the problem of extracting note information from audio signals.

4.2 The Nature of Onset

Early research in the field of audio signal analysis concentrated on features derived from the steady state portion of the sound[48]. It is known that a human can identify a sound within the first 10ms[96, p.111] of hearing it, suggesting a large amount of information is contained within this part of the note. Other literature[97, 56, 94] highlights the need to examine the start of a note. Information contained within the onset transients of a note provide the listener with musical "fingerprint" information such as the sound source and method of excitation. Systems which do not make use of both temporal and spectral features will never achieve performance comparable to the human ear-brain combination[34].

The majority of work on timbral analysis has focused on the steady state part of the wave. This is possibly because in this section of the wave transients and noise have died down, and it is therefore cleaner and easier to analyse. Young's comments on experiments that are only concerned with the steady state portion of a complex tone are particularly relevant:

> "*In discussions of timbre (tone quality) it has long been the custom to state that the differences in quality of tone are solely dependent on the occurrence and strength of partial tones. Although H. von Helmholtz[48], in making this statement recognised that the characteristic tone of some instruments is dependent upon the way the tone stops and starts, he chose to restrict his attention to the 'peculiarities of the musical tone which continues uniformly' and to consider as musical only those tones with harmonic upper partials. Many writers since have adopted these simplifying but not realistic assumptions; according to such simplifications the piano is not a musical instrument! The transient parts of a sound contain important clues by which different instruments are identified. A sustained high tone on the clarinet, for example, is practically indistinguishable from the same tone (sustained) played on the flute, but the initiation of the sound is likely to be noticeably different on the two instruments.*"[97]

Martin[56], when writing about attack transient properties, states that:

> "*It is evident from the available human perceptual data that the attack transient of an isolated musical tone played on an orchestral instrument contains crucial information for identifying the particular instrument that generated the tone. It is not clear, however, which aspects of the attack transient provide the essential information. Indeed, it is not even clear how to define when the "transient" ends and the "steady-state" begins. The literature is at best equivocal on these issues.* "

There is much research that demonstrates the influence that the start of a note has on perception. Grey and Moorer[45], in their conclusions regarding the perceptual evaluations of synthesized musical instrument tones, comment on the "*extreme importance of the offset*". This statement was made as a result of investigation into the perception of musical tones with no "attack" portion. Saldanha and Corso[80] used tape splicing to create a variety of sounds that allowed them to evaluate the contribution of each section of a note to perception. They concluded that:

> "*...the onset of the tone has certain characteristics that aid discrimination, while the stopping of the tone contributes nothing to identification.*"

The second half of their conclusion is open to question; the use of the word "nothing" is sweeping and excludes the possibility that the "stopping" of a tone reinforces an earlier decision. When Saldanha and Corso use the term "stopping" in their article they appear to mean "decay". A pedant would argue that a stopped tone contributes nothing because it has stopped! The argument of the pedant and Saldanha and Corso fall down in a gestural context. The fact that a tone has "stopped"[9] rather than "decayed" aids gestural identification.

Conversely, Fletcher and Sanders[37] in their work on the quality of violin vibrato tones state that "*the beginning and the ending of the tone*" affect the quality of tone produced. They used this fact to increase the "realness" of their sythesized violin tones by adding transient effects to the beginning and end of the sounds they generated. Previous to this Fletcher[36] highlighted the importance of the ending of a violin tone stating that "*certain partials decayed rapidly leaving an afterring*". Berger[6] also used tape splicing methods to create a set of sounds based on recordings of wind instruments. He concluded that the attack and release of a note provide important recognition clues which aid instrument identification. However, his conclusion was based on the simultaneous, rather than independent, removal of onset and decay when creating test sounds.

Fraser and Fujinaga[38] found that the system they used for real time acoustic musical instrument recognition acheived a 10-20% increase in recognition rates when the "attack" portion of a note was used, rather than the "steady state".

Whilst researchers generally concur that there is a large amount of important information in the attack portion of a note, this portion has been little researched to date. Martin[56] surmises:

> "*It has been suggested that the relative onset times of the harmonic partials are important features, as are their attack rates (perhaps measured in dB/ms). Little has been written, however, about how to measure these properties from recordings of real instruments, and I am aware of no published descriptions of techniques for measuring these properties.*"

[9]As in abruptly stopped.

Amongst the "*little*" that "*has been written*" is the work of Nolle and Boner[68] who investigated the initial transients of organ pipes. They presented what would now be regarded as a rather antiquated film-based method for measuring transients. Their work followed the earlier work of Hall[46] who considered the difficulties of measuring transient sounds. Richardson[76] later extended the work of Nolle and Boner by using similar methods and enlarging the scope of instruments included to orchestral wind instruments as well as organ pipes. In related work Schroeder[83] showed the complimentary nature of sound buildup and decay in an enclosure. Martin concludes that:

> "*Attack transient characterization has received frustratingly little attention in the acoustics and synthesis literature. This has the potential to be a fertile area for future research.*"

It is the character of the onset of a violin note that enables the listener to distinguish between a bowed or plucked note. The characterisation of the start of a note is therefore a very important facet of gestural extraction. After the onset section is complete most of the initial gestural information will be lost. This can be shown by considering a note played on a stringed instrument, in this case a violin, using spiccato bowing, as shown in Figure 14 (top).

Although it cannot readily be seen, the crunch of the bow on the string is clearly audible in the attack of the note. However, if the attack portion of the wave is removed and the resulting sound wave auditioned, it sounds exactly as if the string had been plucked, and not excited with the bow. The lower waveform is the result of the same string on the same violin being plucked. Even at this basic level the similarities in the decay of each note are striking[10]. This highlights the importance of the attack portion of the wave in this example. If the attack was ignored it would be very difficult to determine whether the note was bowed or plucked.

4.3 Note Detection Methods

The detection of notes is primarily concerned with the detection of note onset points.

In its most basic form amplitude threshold onset detection is performed by identifying all parts of a sound wave above a certain threshold as note. This

[10]This is to be expected: following an excitation gesture, if left undamped the vibrations of a string will decay in the same manner.

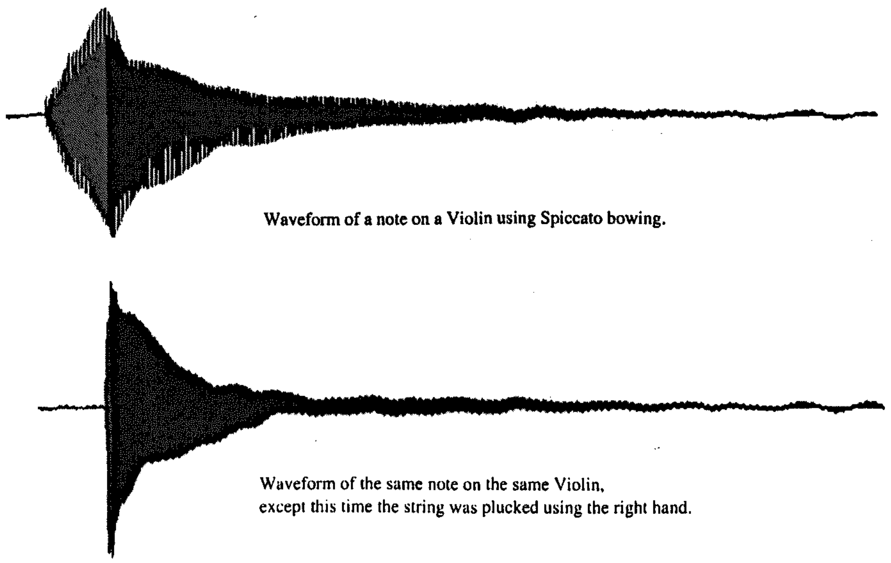


Figure 14: Showing the waveform of bowed and plucked note.

method is a compromise between identifying loud notes at the expense of quieter ones. Amplitude threshold detection can therefore result in loud notes masking quieter notes, producing at best a monophonic tracking of only the loudest notes. This crude method however is still part of some of the most advanced detection systems.

Moorer[63] used a bank of sharply tuned bandpass filters to process a musical signal, allowing the harmonics of each instrument to be extracted from polyphonic music. The actual note was then inferred from this harmonic information. This system was considered a success despite the heavy restraints placed on the definition of polyphonic music. The fundamental frequency of one note was not allowed to overlay a harmonic of another simultaneously sounding note; melodic grouping of notes relied on the restriction that parts would not cross.

A polyphonic piano music transcription system, developed by Scheirer[82], is severely restricted in scope by the requirement that prior to transcription a MIDI score of the piano music must be entered into the system. Scheierer's system employed one of four different methods available to identify the onset of a note. Each method was optimised for a given onset situation. All four methods are undermined by a reliance on previously entered MIDI data when determining whether an actual note has been found.

Dixon furthered work in this field by producing a system which required no prior knowledge of the piano score to be transcribed[30]. Dixon's method used spectral peaks in overlapping FFT windows to give a "*reasonably accurate estimate*" of onset time, relying on the fact that due to the fast attack of a piano note its onset is near to its spectral peak. Dixon went on to produce a subsequent system which was able to transcibe music from other instruments[29]. Another onset detection system presented by Dixon made use of a linear regression technique to find the slope of the amplitude envelope of a signal[31].

The work of Klapuri[51] on automatic music transcription uses an onset detection method which divides incoming sound into six different frequency bands and then calculates the first order difference function of a smoothed amplitude envelope for each band. The results from each band are combined to determine the onset point. This onset point is combined with information from tonal models in order to determine the pitch of the note. As Klapuri admits, this onset method relies on the percussive nature of piano sounds and is not suited to non-percussive instruments, for example string instruments. Klapuri's system is literally a note onset detector; the evolution of a note is not tracked and therefore note endpoints (the knowledge of which being vital for gestural extraction) are not detected. Klapuri[50] reused his onset detection method in a later system which made use of psychoacoustic knowledge. His results confirmed his earlier report of the unsuitability of the method for the detection of non-percussive sounds.

Martin[57] uses a blackboard framework in which a correlation based approach is favoured over sinusoidal analysis. Actual note onsets are found using a system which makes use of first order difference approximation of an amplitude envelope. This approach is virtually the same as that of Klapuri and is therefore not suited for the detection of non-percussive sounds.

Duxbury *et al.*[33] use a hybrid approach to detect what they describe as "hard" and "soft" note onsets. They use two different methods to detect high and low frequency information from which note onsets are inferred. The use of adaptive amplitude threshold levels and filter banks render their approach unsuitable for gestural extraction purposes.

Marolt[54] presents a system for the transcription of music recordings. Adaptive oscillators are used to track partials. The motivation behind a system designed for transcription differs from that for gestural extraction. Transcription systems are concerned with the "steady state" portion of a note. In finding partials, Marolt is seeking to isolate "*the stable frequency components most likely to belong to tones, and discards noisy components*". Such a system is therefore not suited for gestural extraction.

An alternative to the approach of the above systems is that of pitch tracking. Instead of searching for note onset events, note pitches are tracked. Note start and stop times can then be inferred from the output of the sytem. Walmsley *et al.*[95] and Cirotteau *et al.*[15] present such systems. The performance of the system presented by Cirotteau *et al.* is examined in Chapter Five.

The onset detection systems reviewed above are limited in a gestural context because they either rely on threshold levels or remove possible note information from the data by filtering. The human auditory system rejects noise or extraneous sound at a conscious level. That is to say, all sound is heard but a choice is made whether to act following this. This feature is fundamental to the development of a robust note detection system for gestural extraction purposes. If a similar approach is adopted when developing a note onset detection method, the following principles emerge:

- All data must be assumed to be valid. If it is filtered before processing takes place then vital gestural information could be lost.
- Data can only be discarded once the system has established that it does not contribute any gestural information.
- The ear can detect an enormous range of sounds[5] and does not use a threshold cutoff for the elimination of unwanted sound. Therefore the only allowable form of threshold is that which is related to the Just Noticeable Difference threshold of the ear (commononly taken to be 1dB)[4].
- Ensuring that the detection system works with recordings of real instruments. Whilst it is usually necessary to restrict the scope of note onset detection to that of "simple music"[57], such test cases should be recordings of real instruments, rather than digitally generated. Of course this does not rule out using digitally generated test cases for development purposes. Freed and Martens[39] are particularly scathing on this point, stating that "*Research results obtained using oversimplified stimuli have little relevance to perception under natural conditions - they lack ... ecological validity*".
- The end point of a note must also be detected.

4.4 System comparisons

Apart from comparison of the relative merits of a given system implementation, direct comparison of system performance is often not possible. This is mainly due to the use of different material for each system test. Results regarding the number of detected notes do not always permit comparison as such a total can be misleading. No proof is given to show that the total number of detected notes is entirely made up of genuine notes. The total can be offset by the cancellation of missing and extra notes, and the inclusion of "wrong" notes[11], leading to a false total. Performance in terms of temporal resolution for both note start and stop points would also be a useful comparison parameter, but is rarely reported.

Clearly, it would be useful if researchers presented their results in terms of the number of correct, wrong, extra and missing notes and reported on the accuracy of notes start and stop points. These criteria are vital not only for the comparison of different systems, but for the evaluation of the performance of a given system. The next chapter presents an automated system evaluation technique, which provides the means for onset detection system comparison.

[11]A note detected in the correct place but ascribed the wrong pitch.

5 Comparison Techniques

This chapter, having explained the need for a comparison system, examines the existing comparison techniques, drawing on the fields of string matching and (musical) performance analysis. The semantics and rules of error definition are discussed followed by the presentation of a comparator specifically designed for musical comparison. The chapter ends with system tests (of the comparator itself), followed by the testing of Damien Cirotteau's PitchTracker software[15].

The work presented in this chapter is furthered by the papers presented in Appendices A and B.

5.1 The need for comparison

As explained at the end of the previous chapter, comparison of the output of a note detection system with its input provides a quantifiable means of evaluating its performance. Without such comparisons it would be a time consuming and laborious process to manually prove the performance of a note detection system.

The output of the comparator can also be used to compare the performance of different note detection systems using a common set of input files.

5.2 System Specification and Background

A comparator should compare the output of a system with its input, by calculating:

1. The number of extra notes
2. The number of missing notes
3. The number of wrong notes
4. The number of correct notes
5. The accuracy of note onset times
6. The accuracy of note lengths (release time)

Items 1-3 match the editing operations used by Ukkonen[91] in his work on string matching. These editing operations, based on those formulated by Levenshtein[53], are used to determine the edit distance (or minimum cost) of converting string A into string B. At the computational level, pitches from a musical score are represented as a string. Hence score note order is reflected in the order of pitches in a string. Comparison of pitch information is therefore closely related to work in the string matching arena, but not identical to it.

Item 4 is normally implicit in the results provided by items 1-3. However, the purpose of string matching in a musical context is to determine *how* a musician has performed a score. The edit distance which represents the performance of a score is not necessarily the smallest. As a consequence of this, the comparator presented in this chapter uses rules which cause it to find the edit distance which corresponds to the deviation of a given performance from its score. Thus, the number of correct notes becomes an important factor when evaluating the performance of a score. The difference between the optimum edit distance and what will be termed the performance distance, is discussed in Sections 5.5.2.2 and 5.5.2.3.

Items 5 & 6 introduce time dependencies not normally associated with string matching, which move the work presented here into a new, though not unrelated, string/time matching arena.

Whilst similarities with work in the field of string matching have already been highlighted, there are further parallels that can be drawn with the field of musical performance analysis. Performance analysis is typically concerned with examining how a musician has performed a piece of music, be it in comparison to another performance or the performed score. The advent of computerised sound processing techniques have opened up this particular field and allowed musicologists to perform rigorous analysis. Moelants[60] presents a system whereby he is able to analyse both the score itself and perform comparative analysis of the performance of the aforementioned score by three different musicians. Although in depth computer-aided statistical analysis is performed, it appears from his paper that score-performance matching was carried out by hand.

Dillon[27] presents a system to extract salient musical features which permit the comparison of different perfomances and ultimately, recognition of the musicians that produced them. Bora *et al.*[7] present a tool specifically designed for comparison of performances on a MIDI keyboard. Their system compares virtually all the different variables represented in a MIDI file. However, for reasons discussed in Section 5.5.2.2, their algorithm is susceptible to false error matches and as a result does not always find the performance distance. Due to a restriction imposed by a threshold number of (consecutive) errors allowed, their system does not guarantee the successful comparison of two files.

As well as reviewing other comparison systems, Heijink *et al.*[47] present a comparator that they claim copes with "extreme expressive timing" and "ornaments in a satisfactory way". Their approach is similar to that of Ukkonen, in that an error and the identity of its type are evaluated in isolation. Each error can be interpreted in a number of ways which leads to a number of alternative "paths" through the string. Each path represents different combinations of matches and errors. The path with the least number of errors (i.e. smallest edit distance) is deemed to be correct. Section 5.5.2.3 shows that if errors are processed in isolation then the performance distance will not be found.

Vercoe[93] and Dannenberg [21] are generally considered to be the pioneers of real-time computerised score performance matching. The system described by Dannenberg [20] is based on an offline matching system which uses similar techniques to that of Ukkonen. Instead of calculating an array of edit costs, Dannenberg populates an array with values that correspond to the Longest Common Substring (LCS) for any given position. Despite the use of LCSs, Dannenberg's system does not find the performance distance. Dannenberg's work is revisited in Section 5.6.6.

Müller and Mazzola[67] complain that a lack of progress beyond "*fuzzy common language descriptions*" has prevented a truly scientific approach to musicological performance theory. The problem of describing a musical concept was discussed in Chapter 4 of this thesis. They present a real-time system designed to track a performance and match it to a musical score. They tackle difficulties associated with a real-time approach but produce a system unsuitable for analysis when both the score and performance are known in full beforehand.

Pickens *et al.*[69] present a system for the retrieval of a musical score based on an audio query. Such systems are more closely related to work in the field of string searching[8, 92] rather than string matching.

The majority of systems referred to above are strongly influenced by a MIDI/musical approach. In its most basic form, the problem is ultimately one of string matching and will therefore be treated as such in the proposed system.

5.3 Proposed system

Comparisons can only be made between data of the same type. The abstraction of (stored) data from a given file type permits different levels of comparison. For example comparison of two wave files can only indicate whether the files are different or identical. Providing information on the actual differences in

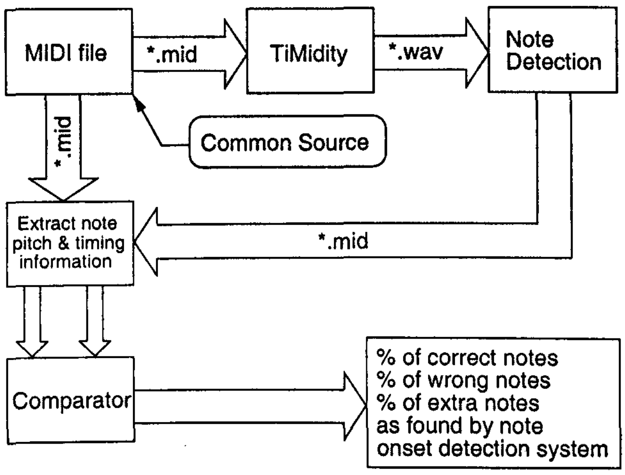


Figure 15: Proposed system overview

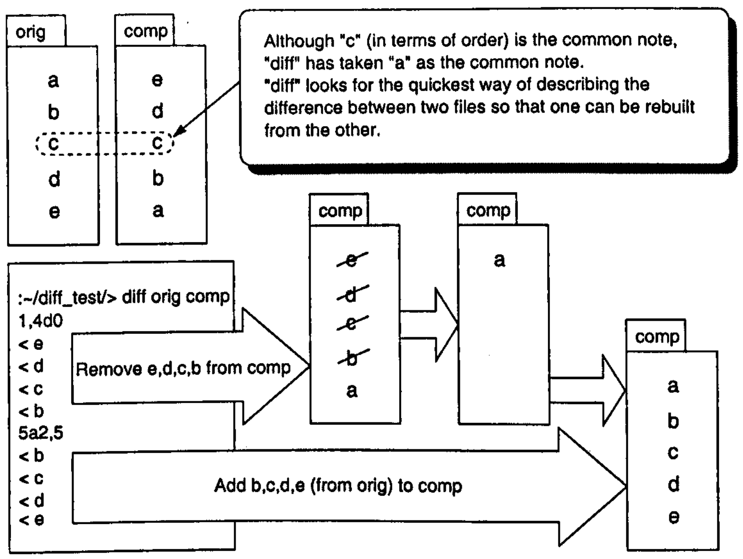
the binary data (of which there would be legion) would be of very limited use. Note detection systems usually output data which is different in type from that inputted. Figure 15 gives an overview of the processes used to overcome this problem of data incompatibility. Data type conversion must not introduce errors. This ensures that the comparator measures only the performance of the note detection system rather than that of the data type conversion process.

The MIDI file format has been chosen as the common source because it provides the system with the required note and timing information. The wealth of readily available MIDI software provides convenient means of generating test cases. The TiMidity[89] utility is a MIDI file interpreter which can be used to produce wave files. TiMidity is a well established utility and it is assumed that it performs error free MIDI to wave conversion. The wave file output of TiMidity is read by a note detection system which "listens" to the wave file and produces a MIDI file. The comparator takes the new MIDI file and compares it with the original MIDI file, providing a means of measuring the performance of the note detection system.

The heart of this system is the comparator, discussed in detail in section 5.5.

5.4 Diff

This section examines the unix utility "diff" to determine whether it can be used for comparison within a musical context. Although musical notes are the


orig
a
b
c
d
e
comp
e
d
c
b
a
Although "c" (in terms of order) is the common note, "diff" has taken "a" as the common note.
"diff" looks for the quickest way of describing the difference between two files so that one can be rebuilt from the other.
comp
~~e~~
~~d~~
~~c~~
~~b~~
a
comp
a
comp
a
b
c
d
e
Remove e,d,c,b from comp
Add b,c,d,e (from orig) to comp

```
:~/diff_test/> diff orig comp
1,4d0
< e
< d
< c
< b
5a2,5
< b
< c
< d
< e
```

Figure 16: The workings of "diff"

intended comparison material, this chapter will initially make use of the generic term "character" rather than "note".

The output from the unix file comparator "diff" is the difference between two files on a line by line basis. At first it would appear that "diff" could be used as the engine for character comparison. However, the output from "diff" only provides difference information such that one file can be modified so that it matches the other. The example in Figure 16 shows the output from "diff" is unsuitable for comparison of the the order of notes. The desired output would be (with respect to the *comp* file) to declare: "e" & "d" as wrong characters, "c" as a correct match and "b" & "a" as wrong characters.

Further problems with "diff" include:

- it only provides information concerning the difference between two files. In result validation the similarities are also important.
- it cannot use note timing information to provide a measure of the accuracy of a note in terms of onset and length.

For these reasons a bespoke comparator will be used, described in the next section.

5.5 The Comparator

5.5.1 System Overview

All discussion in this section ignores timing information, which is considered in Section 5.7.

5.5.1.1 Error types The three possible errors (defined with respect to the file that is being **compared**, e.g *comp* in Figure 16, with the **original**[12]) are:

1. Missing - e.g. *orig* = "abc", *comp* = "ac". Output = "a$\ominus$c"
2. Wrong - e.g. *orig* = "abc", *comp* = "awc". Output = "a$\otimes$c"
3. Extra - e.g. *orig* = "abc", *comp* = "abec". Output = "ab$\oplus$c"

The example given in item 3 above could be described as a wrong character 'e' immediately followed by an extra character 'c'. Two rules have been created to prevent such ambiguity when identifying errors:

1. The rule of maximum matches. It is assumed that the performer attempts to remain faithful to the score they are playing from. Therefore, errors are identified in a manner which gives benefit of doubt to the performer and ensures that the output preserves the original (particularly with respect to order) whenever possible. The wrong and extra character interpretation of item 3 (Extra error type) is rejected because it removes the letter 'c' from the output.
2. The rule of least number of errors. A given scenario must be described using the least number of single errors. Timing information could show that the wrong and extra character interpretation of item 3 is correct. Without such information, this description is rejected in favour of the interpretation which describes the situation in terms of a single error.

These rules lead to non-optimal edit distances, as shown in Section 5.5.2.3. By defininition, each error category is (in terms of a single error) mutually exclusive. Therefore, the "least number of errors" rule will always result in

[12]In musical terms:

- *orig* (the original) equates to a musical score;
- *comp* (that which is compared to the original) equates to a performance of a score.

a definite outcome. However, ambiguities can arise when describing multiple errors; it is possible to create a situation whereby the removal and insertion of characters cancel each other, possibly resulting in multiple wrong characters.

5.5.1.2 An ambiguous case

The errors in the following strings give rise to an ambiguous situation for which two different interpretations are equally valid:

```
orig = abcd
comp = acbd
```

Possible interpretations of the above include:

1. "a$\otimes\otimes$d" - one correct, two wrong followed by one correct.
2. "a$\oplus$b$\ominus$d" - one correct, an extra character 'c', one correct, a missing character 'c' and one correct.
3. "a$\ominus$c$\oplus$d" - one correct, a missing character 'b', one correct, an extra character 'b', and one correct.

Application of the rule of maximum matches eliminates the first interpretation as it only preserves 2 out of 4 characters from the original. The second and third interpretations are the inverse of each other, both preserving 3 out of 4 characters from the original. The decision as to which interpretation is correct is arbitrary.

5.5.1.3 Semantics

Whilst $\ominus\oplus$ and $\oplus\ominus$ are semantically different, both error combinations are the equivalent of $\otimes$. The rules described above dictate that $\otimes$ takes precedence over $\ominus\oplus$ and $\oplus\ominus$. The existence of $\otimes$ means that it is impossible for both $\oplus$ and $\ominus$ to consecutively appear in the same (multiple) error definition. In terms of semantics the only valid multiple error definitions are (quantities are assumed to start from zero):

- any number of $\otimes$
- any number of $\oplus$
- any number of $\ominus$
- any number of $\otimes$ immediately followed by any number of $\oplus$
- any number of $\otimes$ immediately followed by any number of $\ominus$

Errors are separated by correct (matching) characters.

5.5.2 The comparison function

Algorithm 1 shows the underlying functionality of the comparator. The requirement to find the performance distance, rather than the edit distance, results in an initial implementation that deviates from Ukkonen's shortest path through a matrix method. A dual mode recursive method is used so that the comparator takes subsequent errors and matches into account when evaluating a given error. This lookahead functionality is key to finding the performance distance.

5.5.2.1 The compare function This function works by recursively calling itself with the following arguments:

- orig - A pointer to the original string.
- comp - A pointer to the string being compared.
- mode - Determines how the function operates, discussed in detail below.
- iteration - Counts the number of recursions.

The function has two modes of operation, selected by the *mode* parameter, which alter the way matching characters are handled:

1. **Progressive mode.** The matching character is sent to the output stream when recursing. Reaching the end of a string terminates recursion.
2. **Lookahead mode.** Recursion takes place without outputting the matching character. Reaching the end of a string causes the number of correct matches found since entering this mode to be returned.

The input strings are compared character by character. If *orig* and *comp* point to matching characters the comparison function recursively calls itself with an

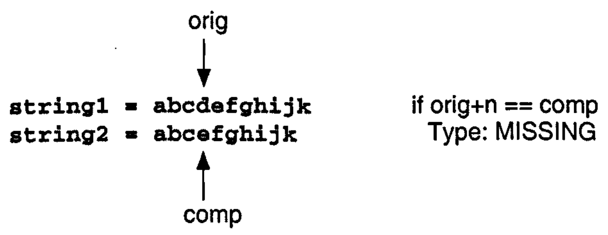


Figure 17: Missing character error

Algorithm 1 The recursive compare function

```
compare(char *orig, char *comp, int mode, int iteration)
if *orig && *comp then
  if *orig == *comp then
    if mode == 1 then
      output matching character
      compare(orig+1, comp+1,mode, iteration+1)
      return 0
    else
      compare(orig+1, comp+1,mode,iteration+1)
      return iteration
    end if
  end if
  check_for_missing(orig, comp,mode==1 ? 0 : iteration)
  check_for_extra(orig, comp,mode==1 ? 0 : iteration)
  if mode == 2 then
    if a List of error candidates exists then
      return winning_score
    else
      return iteration
    end if
  end if
  if mode == 1 && a List of error candidates exists then
    find winning error candidate
    output error code(s)
    return (compare(orig+winning_offset, comp+winning_offset, mode, it-
    eration+1))
  end if
  Deal with no match situation
  recurse with both strings offset by 1 - a wrong character
  return (compare(orig+1,comp+1,1,0))
end if
if mode ==2 then
  return iteration
else
  deal with any remaining characters
end if
return;
```

offset of 1 applied to both pointers. When the characters do not match the system determines the type of error that has occurred by calling the following functions:

- check_for_missing_character()
- check_for_extra_character()

The detection of wrong characters is inherently performed by both functions making a third, *check_for_wrong_character()* function, redundant (the justification for which is given in the next section).

5.5.2.2 Checking for Errors The error checking functions process the strings searching for a position that would realign the *orig* and/or *comp* pointers to matching characters. Figures 17 - 19 show how the error type is inferred from the realignment of each pointer. Algorithm 2 shows the workings of the error checking functions; the example shown searches for missing characters.

Algorithm 2 The recursive comparison function

```
find_missing_characters(char *orig, char *comp, int iteration)
offset = 0
while *(orig+offset) && *(comp+offset) do
  n = 0
  while *(orig+offset+n) && *(comp+offset) do
    if *(orig+offset+n) == *(comp+offset) then
      match[n] = compare((orig+offset+n), (comp+offset),2, iteration)
    else
      match[n] = 0
    end if
    n++
  end while
  find highest scoring match[n]
  add highest match to current recursion level's list of realignment points
  offset++
end while
```

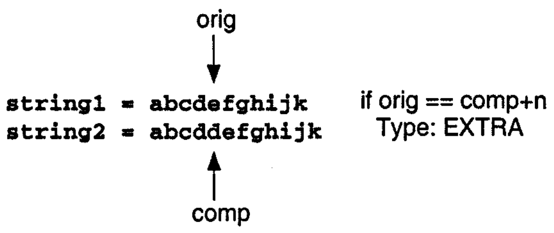


Figure 18: Extra character error

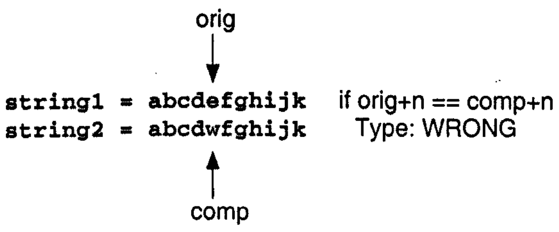


Figure 19: Wrong character error

When an error checking function finds a match it stores the realignment information in a structure which is added to a list[13]. The elements of this structure include:

- value
- orig index
- comp index
- offset
- type

The *value* attribute, a literal implementation of the rule of maximum matches, is an accumulative score of the subsequent matches of a given realignment. It is accumulative because it takes future errors into account by using the Lookahead mode of the compare function. The implementation of this lookahead feature ensures that the performance distance is found, rather than the edit distance. The difference between these two measures is:

[13]Each level of recursion maintains its own list.

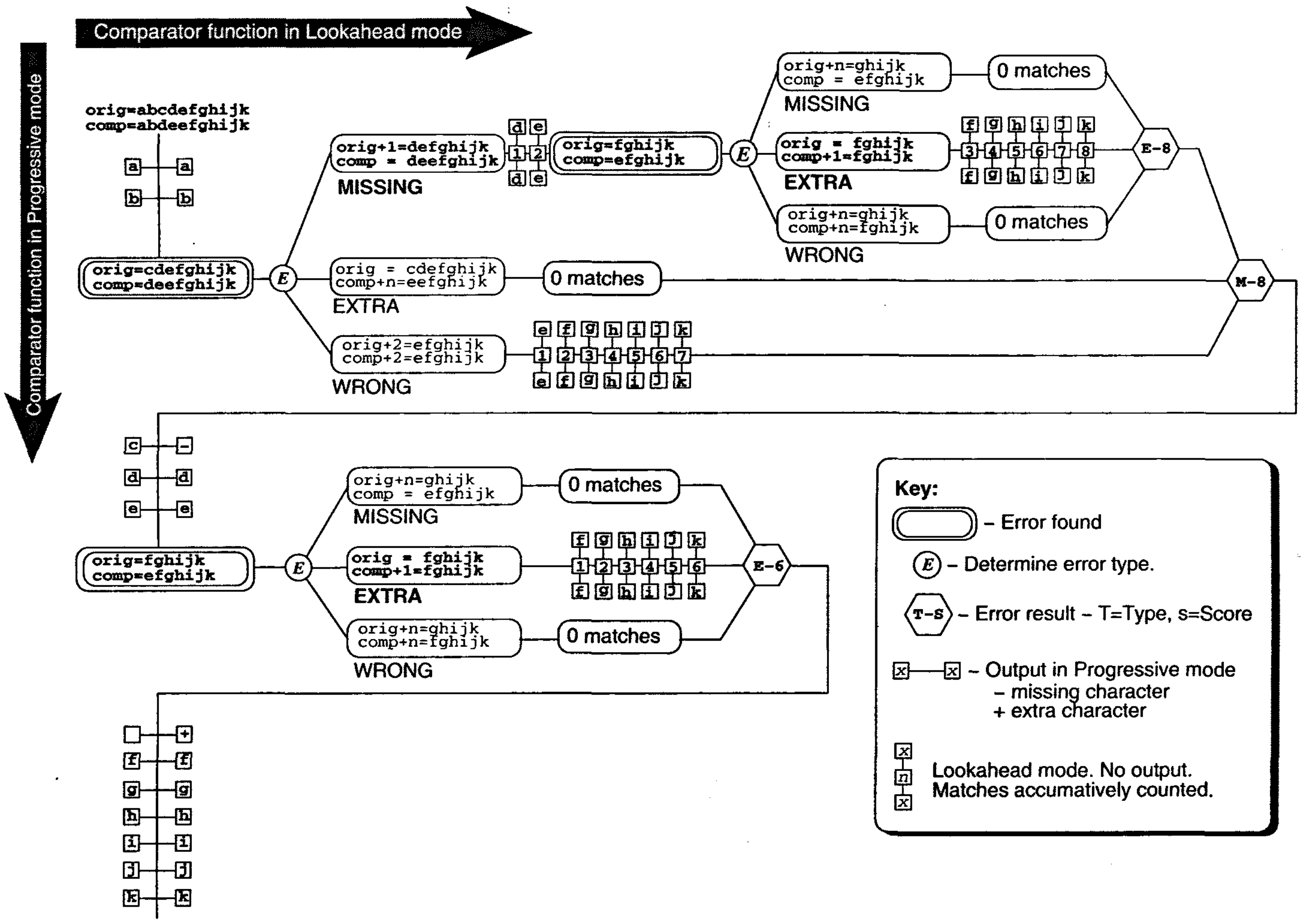


Figure 20: Showing how the comparison function is used in lookahead mode to evaluate the score of an error by taking future errors and matches into account.

- The minimum edit distance is determined by finding the most efficient way of describing the minimum number of changes (errors) that would convert string A into string B.
- The performance distance is determined by finding changes (errors) which provide the best means of indicating how much of string A is present in string B.

This is shown in Figure 20, in which the *comp* string contains two separate errors: a missing character and an extra character. However, due to both the position of the errors and the choice of extra character, this error could also be defined as 2 wrong characters. Application of the rule of least number of errors, described in section 5.5.1, shows that the correct interpretation is that of two separate errors (a missing character and an extra character, separated by two correct characters). As Figure 20 shows, the first error is successfully identified because the second error is taken into account when evaluating the *value* of the first. Thus, the correct interpretation preserves the letter 'd', showing it to be present in the performance of the score. In a musical context, this is effectively giving the benefit of doubt to the performer. It is assumed that the aim of performer is to correctly perform as many notes from the score as possible.

Failure to take into account the repercussion of error identification on subsequent matches is one of the downfalls of the algorithm presented by Bora *et al.*[7]. Their system judges a realignment position to be correct as long it and the next pair of characters in each string match. This, coupled with an unexplained "*if...then...else...*" ordering of error type checking (which imposes an order of precedence on error types), results in their system finding false matches. For example:

```
orig = abcbcde
comp = addbcde
```

It is obvious that "a⊗⊗bcde" is the correct result. Due to the ordering of error checking and the requirement that only two matches indicate successful identification of an error, the system presented by Bora *et al.* would find "a⊕⊕bc⊖⊖de"[14].

[14]This outcome is as a result of terminating error checking as soon as a realignment position that conforms to their matching criteria is found. In other words, the possibility of two wrong characters is not considered.

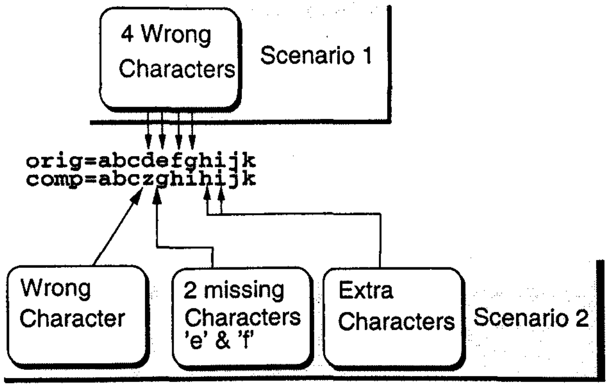


Figure 21: Combined errors.

5.5.2.3 Combined Errors The *offset* element of the realignment structure permits the detection of multiple errors. That is, a second error which occurs immediately after the first, as shown in Figure 21. The rules given in Section 5.5.1.1 dictate that scenario 2 is correct. It uses three individual errors to describe the first error[15] (rather than the four individual errors used in the first scenario) and preserves more of the original. This is an example of a non-optimal edit distance; the performance distance (5 errors) is greater than the edit distance (4 errors).

Thus far, searches for a realignment position have always been referenced from the error point. This fixed reference point can only lead to Scenario 1 of Figure 21. The problem lies in the fact that the error position characters ('d' in *orig* and 'z' in *comp*) never occur again in either string; the missing and extra error checks will never find a realignment position. However, if both the *comp* and *orig* pointers are offset by an index of n (by definition a wrong character, see Figure 19) and error checking resumes from this offset point, the *comp* pointer (pointing at 'g' for $n=1$) will find a realignment position which matches 2 subsequent characters. Thus the errors identified are:

- 1 wrong character 'z'
- 1 missing character 'e'
- 1 missing character 'f'

[15]One wrong character and two missing characters, the extra characters identify a later, separate error

Checking continues until the next error (two extra characters 'h' & 'i'). Thus the offset value indicates the number of wrong notes found. Due to the cancelling out of extra and missing characters (as explained in Section 5.5.1.3) it is only possible for wrong character errors to combine with another error type. Thus by increasing an offset from the initial error position, the missing and extra error checking functions also check for wrong characters making a dedicated wrong character function redundant, even in the case of a single wrong character.

Checking for a combination of errors is the key to ensuring that the performance distance is found. Combining errors allows the comparator to evaluate all valid error combinations. This functionality is missing from comparators that evaluate errors in isolation[16].

Using Figure 19 as an example, the missing and extra functions would both find a realignment position with the following values:

- value = 6
- orig index = 0
- comp index = 0
- offset = 1
- type = Missing or Extra

The *type* element is only used to determine which error character should be printed (it also indicates which function found the wrong character - useful for debugging purposes), rather than the actual type, which is inferred from the index and offset values.

For each error encountered the error checking functions build a (global) list of realignment positions. The realignment position with the highest value is chosen from this list as the realignment position which identifies the error. Using the highest value ensures the rule of maximum matches is adhered to.

If there is a draw between the values of different realignment positions, the position which is closest (i.e. lowest sum of *orig index*, *comp index* and *offset*) to the error point is chosen as the winner (implementing the rule of least number of errors). A number of situations can arise where there is no clear winner:

[16]There is a distinction to be drawn between *multiple errors* and *combined errors*. Multiple errors can appear in the the results of systems which process errors one by one, whereas combined errors are created speculatively by the comparator and then evaluated.

1. The two results are the same in terms of value, index and offset. By definition (due to the exclusive nature of an error) both results represent a wrong note or an ambiguous error (e.g. the situation shown in Section 5.5.1.2). In either case the choice of winner is arbitrary.
2. The two results are the same in terms of value and the sum of index and offset.

The following example illustrates the second draw situation:

```
orig = abcdefg
comp = accdefg
```

Inspection of the above leads to the conclusion that "a⊗cdefg" is the correct representation of the error. However, a consequence of the implementation of the rule of maximum matches (that future errors and matches are taken into account when calculating the value of an error), results in a second representation[17], "a⊖c⊕defg", which has the same value[18]. The least number of errors rule dictates that the first representation is correct. If a "value only" draw arises, the "wrong" error type takes precedence over "missing" and "extra" types. This is the only situation when one error type takes precedence over another.

The evaluation of error candidates, along with the ability to combine error types, enables the comparator to correctly identify errors and find the performance distance.

5.5.2.4 Special Cases The comparator uses characters which follow an error to determine its type. On its own, such an approach contains a fundamental flaw: the inability to cope with an error scenario characterised by both error checking functions returning zero. For example, this flaw would manifest itself when comparing strings of *equal* length when the error arises at the end of the string:

```
orig = abc
comp = abw
```

It would also occur when both strings contain no matching characters:

[17]This second representation is a result of using 'c' as the "wrong" error character which creates a seond "virtual" error (see Section 5.6.4). Any other character would result in a single unambiguous outcome.

[18]This result does not break the semantics of Section 5.5.1.3 as the errors are separated by a correct character.

```
orig = aa
comp = bb
```

Algorithm 1 shows that such a scenario is handled by recursively calling the comparison function with each string offset by 1. By definition this is a wrong character error (as explained in Section 5.5.2.3).

Another special case is the occurence of extra or missing characters at the end of strings which have unequal resolved lengths. The "resolved length" refers to a string length in which error types within the string are taken into account[19]. For example:

```
orig = abcd
comp = abeeecd
```

The comparator ceases recursion because the re-alignment caused by the detection of three extra characters results in the end of both strings being reached at the same time. The following strings are of unequal resolved length:

```
orig = abcde
comp = abeeecd
```

Recursion, and therefore comparison, ceases when the end of a string is reached. Termination conditions are applied when strings are of unequal resolved length. The nature of such operations is dictated by the shortest string. For example, if the end of *orig* is reached first, by definition *comp* contains n extra characters, where $n =$ the number of remaining characters in *comp*.

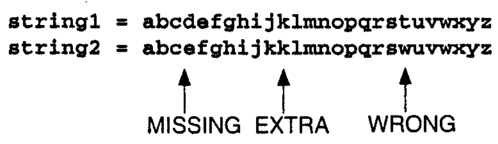
5.5.3 Comparator Tests

5.5.3.1 Test 1 Figures 23 - 25 show possible realignment positions displayed by a debugger and the position chosen as the winner for the three basic error types shown in Figure 22.

System output:

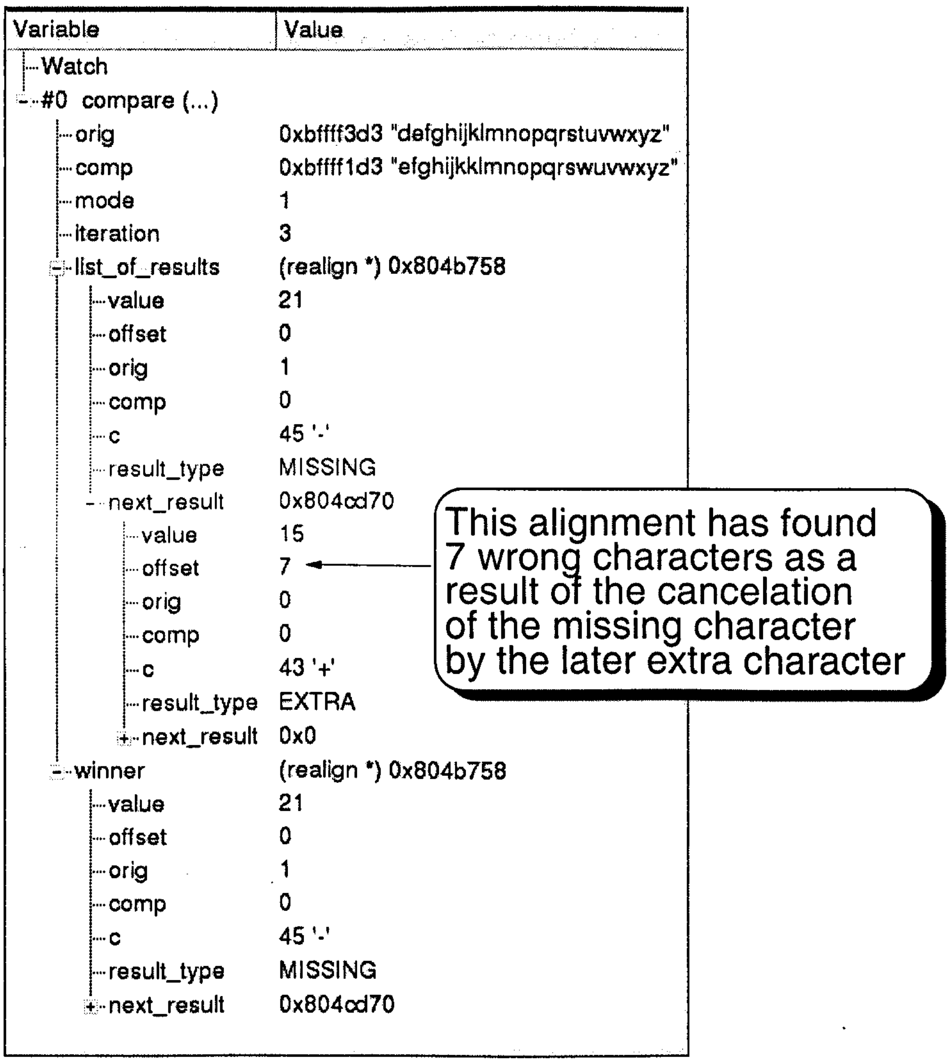
```
  orig = abcdefghijklmnopqrstuvwxyz
  comp = abcefghijkklmnopqrswuvwxyz
Output = abc⊖efghijk⊕lmnopqrs⊗uvwxyz
```

[19]In the example given the resolved length can be expressed either way. E.g. *orig* = ab⊕⊕⊕cd, or *comp* = abcd (extra chararecters removed). Both result in strings of equal resolved length. Note: the term "resolved length" does not necessarily imply the presence of errors in a string; it merely indicates that if errors are present they will be taken into account when measuring string length.



```
string1 = abcdefghijklmnopqrstuvwxyz
string2 = abcefghijkklmnopqrswuvwxyz
```

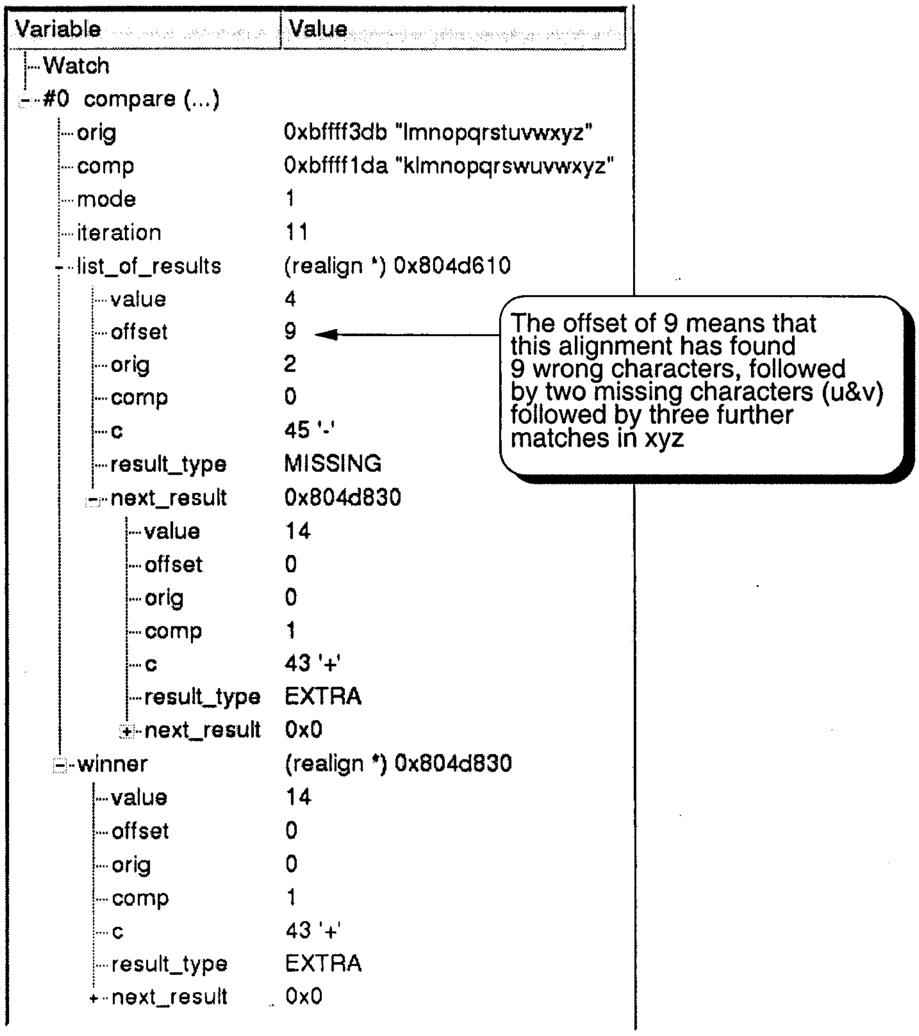
Figure 22: System test of the three basic error types.



```
Variable                Value
|--Watch
--#0 compare (...)
   |--orig              0xbffff3d3 "defghijklmnopqrstuvwxyz"
   |--comp              0xbffff1d3 "efghijkklmnopqrswuvwxyz"
   |--mode              1
   |--iteration         3
   --list_of_results    (realign *) 0x804b758
      |--value          21
      |--offset         0
      |--orig           1
      |--comp           0
      |--c              45 '-'
      |--result_type    MISSING
      --next_result     0x804cd70
         |--value       15
         |--offset      7
         |--orig        0
         |--comp        0
         |--c           43 '+'
         |--result_type EXTRA
         +--next_result 0x0
   --winner             (realign *) 0x804b758
      |--value          21
      |--offset         0
      |--orig           1
      |--comp           0
      |--c              45 '-'
      |--result_type    MISSING
      +--next_result    0x804cd70
```

This alignment has found 7 wrong characters as a result of the cancelation of the missing character by the later extra character

Figure 23: Showing the detection of a missing character.



```
Variable                 Value
|--Watch
--#0  compare (...)
   |--orig               0xbffff3db "lmnopqrstuvwxyz"
   |--comp               0xbffff1da "klmnopqrswuvwxyz"
   |--mode               1
   |--iteration          11
   --list_of_results     (realign *) 0x804d610
      |--value           4
      |--offset          9
      |--orig            2
      |--comp            0
      |--c               45 '-'
      |--result_type     MISSING
      --next_result      0x804d830
         |--value        14
         |--offset       0
         |--orig         0
         |--comp         1
         |--c            43 '+'
         |--result_type  EXTRA
         +-next_result   0x0
   --winner              (realign *) 0x804d830
      |--value           14
      |--offset          0
      |--orig            0
      |--comp            1
      |--c               43 '+'
      |--result_type     EXTRA
      +-next_result      0x0
```

The offset of 9 means that this alignment has found 9 wrong characters, followed by two missing characters (u&v) followed by three further matches in xyz

Figure 24: Showing the detection of an extra character.

5.5.3.2 Test 2 This test shows why the entire file is processed when looking ahead. If the lookahead functionality was restricted to an arbitrary value, it would prevent any consecutive errors which last longer than the lookahead value from being detected.

```
  orig = abcdefghijklmnopqrstuvwxyz
  comp = a7ze
Output = a⊗⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖z⊕
```

Application of the rule of maximum number of matches dictates that the above is the correct output as the order of the original has been preserved when parsing *comp*. Without this rule, there would be an argument for the following output:

```
Output = a⊗⊗⊖e⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖⊖
```

5.5.3.3 Test 3 This test relies on an offset applied to the initial error position, allowing it to detect the combined error, rather than 4 wrong characters (zghi).

```
  orig = abcdefghijklmnopqrstuvwxyz
  comp = abczghihijklmnnnnpqrstuvwxy
Output = abc⊗⊖⊖ghi⊕⊕jklmn⊗⊕⊕pqrstuvwxy⊖
```

5.5.3.4 Test 4 Choosing the wrong error has repercussions for all subsequent errors. The following test illustrates how future errors influence the choice of the type of the current error, helping to ensure that the correct error type is chosen. Chosing the error type which is the most successful in terms of subsequent matches ensures that the output matches the original string as closely as possible (made more obvious in the use of a simple sentence):

```
  orig = the cat sat on the mat
  comp = te catsatton on h rate
Output = t⊖e cat⊖sat⊕⊕⊕ on ⊖h⊖ ⊗at⊕
```

The correct result for this test case hinges on the choice of the "on" which represents extra characters. Had the system decided that the third 't' in "catsatton" corresponded with the space between "sat" and "on" in the first string, it would result in the following outcome:

```
  orig = the cat sat on the mat
  comp = te catsatton on h rate
Output = t⊖e cat⊖sat⊗on ⊗⊕⊕h⊖ ⊗at⊕
```

This alternative outcome fails to preserve the word/space structure of the original string, particularly around the second "the". The rule of maximum number of matches shows that the first outcome is correct as it preserves more of the original string in its output.

5.5.3.5 Test 5 - Ambiguous Error Definitions The test is included for the sake of completeness, to show that the comparator can cope with the ambiguous error definition described in Section 5.5.1.2.

```
  orig = abcd
  comp = acbd
Output = a⊖c⊕d
```

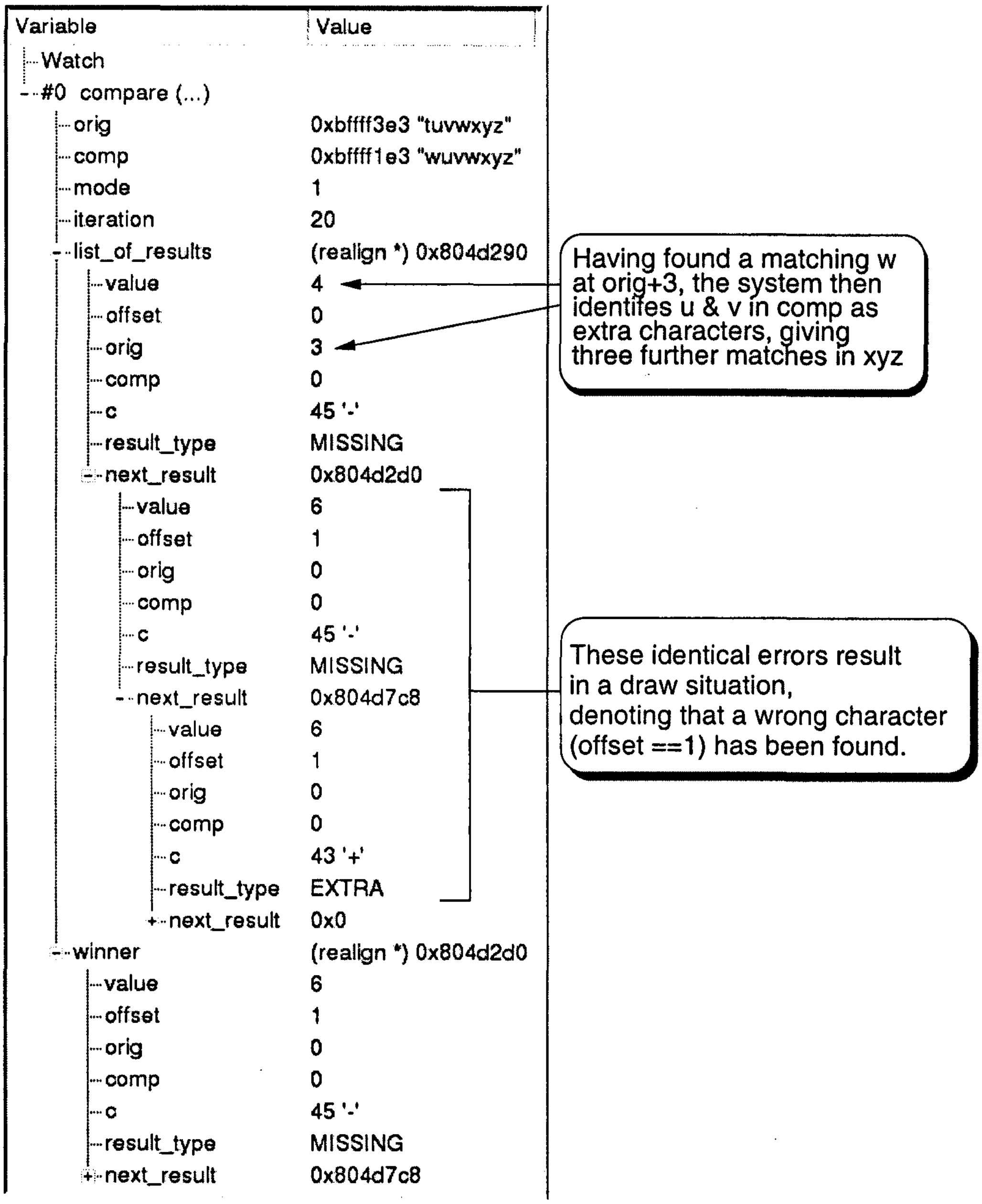
The choice of error definition in this example is arbitrary. This output has arisen because *check_for_missing_character()* is called before *check_for_extra_character()* and its realignment candidate appears first in the list of realignment candidates (in the event of a draw the default behaviour is to pick the first candidate from the list of drawing candidates). If the calls to these functions are reversed, the output is:

```
Output = a⊕b⊖d
```

The next test ensures that the comparator handles a drawn value situation:

```
  orig = abcdefg
  comp = accdefg
Output = a⊗cdefg
```

5.5.3.6 Test 6 - Special Cases These tests are included to show that the comparator handles the special cases described in Section 5.5.2.4:



```
Variable                     Value
|--Watch
--#0  compare (...)
    |--orig                  0xbffff3e3 "tuvwxyz"
    |--comp                  0xbffff1e3 "wuvwxyz"
    |--mode                  1
    |--iteration             20
    --list_of_results        (realign *) 0x804d290
        |--value             4
        |--offset            0
        |--orig              3
        |--comp              0
        |--c                 45 '-'
        |--result_type       MISSING
        --next_result        0x804d2d0
            |--value         6
            |--offset        1
            |--orig          0
            |--comp          0
            |--c             45 '-'
            |--result_type   MISSING
            --next_result    0x804d7c8
                |--value         6
                |--offset        1
                |--orig          0
                |--comp          0
                |--c             43 '+'
                |--result_type   EXTRA
                +-next_result    0x0
    --winner                 (realign *) 0x804d2d0
        |--value             6
        |--offset            1
        |--orig              0
        |--comp              0
        |--c                 45 '-'
        |--result_type       MISSING
        +-next_result        0x804d7c8
```

Having found a matching w at orig+3, the system then identifes u & v in comp as extra characters, giving three further matches in xyz

These identical errors result in a draw situation, denoting that a wrong character (offset ==1) has been found.

Figure 25: Showing the detection of a wrong character.

orig =
comp =
Output =
	orig = a
	comp =
	Output = ⊖
		orig =
		comp = bb
		Output = ⊕⊕

orig =
comp = b
Output = ⊕
	orig = aa
	comp =
	Output = ⊖⊖
		orig = a
		comp = b
		Output = ⊗

orig = aa
comp = bb
Output = ⊗⊗
	orig = aa
	comp = bbb
	Output = ⊗⊗⊕
		orig = aaa
		comp = bb
		Output = ⊗⊗⊖

5.6 The Cost of the Performance Distance

The genuine performance distance can only be found by performing exhaustive string comparisons. This section investigates the consequence of the requirement to find the performance distance and then looks at ways of improving efficiency.

The cost Γ of comparing two strings *orig* and *comp* of absolute length i and j respectively, which consist of characters from an alphabet Σ, containing ε errors, is measured in terms of the number of comparisons which take place. The efficiency of the comparator presented thus far is dependent on a number of variables. The impact each variable has on efficiency will be considered in this section.

5.6.1 String Length and Error Position

It is obvious that the longer the string, the greater the number of comparisons that will be required. If *comp* = *orig* then[20]:

$$\Gamma = i \quad for\, \varepsilon = 0$$

If there is an error in *comp* then the cost of comparison depends on the error position, as shown in Table[21] 1.

As has already been explained, for a given error the comparator searches the remaining length of both strings for realignment positions. It is the searching for, and subsequent evaluation of, realignment positions which incur the greatest cost. For example:

[20]The choice of i or j is arbitary because $\varepsilon = 0$ and *comp* = *orig*.

[21]This table was obtained through modification of the comparator so that it counted every comparison it made and displayed the final total.

	Error Position									
String Length	0	1	2	3	4	5	6	7	8	9
1	-	-	-	-	-	-	-	-	-	-
2	5	1	-	-	-	-	-	-	-	-
3	13	6	2	-	-	-	-	-	-	-
4	24	14	7	3	-	-	-	-	-	-
5	38	25	15	8	4	-	-	-	-	-
6	55	39	26	16	9	5	-	-	-	-
7	75	56	40	27	17	10	6	-	-	-
8	98	76	57	41	28	18	11	7	-	-
9	124	99	77	58	42	29	19	12	8	-
10	153	125	100	78	59	43	30	20	13	9

Table 1: The number of comparisons relation to string length and error position for a missing character.

```
orig = abcdefgh
comp = abcefgh
```

An error occurs at position[22] 3. For all allowed values of *offset*, a realignment match will occur at *orig*+*offset*+1 and *comp*+*offset*. For each match found, the comparator will look ahead through the string to determine how long a given match lasts:

offset	matches	substring length
0	efgh	4
1	fgh	3
2	gh	2
3	h	1
Total		10

This behaviour stems from the nested while loops in the error checking functions (see Algorithm 2). The Total of the above table is given by:

$$\frac{\alpha^2 + \alpha}{2} \tag{1}$$

[22]Error positions are:

- always defined in terms of the *orig* string.
- numbered from zero at the first character of the string.

Where α = the length of the substring of *orig* which matches an equivalent substring of *comp*. These *offset* matches occur when checking for a missing character. The following table shows the comparisons that take place when checking for a missing character:

Compare		Comparisons
e	defgh	5
f	efgh	4
g	fgh	3
h	gh	2
Total		14

There is a difference of 1 between each substring row for this and the previous table[23]. Thus the Total number of comparisons in the above table is given by:

$$\left(\frac{\alpha^2+\alpha}{2}\right)+\alpha \tag{2}$$

The number of comparisons taken by the check for an extra character function is shown in the following table[24]:

Compare		Comparisons
d	efgh	4
e	fgh	3
f	gh	2
g	h	1
Total		10

Thus the cost of checking for an extra character is provided by equation 1. Combining these three costs gives:

$$\left(\frac{\alpha^2+\alpha}{2}\right)+\left(\frac{\alpha^2+\alpha}{2}\right)+\alpha+\left(\frac{\alpha^2+\alpha}{2}\right)$$

$$=3\left(\frac{\alpha^2+\alpha}{2}\right)+\alpha \tag{3}$$

[23]Close examination of the two tables reveals an inefficiency in the implementation of the comparator. A match found in an error checking function is recursively passed to the main comparison function where the same check is re-performed. This second check is the beginning of the evaluation process which gives a given realignment position a score. The cost of this inefficiency is equivalent to the number of matching characters following an error and is therefore deemed neglible when compared to other costs.

[24]Note that the single character does not appear in the substring it is being compared to.

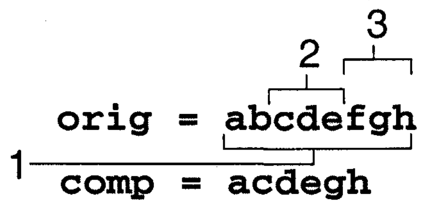


Figure 26: Breaking down the original string into sections.

By definition α represents the number of correct characters following an error. Adding the totals from each table to α $(10 + 14 + 10 + 4 = 38)$ gives the same total (see Table 1) as a string of length 5 with an error at position 0. The cost of comparing the characters which proceeded the error (in this case $\Gamma([a,b,c],[a,b,c]) = 3$) matches ε_p the error position. Thus the cost of comparing strings of any length with one missing character is:

$$= 3\left(\frac{\alpha^2 + \alpha}{2}\right) + 2\alpha + \varepsilon_p \tag{4}$$

Where $\alpha = i - \varepsilon_p - \varepsilon$ and ε_p= error position. The error position is limited to $0 \leq \varepsilon_p < i$.Thus as $\varepsilon_p \rightarrow i$, $\Gamma(orig, comp) \rightarrow (i-1)$ and as $i \rightarrow 0$, $\Gamma(orig, comp) \rightarrow (i-1)$. The constant "3" in equation 4 corresponds to the fact that there are three possible error types.

5.6.2 Number of Errors.

The cost of comparison is also dependent on the number of errors in the *comp* string. Equation 3 shows that the cost of comparison can be broken down into sections. The boundaries of each section are determined by error position(s). Figure 26 shows how a string is broken down into sections by its errors. The contribution each section makes is described below:

1. If the letter "f" is removed from *orig* in Figure 26, effectively shortening the string length by 1, it can be shown that

   (a) for $\alpha = i - \varepsilon - \varepsilon_0$ (where ε=no. of errors and ε_0 is the position of the first error):

$$\left(\frac{\alpha^2 + \alpha}{2}\right) + 2\alpha \tag{5}$$

gives the cost of searching for a missing character

(b) equation 1 gives the cost of checking for an extra character.

In the previous section the cost of the *offset* lookahead function was also given by equation 1, however in this case the lookahead function is interrupted by the second error, resulting in a false result from equation 1. The cost of the *offset* lookahead function is:

$$\left(\frac{\lambda^2+\lambda}{2}\right)+\left(\frac{\beta^2+\beta}{2}\right) \tag{6}$$

Where the number of characters between errors is given by $\lambda = \varepsilon_{p+1} - \varepsilon_p - 1$ (p = error number) and $\beta = i - \varepsilon_1$-1. Combining equations 1,5 and 6 gives:

$$\left(\frac{\alpha^2+\alpha}{2}\right)+2\alpha+\left(\frac{\alpha^2+\alpha}{2}\right)+\left(\frac{\lambda^2+\lambda}{2}\right)+\left(\frac{\beta^2+\beta}{2}\right)$$

$$=\alpha^2+3\alpha+\left(\frac{\lambda^2+\lambda}{2}\right)+\left(\frac{\beta^2+\beta}{2}\right) \tag{7}$$

This cost is essentially the base cost to which subsequent error costs are added.

2. This section serves a different purpose from that of sections 1 and 3. Its role is to determine by virtue of its length (λ in equation 7) how many times section 3 is evaluated when processing section 1.

3. This section is effectively a string of length $i - \varepsilon_1$ with an error at position 0. Its cost is therefore determined by equation 3.

Adding the three sections gives:

$$\alpha^2+3\alpha+\left(\frac{\lambda^2+\lambda}{2}\right)+\left(\frac{\beta^2+\beta}{2}\right)+(\lambda+1)\left(3\left(\frac{\beta^2+\beta}{2}\right)+\beta\right)+\kappa \tag{8}$$

where $\kappa = i - \varepsilon$. Thus the cost of comparison not only depends on string length, but also on the position of errors. The principle of breaking the string down into sections can be applied to calculate the cost of comparing strings containing any number of missing character errors.

However equation 8 is only true for $1 < \varepsilon_1 < (i-1)$ when $\lambda > 1$. For example:

```
orig = abcdefgh
comp = cdefgh
```

In this case comp has two missing characters. Temporarily ignoring the the letter "a" in *comp* results in comparison of strings of length 7 with an error at position zero. Hence equation 4 gives the cost of this temporary comparison. The additional cost of reinstating the letter "a" is that of comparing it with *comp*. Thus the cost of two consecutive errors is given by:

$$= 3\left(\frac{\alpha^2+\alpha}{2}\right) + 3\alpha + \varepsilon_0 \tag{9}$$

Where $\alpha = i - \varepsilon_0 - \varepsilon$. Equation 9 also covers all cases when the second error is at i-1

5.6.3 The Error type

Thus far the cost of comparison has only been considered in terms of a missing character error. This section shows the cost of different error types.

5.6.3.1 Extra Characters Errors The cost of comparing strings containing extra characters is effectively the same as comparing two strings with missing characters. Please consider the following strings:

```
orig = abcdefgh
comp = aabcdeefgh
```

It is obvious that for this error type the error position has to be defined in terms of the *comp* string. With respect to *comp*, *orig* can be described as missing two characters. Thus the cost of comparison is given by equation 8 having swapped *orig* and *comp*.

5.6.3.2 Wrong Character Errors The cost of comparing strings containing wrong characters is closely related to the cost of detecting other error types. It can be shown that the cost of comparing two strings with a single wrong character error is given by:

$$4\left(\frac{\alpha^2+\alpha}{2}\right) - \alpha + \varepsilon - 1 \tag{10}$$

The cost of finding two or more wrong character errors is calculated using the same prinicple shown in Section 5.6.2. The $\frac{n^2+n}{2}$ relationship remains unchanged; the difference is in the constants by which it is multiplied. The constant "4" in equation 10 corresponds to the fact that both error checking functions find wrong character matches as described in Section 5.5.2.3.

5.6.3.3 Combined Error Types

An immediate difficulty with combined error types is the definition of error position. The position of a missing character in *comp* is defined by where it appears in *orig* whereas an extra character in *comp* is defined in terms of where it appears in *comp*. For example:

```
orig = abcdefgh
comp = acdeefgh
```

This difficulty is overcome by using the resolved length of *comp*. Hence the error position at the extra character (the second "e") is 5 ("a+bcdee"). The same approach of breaking the string down into seperate sections would yield an an equation very similar to equation 8. The difference would lie in the constants used to multiply the $\frac{n^2+n}{2}$ relationship. This difference arises from the impact the second error has on the realignment of the last three characters of the string which, in terms of *offset,* provide many more realignment positions which must be evaluated. For example, the comparator makes 99 comparisons when comparing strings 8 characters long with two missing character errors at positions 1 and 5 respectively. It makes 173 comparisons if the second error is changed to an extra character.

5.6.4 Number of Repeated Error Characters.

Error characters are the two differing characters pointed at by *orig* and *comp* when an error occurs. Each subsequent re-use of an error character creates multiple realignment possibilites when determining the error type. The increased complexity of multiple error and character interactions makes it very difficult to derive a model of comparator behaviour. Instead comparator output will be used to determine the underlying relationship between string variables and comparator efficiency.

If it is assumed that for a given sequence of numbers σ, a relationship of the order σ^n exists, then repeated differentiation of σ will eventually yield a series consisting entirely of a constant. The number of differentiations taken to realise a constant yields the value of n and therefore the order of the underlying relationship between the input and comparator efficiency. Table 2 shows this process applied to the error position 1 column of Table 1.

String Length	Error Position 1	Difference	Difference
1	-	-	-
2	1	-	-
3	6	5	-
4	14	8	3
5	25	11	3
6	39	14	3
7	56	17	3
8	76	20	3
9	99	23	3
10	125	26	3

Table 2: Repeated differentiation of a series containing a relationship of the order σ^neventually yields a constant.

Orig	*Comp*	Comparisons	Diff	Diff	Diff
abc	ac	6	-	-	-
abcc	acc	17	11	-	-
abccc	accc	36	19	8	-
abcccc	acccc	65	29	10	2
abccccc	accccc	106	41	12	2
abcccccc	acccccc	161	55	14	2

Table 3: The number of comparisons for the repetition of one error character.

Repeated error characters increase the order of the underlying relationship between the input and the cost of comparison. Consider the following strings:

```
orig = abcccc
comp = acccc
```

The repetition of "c" in both strings results in both error detection functions finding multiple matches for a given offset. Table 3 shows that as the number of repeated error characters increases, the increase in the number of comparisons required is of the order σ^3.

In this example the function which checks for extra characters finds fewer realignment positions than the missing character function due to the absence of the letter "b" in *comp*. Table 4 shows that the effect of repeating both error characters increases the order of the underlying relationship between string length and number of comparisons to σ^4.

Orig	*Comp*	Comparisons	Diff	Diff	Diff	Diff
abcb	acb	15	-	-	-	-
abcbc	acbc	29	14	-	-	-
abcbcc	acbcc	66	37	23	-	-
abcbccc	acbccc	148	82	45	22	-
abcbcccc	acbcccc	303	155	73	28	6
abcbccccc	acbccccc	565	262	107	34	6
abcbcccccc	acbcccccc	974	409	147	40	6

Table 4: The number of comparisons for the repetition of both error characters.

The relevance of a test case is determined by context. The strings in Tables 3 and 4 are equally valid in generic and musical contexts. Moving into a formal language context exacerbates the efficiency problem[25]. Consider the following strings of length 45 characters, with a single missing character at error position = 11:

```
orig = This is a sentence that may have many repeats
comp = This is a sntence that may have many repeats
```

It takes 26,698,407 comparisons to compare these strings. This is due to virtual errors which are created (when the comparator is in "lookahead" mode) by repeated error characters. Figure 27 shows the creation of virtual errors.

The example in Section 5.6.2 showed how the second error type is repeatedly evaluated when evaluating the first error. This repetitious evaluation also occurs when evaluating virtual errors. As the above example shows, such repetition is hugely inefficient. Efficiency can be improved by caching the error type of later errors, as described by Reingold[74], and using cached results rather than recalculating the error type.

5.6.5 Cached Results

Section 5.6.1 showed how comparison strings can be broken into substrings. The recursive nature of the comparator means that once the error type that caused the last substring has been found, this result will always be true and can be reused. Thus the position of the (virtual) error with respect to both *orig* and

[25]The example is given only for interest as a means of introducing the next section on cached results. The comparator has been designed specifically for the comparison of music and as such is not suited for comparing formal language sentences.

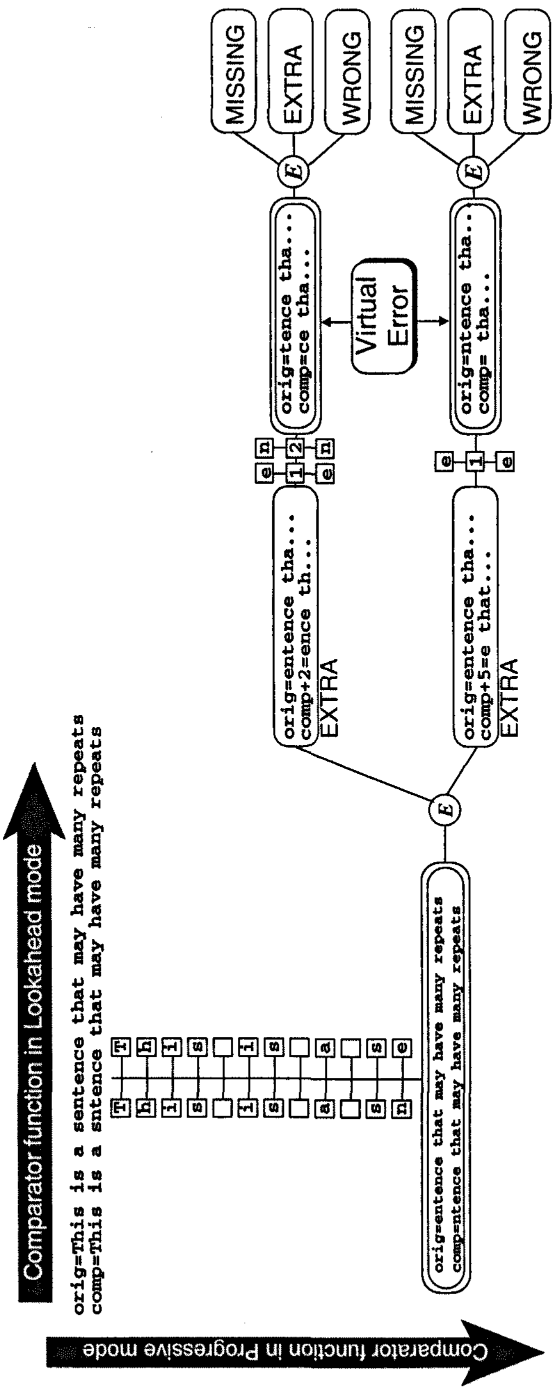

Comparator function in Lookahead mode
Comparator function in Progressive mode
orig=This is a sentence that may have many repeats
comp=This is a sntence that may have many repeats
orig=entence that may have many repeats
comp=ntence that may have many repeats
E
orig=entence tha...
comp+2=ence th...
EXTRA
orig=entence tha...
comp+5=e that...
EXTRA
orig=tence tha...
comp=ce tha...
orig=ntence tha...
comp= tha...
Virtual Error
MISSING
EXTRA
WRONG
MISSING
EXTRA
WRONG

Figure 27: A non-exhaustive example of the creation of virtual errors.

Orig	*Comp*	Comparisons	Cached Comparisons
abcdefghij	adefghij	106	106
abcdefghij	acefghij	238	169
abcdefghij	acdfghij	246	146
abcdefghij	acdeghij	230	128
abcdefghij	acdefhij	199	115
abcdefghij	acdefgij	162	107
abcdefghij	acdefghj	128	104
abcdefghij	acdefghi	106	106

Table 5: Showing the difference in cached and non-cached results for string length =10 and 2 missing characters (shown in bold in *Orig*) at error positions 1,n where $1 > n \leq j$.

comp can be used to reference a cache of error type[26]. If the cache for a given error position is not empty then the error type has previously been identified and the cached error type is used[27].

The multiplication of equation 3 by λ in equation 8 corresponds to the repeated evaluation of the second error. When cached results are used the equation for the number of comparisons taken to compare two strings containing two missing characters is reduced to:

$$\alpha^2 + 3\alpha + \left(\frac{\lambda^2 + \lambda}{2}\right) + 2\beta^2 + 3\beta + \kappa \tag{11}$$

Table 5 shows the difference that cached results make to the number of comparisons required when comparing strings containing two missing characters.

Section 5.6.4 showed that the underlying relationship between string length and the number of comparisons was of the order σ^4 when both error characters were repeated. Differentiation of the series shown in Table 6 reveals that whilst the underlying relationship is still of the order σ^4, the use of cached results has reduced the constant by which this relationship is multiplied. A comparator

[26]An error (virtual or real) can interact with (and create) future (virtual) errors. Thus a cached error type depends not only on its position with respect to *orig*, but also with respect to which character in *comp* caused the (virtual) error.

[27]The implementation of a cache of error types results in slight changes to Algorithms 1 and 2. Apart form the obvious addition of code to store and use the cached error results, a more subtle change is in the process by which the score of an error is accumulated. Previously the iteration was passed to and fro between the compare and error checking functions, as a means of accumulating the score of an error. Caching of results removes the need to pass the iteration to the error checking functions, which in turn removes the mode check when calling the error checking functions.

which takes advantage of cached results uses 31,911 comparisons to compare the sentences in Section 5.6.4. For this example the comparator is just over 836 times more efficient when cached results are used.

The consequence of the requirement to find the performance distance when comparing two strings is that, following an error, the remainder of the string must be exhaustively compared. These comparisons are not in themselves inefficient as they are a system requirement. It is the repetition of the same comparisons that is inefficient and is therefore overcome by the use of cached information.

Speed of comparison could be further increased by dynamically limiting the range of comparator when in "lookahead" mode, which would reduce the creation of virtual errors. This however would compromise the ability of the comparator to find the actual performance distance. Such restrictions would also require the introduction of a limit on the amount by which the *offset* parameter could be increased, otherwise errrors would be incorrectly identified.

5.6.6 The Matrix approach

This section is included as proof of concept and for the sake of completeness. The work presented in this section was realised as a result of the work presented in this entire chapter[28]. The implementation of the recursive comparator gave insights that led to the realisation of a matrix implementation.

Of all the comparison systems reviewed, Dannenberg's offline method (upon which his real-time method was based) came closest to finding the performance distance. However, a path finding rule which consistently selected a "perfor-

[28]In other words, the matrix comparator was only realised (as a proof of concept) following the full implementation, testing and use of the recursive comparator.

Orig	*Comp*	Comparisons	Comparisons Using Cached Results
abcb	acb	15	15
abcbc	acbc	29	29
abcbcc	acbcc	66	60
abcbccc	acbccc	148	129
abcbcccc	acbcccc	303	262
abcbccccc	acbccccc	565	491

Table 6: Showing the difference in number of comparisons for the repetition of one error character when cached results are used.

mance distance" path through the matrix for all test cases, could not be found. Errors arose because of the limited information stored in the matrix. The matrix did not contain information which would allow a path to be chosen (and therefore an error type to be determined) based on the consequence such a choice would have on future errors. Furthermore, the matrix contained no information regarding the number of subsequent matches a given error type would produce.

For a matrix M of size o, c (where o and c are the lengths of orig and comp respectively), Dannenberg follows the normal procedure of finding an optimum path by starting at $M_{o,c}$ and tracking backwards to $M_{0,0}$. He populates his matrix row by row, starting at $M_{0,0}$ calculating the LCS[29] for given string positions. This results in a matrix of information regarding sub-strings that have already been processed. It has been shown in previous sections of this chapter that correct error identification relies on choosing an error type such that it creates the LCS from the remaining, unprocessed, sub-string.

Reversing Dannenberg's matrix populating method by starting at $M_{o,c}$ gives a matrix of Future Longest Common Substrings (FLCS). An example matrix for the strings used Section 5.5.2.3 is shown in Figure 28. The performance distance's path is found by starting at $M_{0,0}$ and obeying the following rules (for O = orig string and C = Comp string and $o = 0, c = 0$):

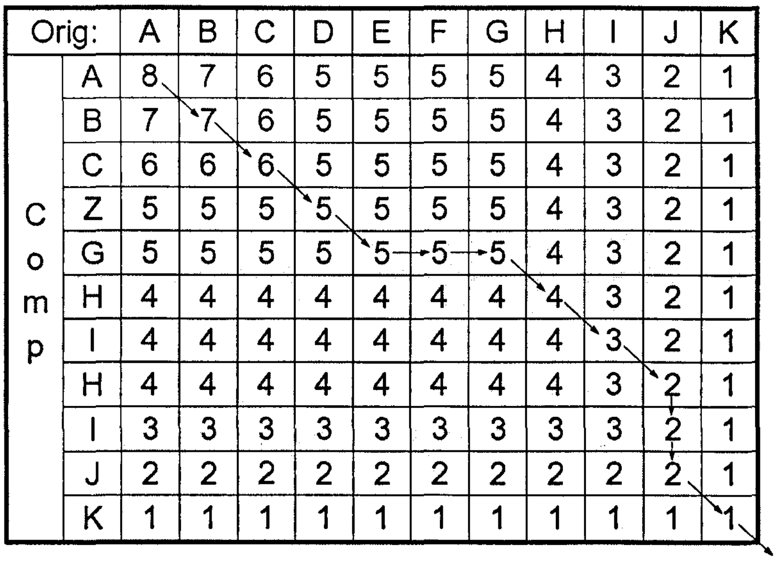
- If $O_o = C_c$ **then** o++, c++, **else**

    1. if $M_{o,c+1} = M_{o+1,c+1} = M_{o+1,c}$ **then** o++, c++, **else**
    2. if $M_{o,c+1} > M_{o+1,c+1}$ and $M_{o,c+1} > M_{o+1,c}$ **then** c++, **else**
    3. if $M_{o+1,c} > M_{o+1,c+1}$ and $M_{o+1,c} > M_{o,c+1}$ **then** o++, **else**
    4. if $M_{o,c+1} = M_{o+1,c}$ **then** o++ or c++

Numbered items 1-3 represent the identification of wrong, extra and missing error types respectively. Item 4 is a special case for catching ambiguous errors (see Section 5.5.1.2) where the choice between a missing or extra error is arbitrary. Each arrow on Figure 28 represents system output that is determined by the above path finding rules.

Tests show that the matrix implementation acheives results that are identical to the recursive implementation. The matrix implementation is obviously far more efficient than the recursive implementation. For example, the matrix implementation uses 90 comparisons to complete the final test of Table 6. The

[29]Longest Common Substring



Orig:		A	B	C	D	E	F	G	H	I	J	K
	A	8	7	6	5	5	5	5	4	3	2	1
	B	7	7	6	5	5	5	5	4	3	2	1
	C	6	6	6	5	5	5	5	4	3	2	1
C	Z	5	5	5	5	5	5	5	4	3	2	1
o	G	5	5	5	5	5	5	5	4	3	2	1
m	H	4	4	4	4	4	4	4	4	3	2	1
p	I	4	4	4	4	4	4	4	4	3	2	1
	H	4	4	4	4	4	4	4	4	3	2	1
	I	3	3	3	3	3	3	3	3	3	2	1
	J	2	2	2	2	2	2	2	2	2	2	1
	K	1	1	1	1	1	1	1	1	1	1	1

Figure 28: Showing the performance distance path through a FLCS matrix.

sentences in Section 5.6.4 require only 2024 comparisons. The cost of comparison is now $(o \times c) + min\,(o, c)$.

Section 5.6.2 shows that correct matches before the first error have no effect on subsequent errors and matches. This fact can be exploited by comparing strings until an error is found, prior to the creation of the matrix. The first error would mark the start of substrings which would be used to create and populate a FLCS matrix. The implentation of such a method reduced the number of comparisons need to compare the sentences in Section 5.6.4 to 1155. The cost of comparison is now dependent on the first error position and is given by $((o - \varepsilon_0) \times (c - \varepsilon_0)) + min\,(o, c)$.

The efficiency of the matrix implementation could be improved further by:

- investigating whether the path finding rules can be optimised to eliminate the need for string comparison when finding the performance distance path.

- implementing an optimised matrix populating routine as described by Ukkonen. The matrix population routine is inefficient as the majority of matrix values calculated are never used.

This implementation obeys the rules of least number of errors and maximum number of matches. It also adheres to the semantic rules laid down in Section 5.5.1.3. This matrix comparator and the recursive comparator are therefore identical in terms of function and differ only in terms of implementation.

5.7 Timing Information

As stated in the introduction to this chapter, the accuracy of timing information is a useful measure of the performance of a note onset detection system. This section considers the comparison of notes and their associated timings, rather than just characters which have no function of time other than that implied by the order of non-repeated characters. Without the use of timing information it is impossible to properly compare a list of repeated notes[30].

5.7.1 Onset times

The introduction of timing information could lead to a fundamental change in the implementation of the comparator. That is a move from a comparator based on character postion, to one based on numerical values. Please consider the following notes:

```
Original notes = cdefgABC
Comparison notes = cddeffABC
```

In terms of order it would appear that the second "d" is an extra note and the second "f" is a wrong note (corresponds with "g" from the original notes). However Figure 29 shows that when timing information is taken into account, the second "f" is an extra note and "g" is missing from the Comparison notes.

The example in Figure 29 uses convenient round numbers producing exact matches between the original and comparison notes. In reality, the timing values of the orignal and comparison notes would not necessarily be exact round numbers. This would mean that in order for the comparator to find mathces in terms of time, two notes would have to occur within a certain time frame. Difficulties would arise when trying to establish the limits of the frame. For example, if the frame was too wide a series of fast notes could fall within the catchment region of one original note. Making the frame too narrow would result in a matching pitch with poor timing being identified as an extra note and the note it matched (in terms of pitch) marked as missing.

[30]For example comparison of "abcd" and "acd" yields "a⊖cd", whereas comparison of "aaaa" and "aaa" yields "aaa⊖". Only timing information can determine the genuine position of the missing character.

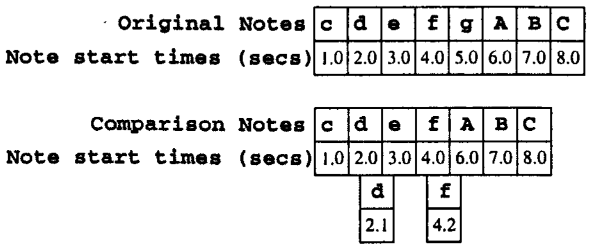


Figure 29: Notes with timing information.

Rather than using the timing information for note matching purposes it is better to use the information as a measure of accuracy once a match has been found, using both position and pitch. A numerically based comparator would also lack the ability to perform comparison solely in terms of note pitch. Such functionality is required when note timing information is either not available (e.g. when testing the performance of a pitch tracking system which does not give any timing output), or to be ignored to enable the pitch tracking ability of a system to be tested without interference from (possibly inaccurate) timing information. It is noted in passing that a numerical (i.e. time based) comparator can only detect missing and extra notes. A further check (using pitch information) would be required in order to detect wrong notes. The conclusion drawn is that the comparator should use both timing and pitch information, but retain the ability to perform comparisons solely in terms of pitch.

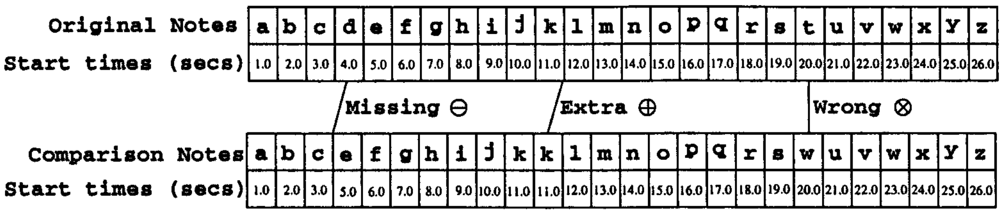
The existing position based comparator was modified to use both timing and note information when performing comparisons. The main modifications were to:

1. Enable calculation of the accuracy of timing information for matching characters.

2. Influence the result of a match if the matching note times differ by more than half a second.

3. Change the "lookahead" capability of the error checking functions to take timing information into account.

The following tests, based on the first two tests in Section 5.5.3, show how timing information influences the output of the comparator.



Original Notes	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Start times (secs)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0	21.0	22.0	23.0	24.0	25.0	26.0

Comparison Notes	a	b	c	e	f	g	h	i	j	k	k	l	m	n	o	p	q	r	s	w	u	v	w	x	y	z
Start times (secs)	1.0	2.0	3.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0	21.0	22.0	23.0	24.0	25.0	26.0

Figure 30: Test one using timing information.

5.7.1.1 Test 1 Figure 30 shows the input strings and their timings and highlights where the errors occur. The table below shows the output from the comparator, with errors shown in bold. The timing column only shows the accuracy of timing for correct notes. A value of zero indicates an exact match.

No.	Orig	Comp	Output	Timing	No.	Orig	Comp	Output	Timing
1	a	a	a	0.00	16	n	n	n	0.00
2	b	b	b	0.00	17	o	o	o	0.00
3	c	c	c	0.00	18	p	p	p	0.00
4	**d**		**⊖**		19	q	q	q	0.00
5	e	e	e	0.00	20	r	r	r	0.00
6	f	f	f	0.00	21	s	s	s	0.00
7	g	g	g	0.00	**22**	**t**	**w**	**⊗**	
8	h	h	h	0.00	23	u	u	u	0.00
9	i	i	i	0.00	24	v	v	v	0.00
10	j	j	j	0.00	25	w	w	w	0.00
11	k	k	k	0.00	26	x	x	x	0.00
12		**k**	**⊕**		27	y	y	y	0.00
13	l	l	l	0.00	28	z	z	z	0.00
14	m	m	m	0.00					

As expected the comparator successfully identified correct notes as being in time. The timing of the certain Comparison notes in Figure 30 were modified as followed:

- "c" - Changed from 3.0 to 4.0
- the first "k" - Changed from 11.0 to 10.3
- the second "k" - Changed from 11.2 to 11.0

- "w" - Changed from 20.0 to 20.7

The Table below shows the new outcome as a result of these changes:

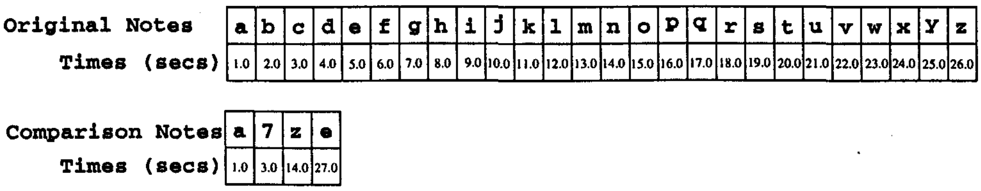
No.	Orig	Comp	Output	Timing	No.	Orig	Comp	Output	Timing
1	a	a	a	0.00	16	n	n	n	0.00
2	b	b	b	0.00	17	o	o	o	0.00
3	c		⊖		18	p	p	p	0.00
4	d	c	⊗		19	q	q	q	0.00
5	e	e	e	0.00	20	r	r	r	0.00
6	f	f	f	0.00	21	s	s	s	0.00
7	g	g	g	0.00	22	t		⊖	
8	h	h	h	0.00	23		w	⊕	
9	i	i	i	0.00	24	u	u	u	0.00
10	j	j	j	0.00	25	v	v	v	0.00
11		k	⊕		26	w	w	w	0.00
12	k	k	k	0.00	27	x	x	x	0.00
13	l	l	l	0.00	28	y	y	y	0.00
14	m	m	m	0.00	29	z	z	z	0.00

The comparator has taken these new timings into account and has produced an output which violates the semantics of Section 5.5.1.3. This is permissable however as the error definition semantics only apply when timing information is not available. Comparison of rows 11 and 12 with the previous table shows how timing information has effected the output revealing that the extra note was played before the correct note, rather than after it.

5.7.1.2 Test 2

Figure 31 shows the input strings and their timings

This test shows how timing information influences the outcome of a match:



Original Notes	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Times (secs)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0	21.0	22.0	23.0	24.0	25.0	26.0

Comparison Notes	a	7	z	e
Times (secs)	1.0	3.0	14.0	27.0

Figure 31: Using timing information

No.	Orig	Comp	Output	Timing	No.	Orig	Comp	Output	Timing
1	a	a	a	0.00	16	o		⊖	
2	b		⊖		17	p		⊖	
3	c	7	⊗		18	q		⊖	
4	d		⊖		19	r		⊖	
5	e		⊖		20	s		⊖	
6	f		⊖		21	t		⊖	
7	g		⊖		22	u		⊖	
8	h		⊖		23	v		⊖	
9	i		⊖		24	w		⊖	
10	j		⊖		25	x		⊖	
11	k		⊖		26	y		⊖	
12	l		⊖		27	z		⊖	
13	m		⊖		28		e	⊕	
14	n	z	⊗						

The timing information has influenced the position of the "z" so that a wrong note match would be found.

5.7.1.3 Test 3 This test shows the system reporting on the accuracy of note onset times with respect to the timing of the orignal. Figure 32 shows the test case which produced the following output:

Original Notes	a	b	c	d	e	f	g
Onset Time (secs)	1.0	2.0	3.0	4.0	5.0	6.0	7.0

Comparison Notes	a	b	c	d	e	f	g
Onset Time (secs)	1.0	2.1	2.9	4.23	5.0	6.0	7.2

Figure 32: Inaccurate notes used for Test Three

Original Notes	a	b	c	d	e	f	g
Onset Time (secs)	1.0	2.0	3.0	4.0	5.0	6.0	7.0
Note Length (secs)	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Comparison Notes	a	b	w	d	e	f	g
Onset Time (secs)	1.0	2.0	3.0	4.0	5.0	6.0	7.0
Note Length (secs)	1.0	1.0	1.0	1.0	1.2	0.7	1.0

Figure 33: Test case with note lengths.

Original Notes	Comparison Notes	Output	Timing
a	a	a	0.00
b	b	b	0.1
c	c	c	-0.1
d	d	d	0.23
e	e	e	0.00
f	f	f	0.00
g	g	g	0.2

Positive or negative times indicate late or early notes respectively. The comparator has worked as expected and has provided a measure of note start time accuracy.

5.7.2 Release Times

Another aspect of music transcription is the detection of the length of a given note. The compare function was modified so that the length of correct matches are compared and a measure of note length accuracy is given.

Figure 33 shows the test case used to produce the output shown below:

Orig	Comp	Output	Timing	Length
a	a	a	0.00	0.00
b	b	b	0.00	0.00
c	w	⊗		
d	d	d	0.00	0.00
e	e	e	0.00	0.20
f	f	f	0.00	-0.30
g	g	g	0.00	0.00

The system behaves correctly in that it:

- ignores the length of the incorrect note
- shows note "e" is 0.2 seconds longer than its counterpart in the original
- shows note "f" is 0.3 seconds shorter than the original

5.8 The Testing of Cirotteau's PitchTracker

Cirotteau's[15] real-time PitchTracker system uses phase vocoder techniques to extract partials from a signal. A maximum-likelihood function is used to determine which partial corresponds to the fundamental frequency of the signal. A tunable stability threshold is used to determine whether the detected fundamental corresponds to a note. If a fundamental is detected x times then a MIDI note-on event is recorded. A more detailed explanation of the PitchTracker can be found in Appendix A.

The ten system tests used in the following sub-sections have been designed to test different aspects of note detection systems. MIDI files were generated by a modifed[31] version of Günter's MIDI Compiler (GMC)[86]. This MIDI file is the common start point defined in Figure 15. TiMidity was used to generate the wave files which were analysed by Cirotteau's PitchTracker system, which in turn produces a MIDI file of what it "hears"[32]. A MIDI data extraction utility parses the MIDI files and creates an output file with the following stucture:

- A list of notes
- A list of note onset times
- A list of how long each note lasts

The comparator uses available data from the above above structure to compare the original and wave file derived MIDI files. TiMidity was configured to use a piano sound (which allows the decay of each note to overlap with the next note). For each test case the MIDI file output of GMC is shown as a musical score to give a good representation of the information the PitchTracker was presented with.

[31]The modified software generates minimalist midi files, making the implementation of the midi data extraction utility much easier.

[32]The wave files generated by TiMidity had to be amplified by -24dB. Without this amplification the PitchTracker was unable to detect the notes in the generated files.

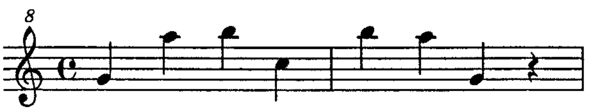
The PitchTracker was tuned so that it was able to detect and register the fundamental frequencies of notes in the tests. The tests were performed three times; one run for each of the available FFT bin resolutions. Apart from Test Three, the detailed results that follow are all from tests performed with 512 bins. The results of test three come from a test performed with 1024 bins.

The PitchTracker does not provide timing information other than that implied by note order. Comparisons will therefore be based on note order.

5.8.1 Test One

This test consists of seven notes of 1 second duration, four of which are in a very high register (note the octave marking on the score).

Original MIDI Score, notated by midi2ly[33]:



Comparator Output - Test One		
Orig	Comp	Output
O	#	⊗
]	]	]
_		⊖
T	T	T
_		⊖
]	]	]
O	O	O

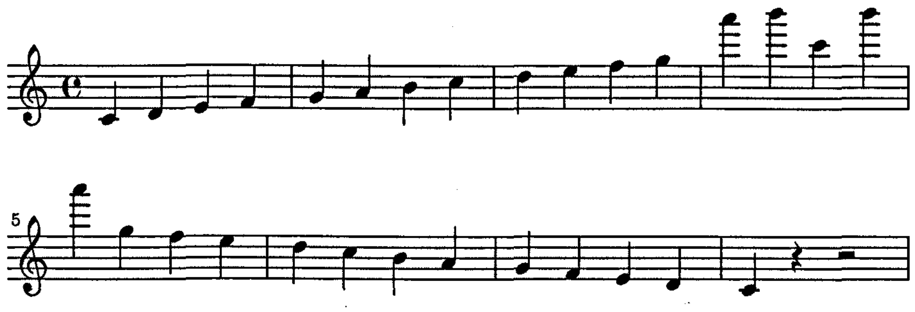
The characters in the above table[34] reflect the fact that the system is now comparing actual MIDI note data[35]. The ASCII pitch note values used all fall within the printable range of characters. The comparator provides the following summary:

```
Notes found:
Total Correct   4   57.14%
Total wrong     1   14.29%
Total Missing   2   28.57%
Total extra     0   0.00%
```

[33]Midi2ly is a utility that comes with the lilypond[1] musical typesetting software used througout this thesis for all musical excerpts.
[34]This applies to all results tables in Section 5.8.
[35]The pitch byte is used for comparison purposes.

5.8.2 Test Two

This test is based on a two octave scale (29 notes) with the note length set to 1 second. Original Score:



As can be seen from the score, the sixth and seventh notes of the second octave are an octave higher than normal. Such notes would be described as unexpected by a human listener, as they will assume they are listening to a scale. The change in octave would lead to the human questioning their original assumption. The inclusion of such high notes tests the ability of the PitchTracker to detect notes which are short due to their high register.

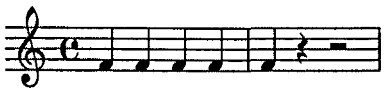
Comparator Output - Test Two							
No.	Orig	Comp	Output	No.	Orig	Comp	Output
1	<	<	<	16	_		⊖
2	>	>	>	17	]	]	]
3	@	@	@	18	O	O	O
4	A	A	A	19	M		⊖
5	C	C	C	20	L	L	L
6	E	E	E	21	J	J	J
7	G	G	G	22	H	H	H
8	H	H	H	23	G	G	G
9	J	J	J	24	E	E	E
10	L	L	L	25	C	C	C
11	M		⊖	26	A	A	A
12	O	O	O	27	@	@	@
13	]	]	]	28	>	>	>
14	_		⊖	29	<	<	<
15	T	T	T	-	-	-	-

As the above results show, the PitchTracker missed two of the "high" notes. Comparator Summary:

```
Notes found:
Total Correct  25  86.21%
Total wrong     0  0.00%
Total Missing   4  13.79%
Total extra     0  0.00%
```

5.8.3 Test Three

This test consists of five repeated notes, at the same pitch, of 1 second duration. Original Score:



Comparator Output - Test Three		
Orig	Comp	Output
A	A	A
A		⊖
A		⊖
A		⊖
A		⊖

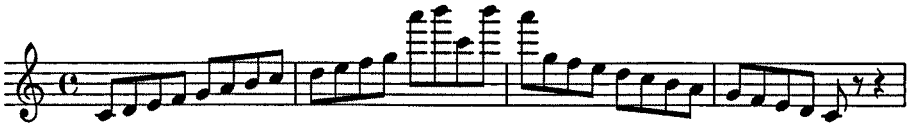
Comparator Summary:

```
Notes found:
Total Correct  1  20.00%
Total wrong    0  0.00%
Total Missing  4  80.00%
Total extra    0  0.00%
```

The summary table in Section 5.9 shows that the PitchTracker struggled with this test. The test performed with 1024 bins was the only one that produced a result. This result arises as a direct result of the PitchTracker's approach to note detection. Each subsequent note is viewed by the PitchTracker as a continuation of the first.

5.8.4 Test Four

This test is the same as Test Two but the note length has been halved. Original Score:



Comparator Output - Test Four							
No.	Orig	Comp	Output	No.	Orig	Comp	Output
1	<	Z	⊗	16	_		⊖
2	>	>	>	17	]	]	]
3	@	@	@	18	O	O	O
4	A	A	A	19	M		⊖
5	C	C	C	20	L	L	L
6	E	E	E	21	J	J	J
7	G	G	G	22	H	H	H
8	H	H	H	23	G	G	G
9	J	J	J	24	E	E	E
10	L	L	L	25	C	C	C
11	M		⊖	26	A	A	A
12	O	O	O	27	@	@	@
13	]	]	]	28	>	>	>
14	_	=	⊗	29	<	<	<
15	T	T	T	-	-	-	-

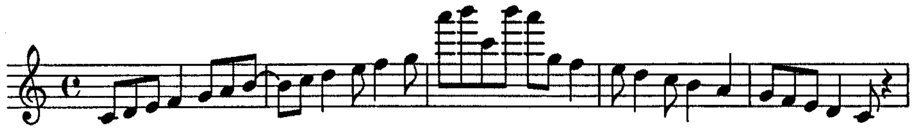
Comparator Summary:

```
Notes found:
Total Correct  24  82.76%
Total wrong     2  6.90%
Total Missing   3  10.340%
Total extra     0  0.00%
```

As the speed at which the notes are played increases, the accuracy of the PitchTracker decreases.

5.8.5 Test Five

This test is the same as Test Two but the with a combination of note lengths.
Original Score:



Comparator Output - Test Five							
No.	Orig	Comp	Output	No.	Orig	Comp	Output
1	<	<	<	16	_		⊖
2	>	>	>	17	]	]	]
3	@	@	@	18	O	O	O
4	A	A	A	19	M		⊖
5	C	C	C	20	L	L	L
6	E	E	E	21	J		⊖
7	G	G	G	22	H	H	H
8	H	H	H	23	G	G	G
9	J	J	J	24	E	E	E
10	L	L	L	25	C	C	C
11	M		⊖	26	A	A	A
12	O	O	O	27	@	@	@
13	]	]	]	28	>	>	>
14	_		⊖	29	<	<	<
15	T	T	T	-	-	-	-

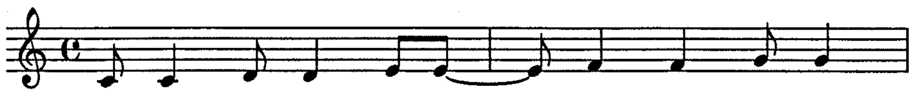
Comparator Summary:

```
Notes found:
Total Correct  24  82.76%
Total wrong     0   0.00%
Total Missing   5  17.24%
Total extra     0   0.00%
```

The PitchTracker has missed higher, and in some cases shorter, notes.

5.8.6 Test Six

This test is a five note ascending scale with each note repeated. The repeated note is twice the length of the first. Original Score:



Comparator Output - Test Six		
Orig	Comp	Output
<		⊖
<		⊖
>	>	>
>		⊖
@	@	@
@		⊖
A	A	A
A		⊖
C	C	C
C		⊖

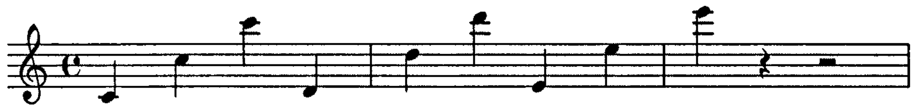
Comparator Summary:

```
Notes found:
Total Correct   4   40.00%
Total wrong     0   0.00%
Total Missing   6   60.00%
Total extra     0   0.00%
```

Apart from the first pair, the PitchTracker has assumed that the harmonics of the second note are a continuation of the first note.

5.8.7 Test Seven

This test is a three note ascending scale of grouped notes. Each group has three notes, of which each is an octave higher than the previous. Original Score:



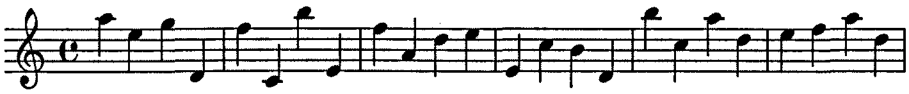
Comparator Output - Test Seven		
Orig	Comp	Output
<		⊖
H	H	H
T	T	T
>	>	>
J	J	J
V	V	V
@	@	@
L	L	L
X		⊖

Comparator Summary:

```
Notes found:
Total Correct  7  77.78%
Total wrong    0  0.00%
Total Missing  2  22.22%
Total extra    0  0.00%
```

5.8.8 Test Eight

This test contains twenty-four random notes all of length 1 second. Original Score:



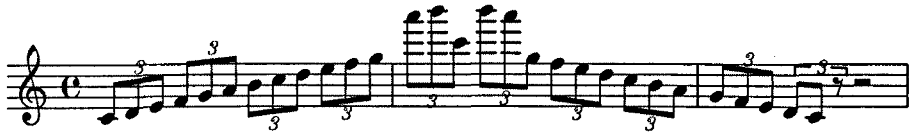
Comparator Output - Test Eight							
No.	Orig	Comp	Output	No.	Orig	Comp	Output
1	Q	r	⊗	13	@	@	@
2	L	L	L	14	H	H	H
3	O	O	O	15	G	G	G
4	>	>	>	16	>	>	>
5	M	=	⊗			=	⊕
		=	⊕	17	S	S	S
		>	⊕	18	H	H	H
6	<	<	<	19	Q	Q	Q
7	S	S	S	20	J	J	J
8	@	@	@	21	L	L	L
9	M		⊖	22	M		⊖
10	E	E	E	23	Q	Q	Q
11	J	J	J	24	J	J	J
12	L	L	L	-	-	-	-

Comparator Summary:

```
Notes found:
Total Correct  20  74.07%
Total wrong     2   7.41%
Total Missing   2   7.41%
Total extra     3  11.11%
```

5.8.9 Test Nine

This test is the same as Test Two but this time the note length is set to 1/3 of second. Original Score:



Comparator Output - Test Nine							
No.	Orig	Comp	Output	No.	Orig	Comp	Output
1	<	=	⊗	16	_		⊖
2	>	>	>	17	]	]	]
3	@	@	@	18	O	O	O
4	A	A	A	19	M		⊖
5	C	C	C	20	L	L	L
6	E		⊖	21	J	J	J
7	G	G	G	22	H	H	H
8	H	H	H	23	G	G	G
9	J	J	J	24	E	E	E
10	L	L	L	25	C	C	C
11	M		⊖	26	A	A	A
12	O	O	O	27	@	@	@
13	]		⊖	28	>	>	>
14	_		⊖			=	⊕
15	T	T	T	29	<	<	<

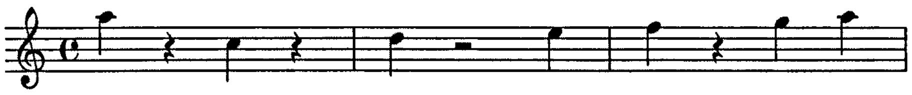
Comparator Summary:

```
Notes found:
Total Correct   22   73.33%
Total wrong      1   3.33%
Total Missing    6   20.00%
Total extra      1   3.33%
```

A consequence of the speed increasing is that the accuracy of the PitchTracker decreases.

5.8.10 Test Ten

This test is a series of random notes interspersed with rests. Original Score:



Comparator Output - Test Ten		
Orig	Comp	Output
Q	#	⊗
H	H	H
J	J	J
L	L	L
M		⊖
O	O	O
Q	Q	Q

Comparator Summary:

```
Notes found:
Total Correct   5   71.43%
Total wrong     1   14.29%
Total Missing   1   14.29%
Total extra     0   0.00%
```

5.9 Summary of PitchTracker Results

The results for the PitchTracker in terms of the number of Correct, Wrong, Missing and Extra notes are shown below[36] (Bold entries indicate the best result for a given test):

[36]A blank entry indicates that the PitchTracker failed to detect any notes and as a result did not produce a MIDI file.

	512				1024				2048			
Test No.	C	W	M	E	C	W	M	E	C	W	M	E
1	4	1	2	0	1	0	6	2	-	-	-	-
2	25	0	4	0	21	0	8	0	20	7	2	12
3	-	-	-	-	1	0	4	0	-	-	-	-
4	24	2	3	0	22	1	6	0	17	8	4	10
5	24	0	5	0	21	0	8	0	20	4	5	15
6	4	0	6	0	4	0	6	0	9	0	1	0
7	7	0	2	0	8	1	0	0	6	2	1	3
8	20	2	2	3	20	1	3	1	15	6	3	12
9	22	1	6	1	21	0	8	0	13	4	12	7
10	5	1	1	0	0	4	3	0	3	3	1	1

The results of Tests Three and Six, highlight the problem with the approach of the PitchTracker: repeated consecutive notes of the same pitch will be missed. Systems which rely on the "steady state" of a note will always underperform when compared to systems that rely on note onsets. This is borne out by the correlation between an increase in the speed of play resulting in a decrease in note detection accuracy, shown most explicitly by Test Nine.

The above results show that the PitchTracker performs best when set to use an FFT resolution of 512 bins. In theory, the performance for larger bin sizes could be improved by tuning other PitchTracker control variables. For example, the high number of extra notes in the case of 2048 FFT bins could be reduced by increasing the number of times a fundamental has to be present in consecutive windows before it is counted as a note.

The fact that the PitchTracker has to be tuned can be viewed as both a weakness and a strength. It's weakness is that it is not suited to universal pitch tracking situations. The strength of tuning means that if only one instrument is to be tracked, the PitchTracker can be adjusted to find the optimum settings for the instrument in question. The comparator along with a script, could be to used determine the optimum settings for a given instrument.

In summary the PitchTracker produced by Cirotteau has satisfactorily identified and tracked the pitches in the test wave files. These tests have however highlighted the need to detect the actual start of a note. A note detection system, based on the principles established in the preceding chapters, is presented in the next chapter.

6 The Detection of Notes

Inherent characteristics of a musical note produced by a real musical instrument are its harmonic and inharmonic structure. These features can be exploited in order to determine:

- the start of a note;
- the life of a note.

This section describes the processes involved in detecting the start of unknown notes and the subsequent tracking of their (unknown) harmonics through time sliced FFT frames and then storing relevant[37] data.

6.1 Overlapping Data

When moving from the time to frequency domain it is customary to employ overlapping FFT windows in order to build the spectral content of a signal over time. However, whilst the actual FFT windows overlap, the data within the window is zero-padded according to the level of window overlap, such that the signal is sliced into thin strips as shown in Figure 34. The problem with slicing the signal in this way is that the abrupt crossover from zero padding to actual signal introduces artefacts in the FFT results. The greater the overlap, the thinner the strip of data. With a frame size of 2048 samples an overlap of 8 gives a strip of data 256 samples wide, or 5ms long (using a Sample Rate of 48000). By definition such thin slices of signal cannot yield as much spectral information as a FFT window populated entirely with signal data. This makes it very difficult to track the growth of note information on a frame by frame basis; such thin strips of data result in FFT bins with very low magnitudes.

[37]Relevant data includes inharmonic and onset transient information.

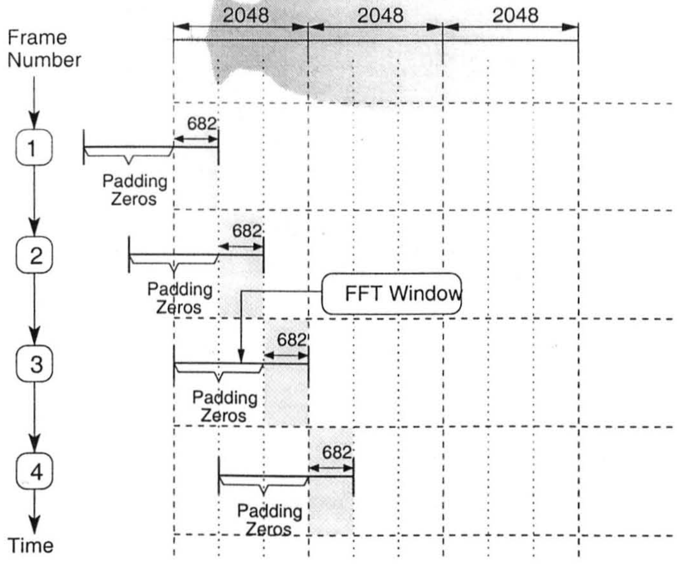


Figure 34: Overlapping zero padded FFT windows.

The alternative to thin strips of non-overlapping data is to use fully populated FFT frames as shown in Figure 35. This ensures that for a given slice of the input signal, there is sufficient data in the window to adequately represent each partial. As there is no crossover from zero-padded entries to actual data within a window, FFT artefacts are vastly reduced. This latter approach was adopted because it gives consistent partial information from frame to frame.

The system processes a wave by slicing it up into sections of 64 overlapping frames with an overlap of 256 samples. As each frame is processed the contents of each bin are stored in a number of arrays. These arrays store information concerning bin magnitude, phase and phase difference when compared to the same bin in the previous frame. Once the 64 frames of a given section have been processed, partial lists can be built which track spectral energy from frame to frame.

In order to build such lists it is necessary to identify which bins contain spectral peaks which relate to note information. The orthodox methodolgy is to employ a peak picking algorithm, having first eliminated peaks which fall below

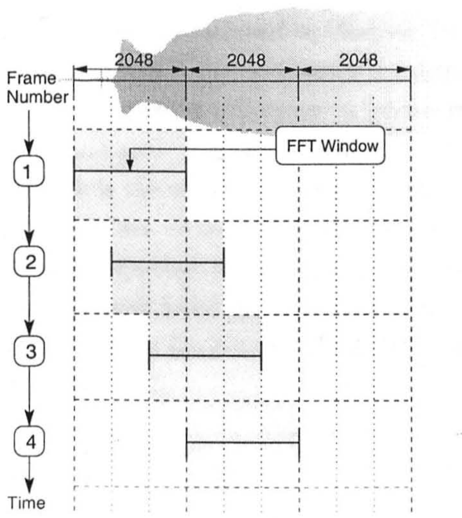


Figure 35: Fully populated overlapping FFT windows.

a certain amplitude threshold[50] . Locating (in)harmonic peaks within FFT results is non-trivial and in the context of note onset detection, is something of a chicken and egg situation. Before a note starts its (in)harmonics are (obviously) unknown, the system will therefore not know which peaks are relevant.

This problem could be overcome by employing some form of harmonic model. Such harmonic models have to make an assumption regarding the frequencies of the harmonics of a potential note. Such assumptions would be based on the "steady state" part of a note. This in turn relies on the false assumption that the harmonic content of the steady state is the same as that at the start. Such an approach would lead to vital, short lived (in)harmonics, being missed. At the start of a note the very low spectral energy of a harmonic makes it indistinguishable from surrounding noise peaks, rendering an amplitude-based peak picking algorithm virtually useless.

An alternative approach adopted in this method is to treat all peaks in a frame as valid note information. Spectral peaks which relate to a genuine note will persist in consecutive frames, whereas noise peaks will feature inconsistently in non-consecutive frames. The outcome of this is that the relevant peaks automatically group themselves into lists of genuine note data and spurious data e.g. noise. Noise produce very short intermittent lists which can be ignored but not necessarily discarded. This method is in keeping with the guiding principles of the human auditory system, established in Chapter Two.

As each peak is assigned to a list its frequency is calculated using the Phase-Vocoder technique. The expected difference in phase from the same bin in the previous frame is compared with the actual difference in bin phase. The expected phase difference is the advance in phase of the centering frequency of the bin, taking the proportion of frame overlap into account. The difference in the actual phase and expected phase relates directly to the deviation in frequency of the spectral peak from the centering frequency of the bin. This process is explained in depth by Cooper and Bailey[18], who in turn refer to the work of Brown and Puckette[12].

Algorithm 3 Peak Search Method

```
while $n < number\_of\_frames$ do
  if $env[i] < 0$ and $env[i-1] > 0$ then
    create a new instance of a list item shown in Figure 36
    fill data structure instance with peak data
    $n = n + 1$
  end if
end while
```

The first order differential of a frame is calculated and all positive to negative turning points are recorded as possible note related peaks, as shown in Algorithm 3. Having processed all overlapping frames, each spectral peak is added to the list structure, shown in Figure 36.

6.2 Building Lists

The first frame of results are used to construct an initial list of spectral peaks against which subsequent frames are compared. Algorithm 4 shows how data from subsequent frames are either added to an existing partial list or used to create a new list resulting in a store of all known spectral information (an example of which is shown in Figure 36). From this point on the phrase "Partial Lists" will be used to refer collectively to each List of Partials (LoP). Table 7 shows the raw data (as stored by the data structure in Figure 36) used for Figure 39. The data is shown in an expanded form (non-consecutive list items have been padded with zeros) for ease of display in a table (in total, 50 items of data were used to plot Figure 39).

The list building routines construct lists on a bin by bin basis. Vibrato,

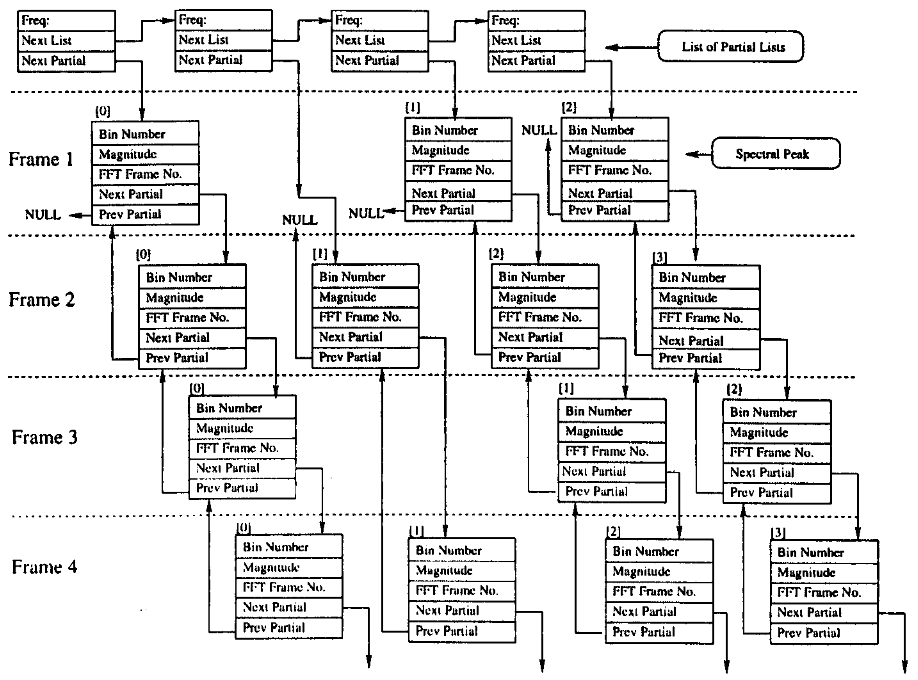


Figure 36: An example of the data structure used to store all potentially relevant note information.

Frame No.	Noise	Genuine Harmonic	Frame No.	Noise	Genuine Harmonic
1	0	3.53	16	0	1.66
2	0.74	3.27	17	0	3.85
3	0.5	2.97	18	0	10.37
4	0.54	2.78	19	0	19.73
5	0	2.37	20	0	29.51
6	0	1.76	21	0	38.27
7	0	1.39	22	0	44.13
8	0	0.83	23	0	47.56
9	0	0.97	24	2.07	50.4
10	0.98	0.86	25	0	53.8
11	0	1.05	26	0	58
12	0.28	1.01	27	0	62.76
13	0.62	1	28	0	67.06
14	1.06	0.94	29	0	70.98
15	0	0.77	30	0	73.48

Table 7: Table of the first thirty items of data used to plot part of Figure 39

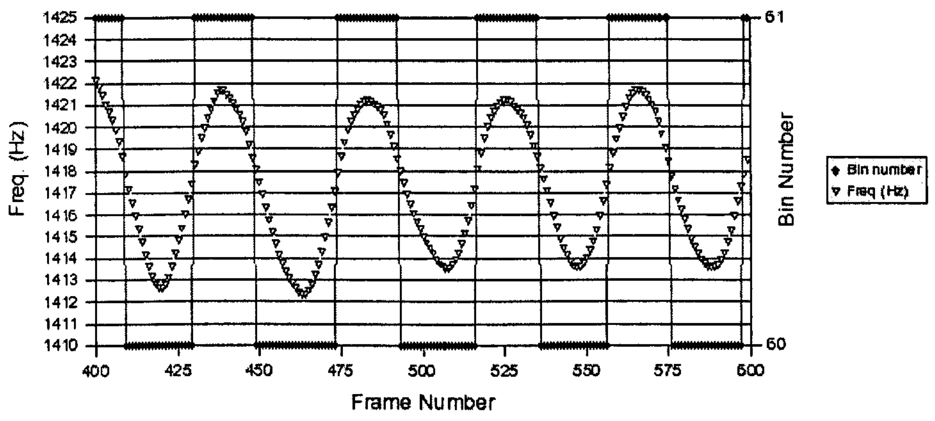


Figure 37: Showing the frequency and bin information of a spectral peak as it is tracked across successive frames.

and the volatile nature of transients in the onset of a note result in spectral energy peaks jumping back and forth across bin boundaries. A spectral peak tracking routine is used to determine whether an energy peak jumps, or "hops", from one bin to another. The routine builds a paired list of points where a LoP starts and stops. For example the start-stop list for the noise column of Table 7 would be $\{[2 \rightarrow 4], [10 \rightarrow 10], [12 \rightarrow 14], [24 \rightarrow 24]\}$. Starting with the LoP that contains the largest spectral peak, the start-stop list of each LoP is compared with an immediately adjacent LoP start-stop list. If the start-stop points inversely correspond, then the LoPs are merged. Start-stop lists that clash or overlap prevent LoPs from merging.

Each data item in the merged LoP stores the bin number of the list it originated from. Figure 37 shows a small section of a LoP for the third harmonic of a note played with vibrato on an oboe, recorded at a sampling frequency of 48,000Hz. As Figure 37 shows, the routine successfully tracks the spectral peak as it alternates between bins 60 and 61. According to Figure 37 the spectral peak "hops" from one bin to another when its frequency passes 1418Hz. This behaviour is entirely expected: the bin resolution for 2048 bins at a sample rate of 48,000Hz is 23.44Hz. Thus the cross over point, or bin hop threshold, of bins 60 and 61 is $(60 \times 23.44) + \frac{23.44}{2} = 1417.97$Hz.

At present LoPs are selected for submission to the spectral peak tracking routine using a top down approach, i.e. according to the magnitude of their

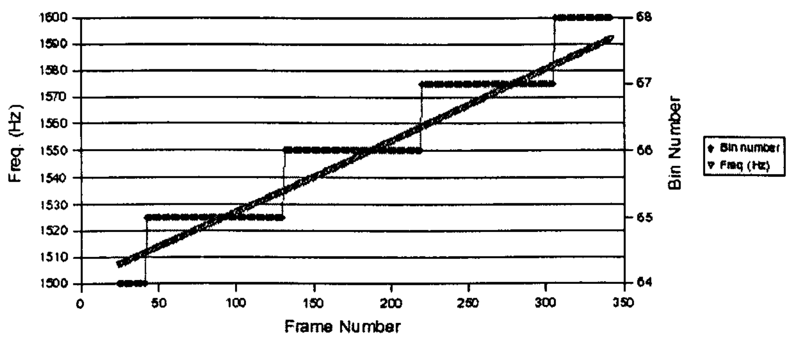


Figure 38: Showing the tracking of glissando as it crosses successive bins.

spectral peaks. This process is carried out once: that is, no attempt is made to merge a currently merged LoP with further LoPs. Further checks are deemed unnecessary because a spectral peak does not usually hop between more than two bins. The exception to this rule are glissandos which can tranverse a number of bins. For completeness, Figure 38 shows that if the peak tracking routine is made to operate accumulatively (that is, merged lists are subsequently compared to and merged with other adjacent LoPs (assuming such lists exist)), it can successfully track bin hops which result from glissandos. Figure 38 shows the tracking of a sine wave generated to change in a linear manner from 15,000Hz to 16,000Hz. A generated tone has been used in this example to provide proof of concept. In practice interference from other harmonics within a note causes noise in sections of a LoP which would need to be empty in order for it to be repeatedly merged. Bin 67 in Figure 38 covers frames 220 to 305. Any spurious noise peaks in frames 131 to 219 and 306 to 342 would prevent the LoP associated with bin 67 from merging with adjacent LoPs.

Spurious spectral peaks form non-consecutive lists of isolated peaks (e.g. the second list of Figure 36), which can be discarded. This method successfully tracks the growth of a harmonic from very low spectral energy levels as shown in Figure 39.

6.3 Identifying Potential Onset Points

At this stage the Partial Lists contain either noise or noise and a potential note onset. When presented with such a scenario, the brain-ear combination is able

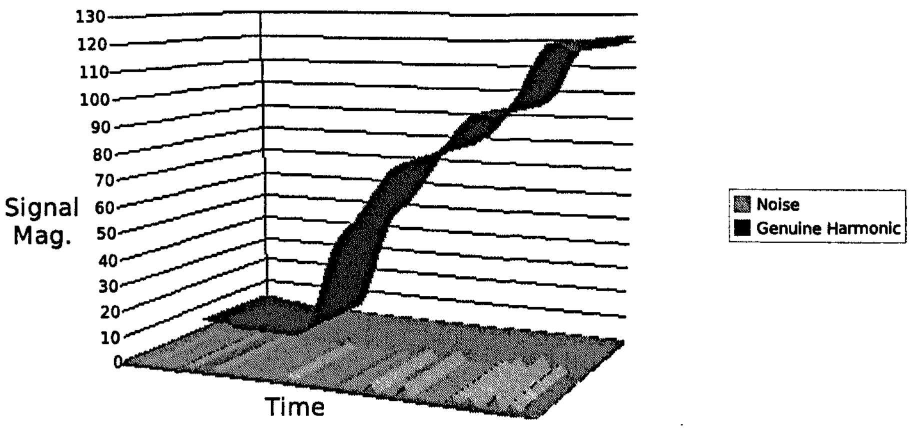


Figure 39: Magnitude of noise peaks and a genuine harmonic plotted against time.

to identify a change in sound energy and distinguish whether the cause of this change was due to noise or a note. A factor in making this distinction is that the magnitude of data relating to a genuine note will increase throughout its onset, whereas whilst noise energy levels can fluctuate, they do not exhibit the same growth characteristics as a note. The sound from a musical instrument is complex, built up from a number of related harmonics, each contributing to the overall energy level of the note. The growth of a note can be identified as a rising trend in the Partial Lists, separating note data from that produced by noise. The system flags any LoP which contains four consecutive frames whose partial magnitudes exhibit an overall minium increase of 0.5% of the total available FFT energy. This mimics the just noticeable difference threshold of the human ear. Such growth is considered as primary growth and the LoP is modified to indicate this. A second sweep flags lists which exhibit an overall minium increase of 0.1% of the total available FFT energy. This lower rate of change is considered as secondary growth. This second sweep ensures that as much relevant information as possible is included in the data of a note.

The purpose of these low minimum levels is twofold: it accounts for the fact that overall note energy is spread across a number of (in)harmonics, and also allows notes with slow onsets to be detected. Simultaneous primary growth of

Algorithm 4 The creation of Harmonic Lists.

```
for all peaks in the first fft frame do
  add a peak'sBin Index to the Harmonic List
end for
while $n < number\_of\_frames$ do
  for all peaks in fft frame $n$ do
    if peak Bin Index := Bin Index already in Harmonic List then
      add peak to end of list for current Bin Index
    else
      insert new list heading for this peak Bin Index
    end if
  end for
  $n = n + 1$
end while
```

2 or more LoPs, which are non-adjacent in a numerical rather than list sense, are deemed to be genuine notes. In theory, ensuring that LoPs are numerically non-adjacent is surplus to requirement. A LoP is a series of the same spectral bin through time, which by definition means adjacent LoPs cannot represent consecutive bins. However, this situation can arise if LoPs have been merged because of bin hopping. For example, a merged list containing data from bins 2 and 3 is considered numerically consecutive to another list containing data from bins 4 and 5.

6.4 Harmonic Growth.

Having found a position in a LoP which exhibits primary or secondary growth, the system progresses through the remainder of the LoP and records the number of list items which continue to exhibit growth. The harmonic growth length is not necessarily the length (through time) of the traditional "attack" of a note (i.e. until peak amplitude is achieved). Figure 40 illustrates this difference. By the time point A in Figure is reached, the harmonics of the note are fully established, the subsequent growth of the harmonic is considered to be a change in the dynamic level of the note. Thus, point A is the stop point of the growth of that particular harmonic. This restriction reduces the range in which simultaneous growth of harmonics can be detected, which in turn aids the allocation of Partial Lists into groups, explained in the next section.

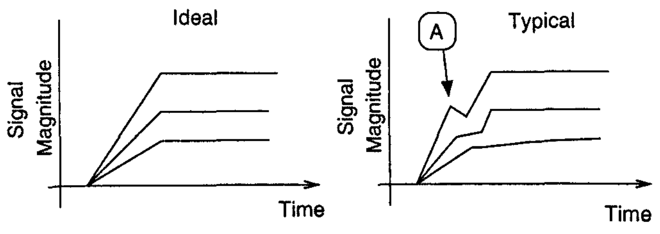


Figure 40: Showing the difference between the onset growth of an ideal note and an example of a typical note.

6.5 Grouping

Chapter Two discussed the manner in which the human auditory system groups incoming sound information. As the use of harmonic models is being avoided LoPs are grouped according to temporal information. Thus, in order to determine where (in)harmonic growth occurs, flagged LoPs are grouped according to their growth start and stop points. The scope of each group is determined by its members. A LoP can only be a member of one group. Membership of a group is dependent on the start or stop point of a LoP overlapping with the start or stop point of a group.

Any group containing only one LoP does not meet the two or more LoP requirement set earlier and is therefore not indicative of the start of a new note. However, it is not immediately discarded. The single LoP is compared against existing note data, the reason for this is given in Section 6.7. The group with the lowest start point is taken to be indicative of the position in the Partial Lists where a genuine note starts. If this note start point is towards the end of the results in a section the system repositions itself by advancing forwards as if no note has been found and restarts the list building process. The potential note start will then appear towards the middle of a LoP, ensuring that all the (in)harmonics of a note and their growths will be detected.

The overlap described in Section 6.1 gives a time resolution of 5.33ms. This resolution can be dramatically improved through the use of small steps to reverse through signal data.

6.6 Listening Backwards

In order to find an accurate actual note start, the system steps backwards[38] through the signal data by 16 samples, repositioning a single FFT frame as long as:

- it detects 2 or more spectral peaks that correspond to the harmonics of a group.
- for a given bin, the magnitude of the spectral peak detected is less than that of the bin in the original LoP where the potential start was detected. This is so that repeated notes which run into each other can be detected and tracked.

The step back size of 16 samples results in a time resolution of 333μs (using a sample rate of 48,000Hz). The plots[39] shown in Figure 41, starting at the top left and ending at the bottom right, show what the system "hears" as it steps back through the wave.

As the frame moves closer to the start of a note, it becomes more difficult to identify spectral peaks which correspond to the note. The spectral energy of a partial becomes spread across bins such that there is no discernable peak. This loss of peak marks the point at which a given partial begins to grow. In this way each note (in)harmonic is assigned an individual start point. The start point is taken as the end of the frame in which the (in)harmonic peak loss occured, as shown in Figure 42[40]. The signal level at the assignment point is typically of the order of -49dB, compared to the signal at its peak. In reality (and by definition) this point will always lag the physical point in the wave file where data is known to begin. The note (rather than (in)harmonic) start point is derived from the average of its (in)harmonic start points.

The details of the note including (in)harmonic start points, actual start point and LoP Bin numbers are gathered into an instance of a note storage class. Each instance of this class is added to a database of known notes. Having indentified a new note, the Partial Lists are used to find its peak and the next section of

[38]This backtracking routine is designed to increase accuracy. However it can be thwarted by the presence of matching harmonics in a previous note which obey the stated magnitude rule. Thus backtracking is limited to a quarter of a second. In practice, this limit is rarely reached.

[39]A plot was taken once every four steps - the equivalent of stepping back 64 samples at a time.

[40]The positioning of the FFT frame in Figure 42 is purposely shown as ideal so that its position relative to the note can be seen.

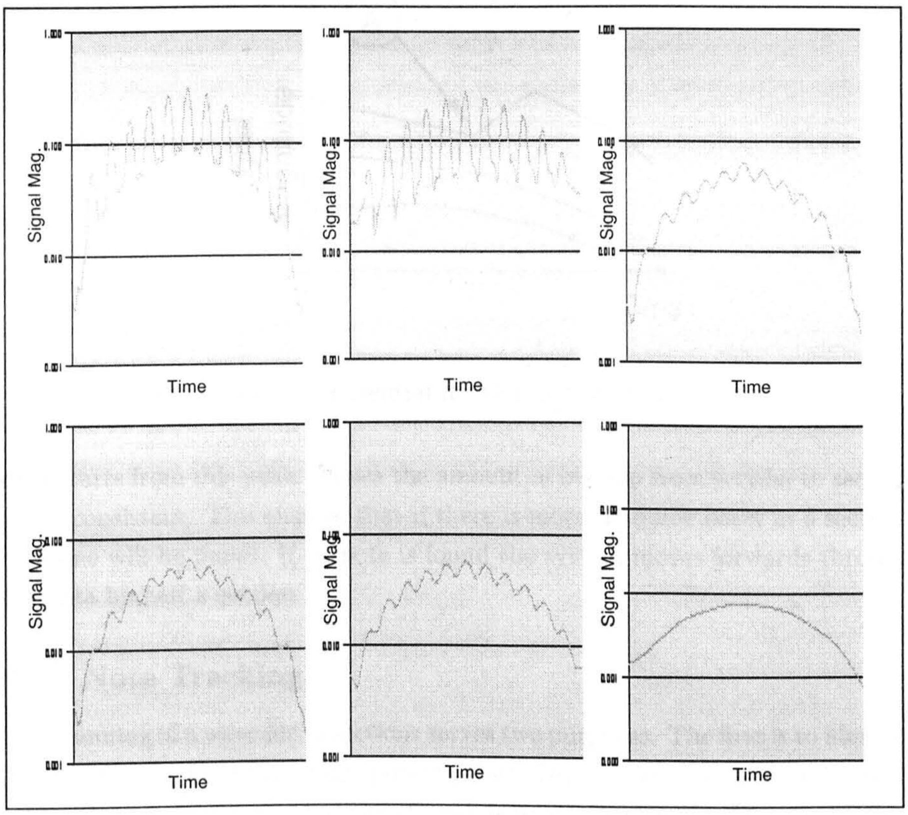


Figure 41: The detection of the onset of a note when stepping backwards.

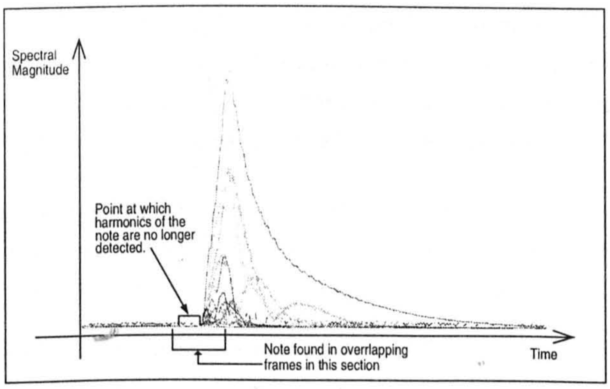

Spectral Magnitude
Point at which harmonics of the note are no longer detected.
Note found in overrlapping frames in this section
Time

Figure 42: Positioning of frames when detecting the onset of a note.

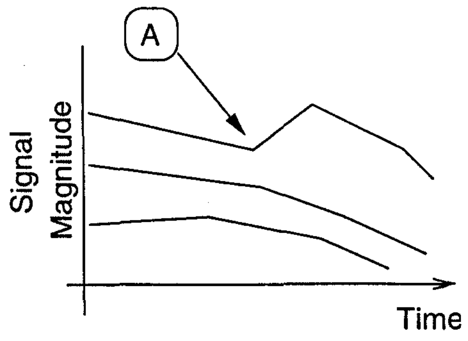


Figure 43: The potential death point of a harmonic.

data starts from this peak. Hence the amount of overlap from section to section is not consistent. This ensures that if there is more than one onset in a section, each one will be found. If no note is found the system moves forwards through the data by half a section.

6.7 Note Tracking

The scanning of a wave file in sections serves two purposes. The first is to identify positions in the file which correspond to the onset of a note. The second, which can only occur once a note has been found, is to monitor its life. This monitoring takes place before the detection of new potential notes. Fully populated LoPs with bin indexes which correspond to existing note (in)harmonics are, subject to item 2 below, removed rom the Partial Lists. This removal prevents dynamic fluctuations from triggering the onset detection mechanism. As the system progresses through a file it changes the status of a note from "growing", to "alive" and finally "dead". Conversely, note (in)harmonics have only two states: living or dead. There are two circumstances which signal the death of a (in)harmonic:

1. When the system can no longer detect partials that correspond to a given (in)harmonic, the (in)harmonic is allocated a death point (which corresponds to the last occurence of a partial in a LoP).

2. If a LoP contains growth which is characteristic of a note onset (see Figure 43), linear regression is used to determine if the harmonic has a negative gradient (indicative of a decaying note) prior to the growth point. If the harmonic is decaying it is given a death point which corresponds to

the beginning of the growth (point A on Figure 43). The LoP is not removed from the list of Partial Lists and can therefore trigger the onset detection routines. This permits the detection of overlaping, repeated identical notes. An isolated occurence of growth (as shown in Figure 43) is obviously not the beginning of a new note and this LoP will therefore be the sole member of a group, as described in Section 6.5. This LoP shows an increase in the dynamic level of the harmonic. The comparison referred to in Section 6.5 checks to see if this growth point corresponds to the death point of a matching note harmonic. If it does the harmonic is brought back to life by the removal of its death point.

The status of the note is set by the number of live harmonics it contains. A note dies when less than two of its harmonics are alive. The status of a note helps to prevent the occurence of duplicate notes (which can occur if partials that correspond to existing note harmonics are allowed to retrigger the onset mechanism).

The non-linear repositioning and therefore non-linear spacing of each section make it very difficult to build a continuous LoP that lasts for the entire length of a note. To overcome this problem, the information in the note database is used to perform subsequent sweeps of the data on a per-note basis allowing LoPs to be built for the entire life of a note.

The process of the second sweep is slightly different. The system automatically sets the size of a data section using the start and stop points of a note. Appropriate calculations are made to determine the number of overlapping frames needed in order to contruct LoPs which will depict the life of the note (in)harmonics. Each instance of a note provides the following information:

- Onset times of its (in)harmonics.
- Rise times of its (in)harmonics.
- Frequency of its (in)harmonics.
- Finishing times of its (in)harmonics.

This note detection system will hereby be referred to as a Note Tracking System (NTS). System tests are presented in the next chapter.

7 System Tests and Validation of Output

This chapter begins with the analysis of CSound generated test cases. The use of CSound gave complete control of the nature of the generated sounds allowing the performance of the frequency extraction routine to be evaluated. The ten system tests from Chapter Five used on the PitchTracker software, are used to test the NTS. Tests using recordings of real instruments are included at the end of the chapter.

7.1 Monophonic CSound Test Case

A wave file containg three notes was generated using CSound. Each note has 4 harmonics weighted to be 25% less than the previous harmonic as shown below:

```
f1      0       8192    10      1       0.75    0.5     0.25
;ins strt dur  amp(p4)   freq(p5)  attack(p6)     release(p7)
i3   1    1    30000     2930       0.45             0.25
i3   3    1    30000     3412       0.45             0.25
i3   5    1    30000     3413       0.45             0.25
```

The first sweep of the data gave the following results (shown graphically in Figure 44 where each vertical line indicates where a note start has been identified):

```
TN 3
nN 0
45916 45856 95616 96256 4
124 125 46096 95744
250 0 45856 96256
375 0 45856 95232
500 0 45856 95232
nN 1
141916 141856 192128 193536 4
145 146 142096 192000
291 0 141856 191488
```

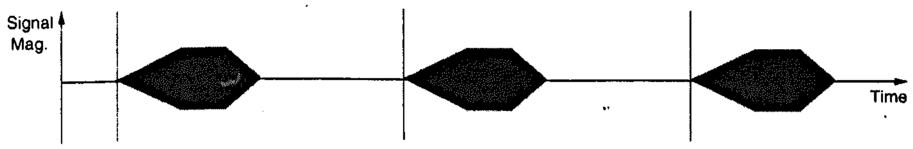


Figure 44: Note start points for a monophonic test case.

```
437 0 141856 191488
582 0 141856 193536
nN 2
237916 237856 301312 287744 4
146 0 237856 287488
291 292 238096 287488
437 0 237856 287488
582 0 237856 287744
```

Where TN = Total Notes, nN = Note Number. The line following the note number reports[41]: the average harmonic start point, the lowest harmonic start point, the average harmonic finish point, the latest harmonic finish point and the number of detected note harmonics. Each subsequent line provides details of each harmonic: Bin number, merged bin number (zero if no bin hopping occurred) and the start and stop points of the harmonic (numbers relate to nth sample in a wav file). The NTS successfully found three notes each with four harmonics. Using the average start and stop points of nN0:

$$start\ time = \frac{45916}{48000} = 0.96s$$

$$length = \frac{96616 - 45916}{48000} = 1.035s$$

Thus the NTS has determined that the first note lasted for 1.035s and started 0.96s into the file. A second sweep of the data was made to enable a complete LoP for each harmonic to be built, from which the frequency of the harmonic can be calculated. Figure 45 shows how the frequency and magnitude of a harmonic change on a frame by frame basis.

As the harmonic number increases, the signal strength of the harmonic decreases making it harder to track its frequency. However, as Figure 46 shows, the NTS is able to correctly determine the frequency of each harmonic for the above example. A comparison of the CSound file and the data shown in Table 8, which contains extracts of the data used to create the graph in Figure 46, shows the accuracy of frequency resolution for a given note harmonic. For example, the 2nd harmonic is double the frequency of the first. $2930 \times 2 = 5860$Hz, which as Table 8 shows, is what the NTS has found.

The 2nd and 3rd notes, described in the Csound score file, are only 1Hz

[41] In this example, due to the harmonic nature of the synthesised test case, all results are described in terms of harmonics.

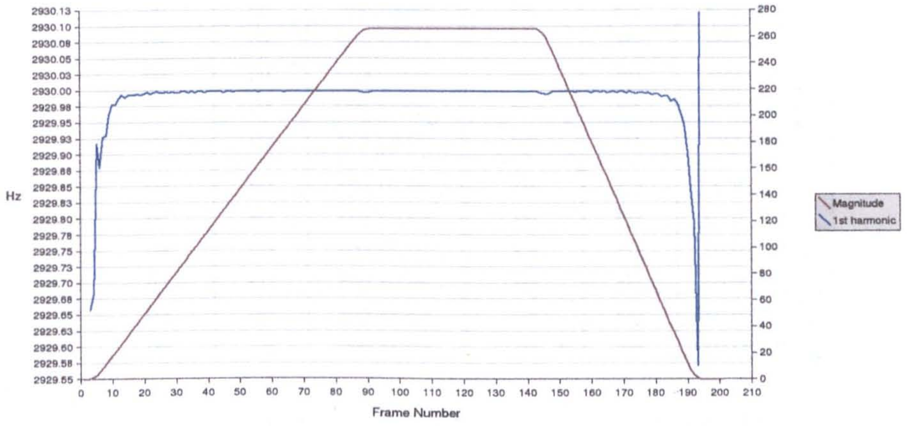


Figure 45: The frequency and magnitude of the first harmonic of a note against time.

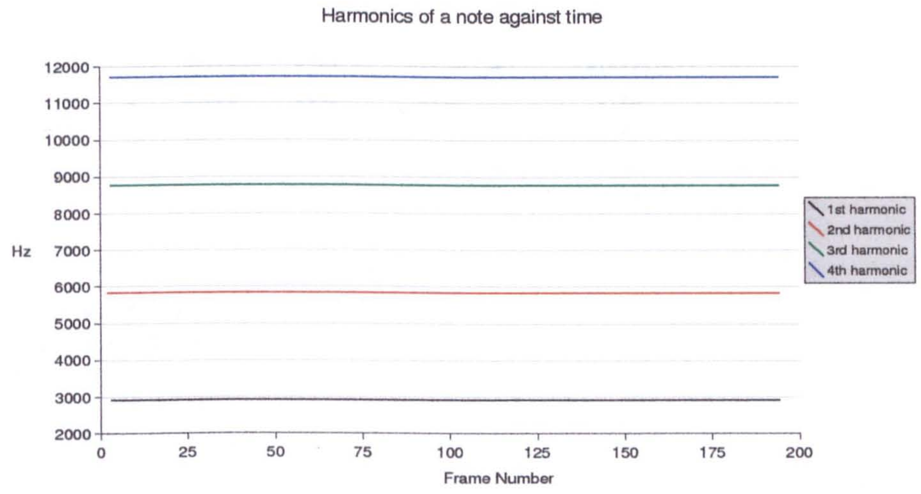


Figure 46: The frequency of all the harmonics from the first note in the monophonic synthesized test case.

Frame No.	Bin No.	Frequency (Hz)	Frame No.	Bin No.	Frequency (Hz)
120	125	2929.998535	120	375	8789.994141
121	125	2929.999268	121	375	8789.994141
122	125	2929.998535	122	375	8789.994141
123	125	2929.999268	123	375	8789.994141
124	125	2929.998535	124	375	8789.994141
125	125	2929.998535	125	375	8789.995117
120	250	5859.997559	120	500	11719.995117
121	250	5859.997070	121	500	11719.995117
122	250	5859.997559	122	500	11719.995117
123	250	5859.997559	123	500	11719.995117
124	250	5859.997559	124	500	11719.995117
125	250	5859.997070	125	500	11719.995117

Table 8: Extracts of the data used to plot Figure 46

apart. Figure 47 shows that the NTS has successfuly resolved the separate frequencies of these notes.

7.2 Simple Polyphonic CSound Test Case

The CSound score file was modified so that it contained two notes which would overlap:

```
;ins strt dur  amp(p4)  freq(p5)  attack(p6)   release(p7)
i3   1    4    30000    2930       0.45          0.25
i3   3    1    30000    3412       0.45          0.25
```

The NTS correctly found two notes, as shown below:

```
TN 2n
N 0
45848 45776 239790 240128 4
125 0 45776 240056
250 0 46064 240128
375 0 45776 239360
500 0 45776 239616
nN 1
142840 142832 215872 226560 4
146 147 142848 226048
290 291 142832 219648
437 0 142848 226560
582 583 142832 191232
```

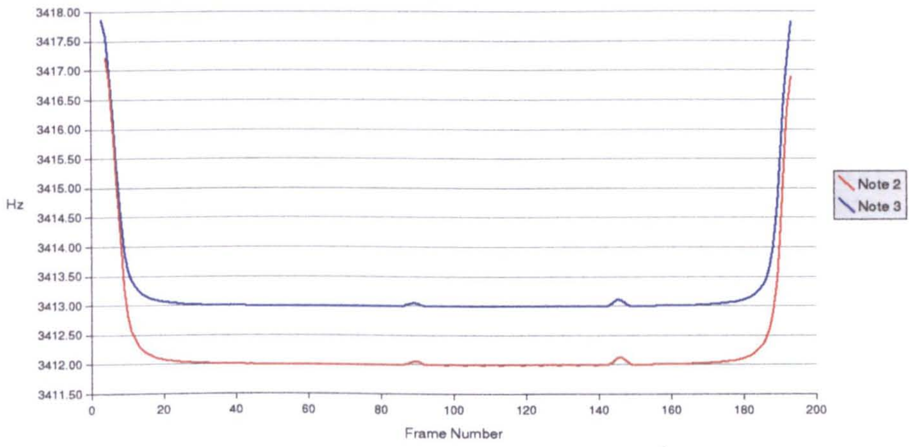


Figure 47: The frequency of the first harmonic from the second and third notes in the monophonic synthesised test case.

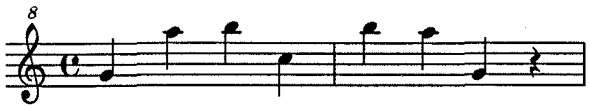
7.3 Comparator System Tests

This sub-section is repeated from Section 5.8. For each test case the midi file output of gmc is shown as a musical score, having been converted into LilyPond format by midi2lily. The LilyPond user manual states that "*human players are not rhythmically exact enough to make a midi to LY conversion trivial. midi2ly tries to compensate for these timing errors, but is not very good at this. It is therefore not recommended to use midi2ly for human-generate midi files.*" The files generated by the NTS are rather "human" in nature. The outcome of midi2ly being "not very good" at compensating for human timing is a segmentation fault. Consequently only test one shows a musical representation of the NTS's output. The NTS provides timing data which will be used in the tests. For ease of reference, the test descriptions will be repeated.

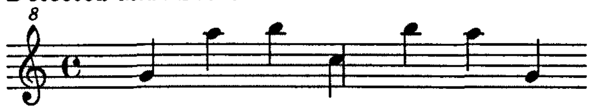
7.3.1 Test One

This test consists of seven notes of 1 second duration, five of which are in a very high register.

Original MIDI Score:



Detected MIDI Score:



Comparator Output - Test One				
Orig	Comp	Output	Timing	Length
O	O	O	0.04	0
]	]	]	0.04	-0.62
_	_	_	0.04	-0.4
T	T	T	0.04	-0.03
_	_	_	0,04	-0.6
]	]	}	0.04	-0.55
O	O	O	0.04	0.04

The comparator provides the following summary:

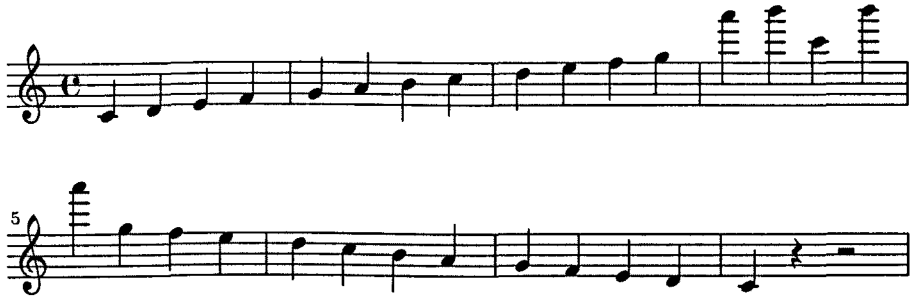
```
Notes found:
Total Correct  7  100.00%
Total wrong    0  0.00%
Total Missing  0  0.00%
Total extra    0  0.00%
```

Every note in this test has been given an onset time that leads the timing of the original MIDI data file. The onset position is recorded as the point at which note harmonics can no longer be heard. Thus it is acceptable for the onset position to lead in this way.

The timings given in the Length column show the difference between the length of the original note and the measured note time. A negative number indicates that the measured note is shorter than the original. The shorter note lengths are accounted for by the difference between the MIDI note length (the time between the note-on and note-off signals) and perceived note length. As the musical scores above show, the notes for this test are in the upper registers of the piano where notes decay very quickly. Consequently the perceived length of the note will always be shorter than the MIDI note length.

7.3.2 Test Two

This test is a two octave scale (29 notes) with the note length set to 1 second.
Original Score:

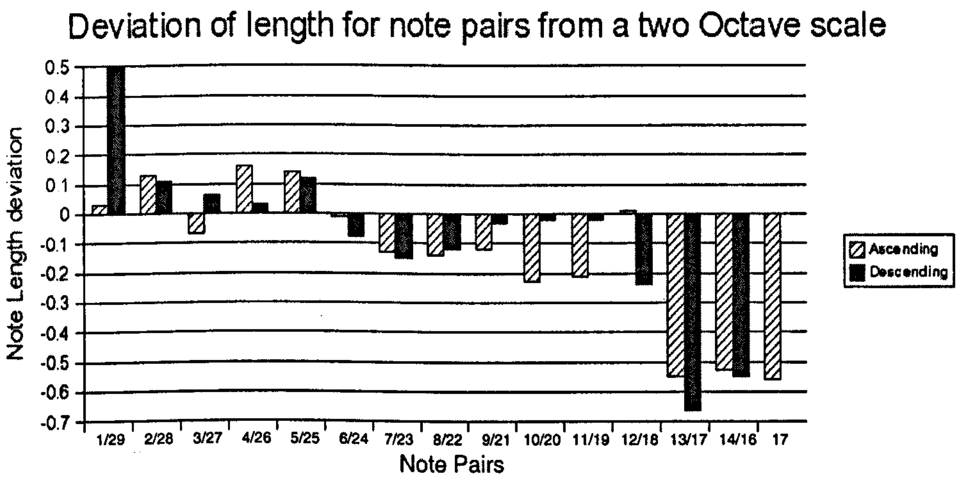


Comparator Output - Test Two											
No.	Orig	Comp	Output	Timing	Length	No.	Orig	Comp	Output	Timing	Length
1	<	<	<	0.04	0.03	16	_	_	_	0.04	-0.55
2	>	>	>	0.04	0.13	17	]	]	]	0.04	-0.66
3	@	@	@	0.04	-0.07	18	O	O	O	0.04	-0.24
4	A	A	A	0.03	0.16	19	M	M	M	0.04	-0.02
5	C	C	C	0.04	0.14	20	L	L	L	0.05	-0.02
6	E	E	E	0.06	-0.01	21	J	J	J	0.04	-0.03
7	G	G	G	0.03	-0.13	22	H	H	H	0.04	-0.12
8	H	H	H	0.04	-0.14	23	G	G	G	0.04	-0.15
9	J	J	J	0.03	-0.12	24	E	E	E	0.05	-0.08
10	L	L	L	0.04	-0.23	25	C	C	C	0.04	0.12
11	M	M	M	0.04	-0.21	26	A	A	A	0.05	0.03
12	O	O	O	0.04	0.01	27	@	@	@	0.05	0.06
13	]	]	]	0.04	-0.55	28	>	>	>	0.05	0.11
14	_	_	_	0.04	-0.53	29	<	<	<	0.05	0.5
15	T	T	T	0.04	-0.56	-	-	-	-	-	-

Comparator Summary:

```
Notes found:
Total Correct   29   100.00%
Total wrong      0   0.00%
Total Missing    0   0.00%
Total extra      0   0.00%
```

The detected note start times lead those of the original data as expected. The following graph shows the deviation of the length of each note from the 1 second length specified in the original MIDI file:

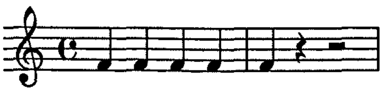


The last note of a scale is not masked by subsequent notes and is left to naturally decay.[42]. This accounts for the longer length of note 29 in the first note pair (similar behaviour is found in the other tests based on a 2 octave scale). The trend towards shorter notes on the right hand side of the graph is due to the rapid decay of high notes, as explained in Test one.

[42]Assuming the "steady state" section of a note is still sounding, unless given an explicit volume value of zero, a MIDI "note-off" command signals the start of a note's decay not its actual end[72, p.297].

7.3.3 Test Three

This test consists of five repeated notes at the same pitch of 1 second MIDI duration. The actual sounds in the wave file overlap. Original Score:



Comparator Output - Test Three				
Orig	Comp	Output	Timing	Length
A	A	A	0.03	0.05
A	A	A	0.05	-0.92
A	A	A	0.05	-0.99
A	A	A	0.07	-0.12
A	A	A	0.04	0.35

Comparator Summary:

```
Notes found:
Total Correct  5  100.00%
Total wrong    0  0.00%
Total Missing  0  0.00%
Total extra    0  0.00%
```

The MIDI conversion utilities were proved to be error free by extracting note and timing information from MIDI files generated by GMC. It was therefore assumed that the length timing errors of the second and third notes were due to a bug in the NTS. Due to the fact that two of the notes have a measured length close to zero it was initially assumed that the death point of the previous note was somehow being used to inadvertently set the death point of the current note. A re-run with debugging information switched on however showed this was not the case. The following table shows a summary of average start and stop times (given in terms of sample position) for each note generated by the NTS, along with manually calculated length timings (sample rate = 48000Hz):

No.	Start	Stop	Length(secs)
1	46321	94911	1.01
2	93336	139750	0.97
3	141453	186628	0.94
4	188766	229031	0.84
5	237659	300393	1.31

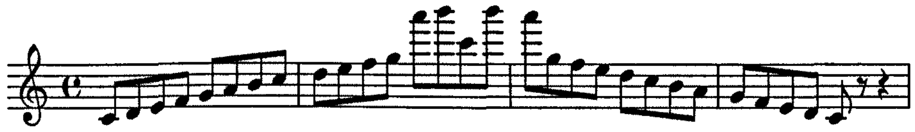
This table shows that NTS has performed correctly. Inspection of the resulting MIDI file shows that an error has arisen from an assumption. The table above shows that note 1 ends after the start of note 2. The MIDI file reflects this with two consecutive note-on events, followed by the note-off event for the first note. When extracting note timings, the extraction utility assumed that having found a note-on event, the next note-off event with the same pitch would be the end point of the note. This assumption is correct as long as two consecutive (overlapping) notes are of different pitch[43]. The extraction utility was modified so that it could not reuse note-off events and a new timing file was generated, giving the following results:

Comparator Output - Test Three with note length correction				
Orig	Comp	Output	Timing	Length
A	A	A	0.03	0.05
A	A	A	0.05	0.01
A	A	A	0.05	-0.02
A	A	A	0.07	-0.12
A	A	A	0.04	0.35

[43]Technically this assumption is correct because it is impossible, using orthodox methods, to produce consecutive notes of the same pitch with overlapping note-on/off events.

7.3.4 Test Four

This test is the same Test Two but the note length has been halved. Original Score:



Comparator Output - Test Four											
No.	Orig	Comp	Output	Timing	Length	No.	Orig	Comp	Output	Timing	Length
1	<	<	<	0.03	0.16	16	_	_	_	0.04	-0.01
2	>	>	>	0.04	0.29	17	]	]	]	0.04	-0.12
3	@	@	@	0.04	0.21	18	O	O	O	0.04	-0.22
4	A	A	A	0.04	0.33	19	M	M	M	0.05	0.3
5	C	C	C	0.04	0.17	20	L	L	L	0.04	0.07
6	E	E	E	0.04	0.29	21	J	J	J	0.04	0.15
7	G	G	G	0.04	0.18	22	H	H	H	0.04	0.13
8	H	H	H	0.04	0.35	23	G	G	G	0.04	0.14
9	J	J	J	0.04	0.08	24	E	E	E	0.04	0.02
10	L	L	L	0.04	0.16	25	C	C	C	0.04	0.13
11	M	M	M	0.04	0.34	26	A	T	⊗	-	-
12	O	O	O	0.04	-0.01	-	-	T	⊕	-	-
13	]	]	]	0.04	-0.11	27	@	@	@	0.04	0.13
14	_	_	_	0.04	-0.13	28	>	>	>	0.04	0.17
15	T	T	T	0.04	0.14	29	<	<	<	0.05	0.51

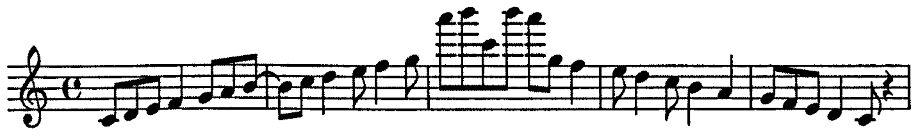
Comparator Summary:

```
Notes found:
Total Correct  28  93.33%
Total wrong     1  3.33%
Total Missing   0  0.00%
Total extra     1  3.33%
```

This test shows similar trends in terms of note length, to Test 2. The wrong note (number 26) is as a result of the system failing to detect the fundamental harmonic of the note. This failure is attributed to a combination of the over-run of the previous note and the note order of the scale, resulting in a masking effect. This failure has has had a subsequent effect as the same wrong note is re-detected causing an extra note error. Missing the fundamental frequency can result in the premature death of a note, allowing its harmonics to be redetected.

7.3.5 Test Five

This test is the same as Test Two but the with a combination of note lengths.
Original Score:



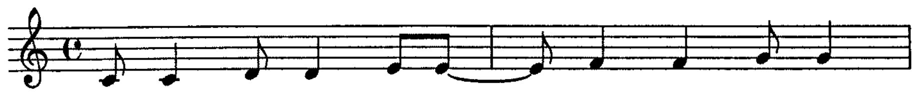
Comparator Output - Test Five											
No.	Orig	Comp	Output	Timing	Length	No.	Orig	Comp	Output	Timing	Length
1	<	<	<	0.03	0.16	16	_	_	_	0.03	-0.2
2	>	>	>	0.04	0.29	17	]	]	]	0.04	-0.12
3	@	@	@	0.04	0.21	18	O	O	O	0.04	-0.02
4	A	A	A	0.04	-0.01	19	M	M	M	0.04	-0.18
5	C	C	C	0.05	0.35	20	L	L	L	0.03	0.09
6	E	E	E	0.04	0.23	21	J	J	J	0.04	-0.06
7	G	G	G	0.04	-0.11	22	H	H	H	0.03	0.14
8	H	H	H	0.03	0.27	23	G	G	G	0.03	-0.04
9	J	J	J	0.03	-0.12	24	E	E	E	0.04	-0.1
10	L	L	L	0.04	0.12	25	C	C	C	0.04	0.15
11	M	M	M	0.04	-0.21	26	A	A	A	0.05	0.35
12	O	O	O	0.04	0	27	@	@	@	0.04	0.16
13	]	]	]	0.04	-0.05	28	>	>	>	0.03	0.13
14	_	_	_	0.04	-0.14	29	<	<	<	0.04	0.51
15	T	T	T	0.09	0.05	-	-	-	-	-	-

Comparator Summary:

```
Notes found:
Total Correct   29   100.00%
Total wrong      0     0.00%
Total Missing    0     0.00%
Total extra      0     0.00%
```

7.3.6 Test Six

This test is a five note ascending scale with each note repeated. The repeated note is twice the length of the first. Original Score:



Comparator Output - Test Six				
Orig	Comp	Output	Timing	Length
<	<	<	0.03	0.85
<	<	<	0.04	0.15
>	>	>	0.04	0
>	>	>	0.04	.13
@	@	@	0.04	0.03
@	@	@	0.05	0.04
A	A	A	0.03	-0.01
A	A	A	0.05	0.18
C	C	C	0.03	0.02
C	C	C	0.04	0.24

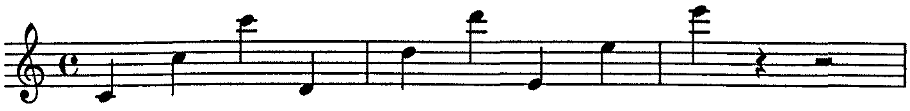
Comparator Summary:

```
Notes found:
Total Correct   10   100.00%
Total wrong      0     0.00%
Total Missing    0     0.00%
Total extra      0     0.00%
```

As this test contains repeated notes the test results initally displayed the same timing errors as Test Three. The results above are from a second run with MIDI timing data provided by the modified extraction utility.

7.3.7 Test Seven

This test is a 3 note ascending scale of grouped notes. Each group has three notes and each note is an octave higher than the previous note in a group. Original Score:



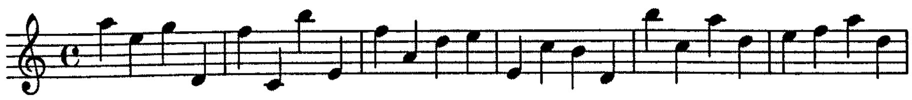
Comparator Output - Test Seven				
Orig	Comp	Output	Timing	Length
<	<	<	0.03	0.26
H	H	H	0.04	-0.21
T	T	T	0.04	-0.17
>	>	>	0.04	0.24
J	J	J	0.04	-0.1
V	V	V	0.04	-0.08
@	@	@	0.04	-0.05
L	L	L	0.04	-0.12
X	X	X	0.04	0.12

Comparator Summary:

```
Notes found:
Total Correct  9  100.00%
Total wrong    0  0.00%
Total Missing  0  0.00%
Total extra    0  0.00%
```

7.3.8 Test Eight

This test contains twenty-four random notes all of length 1 second. Original Score:



Comparator Output - Test Eight											
No.	Orig	Comp	Output	Timing	Length	No.	Orig	Comp	Output	Timing	Length
1	Q	Q	Q	0.04	-0.05	13	@	@	@	0.04	0.03
2	L	L	L	0.04	-0.17	14	H	H	H	0.04	-0.11
3	O	O	O	0.04	-0.05	15	G	G	G	0.04	-0.11
4	>	>	>	0.04	0.04	16	>	>	>	0.04	-0.05
5	M	M	M	0.05	-0.28	17	S	_	⊗	-	-
6	<	<	<	0.04	-0.03	18	H	H	H	0.05	-0.17
7	S	_	⊗	-	-	19	Q	Q	Q	0.04	-0.03
8	@	@	@	0.07	0.2	20	J	J	J	0.04	-0.11
9	M	M	M	0.04	-0.27	21	L	L	L	0.05	-0.16
10	E	E	E	0.04	0.12	22	M	M	M	0.04	-0.1
11	J	J	J	0.05	-0.14	23	Q	Q	Q	0.05	-0.02
12	L	L	L	0.04	-0.26	24	J	J	J	0.04	0.83

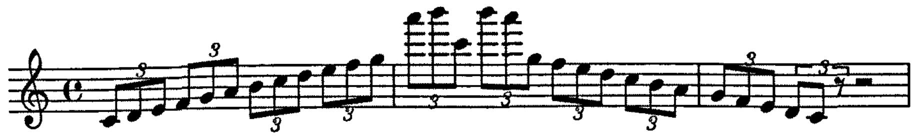
Comparator Summary:

```
Notes found:
Total Correct  22  91.67%
Total wrong     2  8.33%
Total Missing   0  0.00%
Total extra     0  0.00%
```

The majority of the notes in this test are in the middle register of the keyboard resulting in small deviations from the MIDI data note length. The two wrong notes are as a result of the failure of the frequency extraction routine to find the fundamental frequency of the note.

7.3.9 Test Nine

This test is the same as Test Two but this time the note length is set to 1/3 of second. Original Score:



Comparator Output - Test Nine											
No.	Orig	Comp	Output	Timing	Length	No.	Orig	Comp	Output	Timing	Length
1	<	<	<	0.03	0.21	16	_	_	_	0.04	-0.01
2	>	>	>	0.03	0.31	17	]	]	]	0.04	0.11
3	@	s	⊗	-	-	18	O	O	O	0.04	-0.04
4	A	A	A	0.04	0.2	19	M	M	M	0.03	0.18
5	C	C	C	0.04	0.21	20	L	L	L	0.03	0.22
6	E	E	E	0.03	0.25	21	J	J	J	0.04	0.21
7	G	G	G	0.03	0.25	22	H	H	H	0.03	0.17
8	H	H	H	0.04	0.2	23	G	G	G	0.03	0.2
9	J	J	J	0.04	0.22	24	E	E	E	0.03	0.28
10	L	L	L	0.04	0.16	25	C	C	C	0.03	0.3
11	M	M	M	0.03	0.21	26	A	A	A	0.03	0.17
12	O	O	O	0.04	0.17	27	@	@	@	0.04	0.2
13	]	]	]	0.03	0.12	28	>	>	>	0.03	0.38
14	_	_	_	0.03	-0.01	29	<	>	⊗	-	-
15	T	'	⊗	-	-	-	-	-	-	-	-

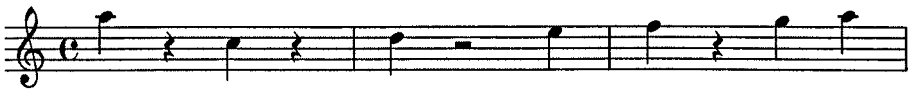
Comparator Summary:

```
Notes found:
Total Correct  26  89.66%
Total wrong     3  10.34%
Total Missing   0  0.00%
Total extra     0  0.00%
```

The fact that the wrong notes occur in different places in the scale (i.e. no correlation between ascending and descending notes) indicates that the error has occured at the frequency extraction stage, rather than the note detection stage.

7.3.10 Test Ten

This test is a series of random notes interspersed with rests. Original Score:



Comparator Output - Test Ten				
Orig	Comp	Output	Timing	Length
Q	Q	Q	0.03	0.26
H	H	H	0.04	0.3
J	J	J	0.04	0.23
L	L	L	0.03	-0.16
M	M	M	0.04	-0.31
O	O	O	0.04	-0.03
Q	Q	Q	0.04	0.28

Comparator Summary:

```
Notes found:
Total Correct  7  100.00%
Total wrong    0  0.00%
Total Missing  0  0.00%
Total extra    0  0.00%
```

7.4 Comparison of Note Detection Systems

The comparator output allows a direct comparison of the PitchTracker with the Note Tracking System described in this thesis. The following table displays the results from both systems (PitchTracker values were taken from the best results for a given test), shown in terms of the number of Correct, Wrong, Missing and Extra notes:

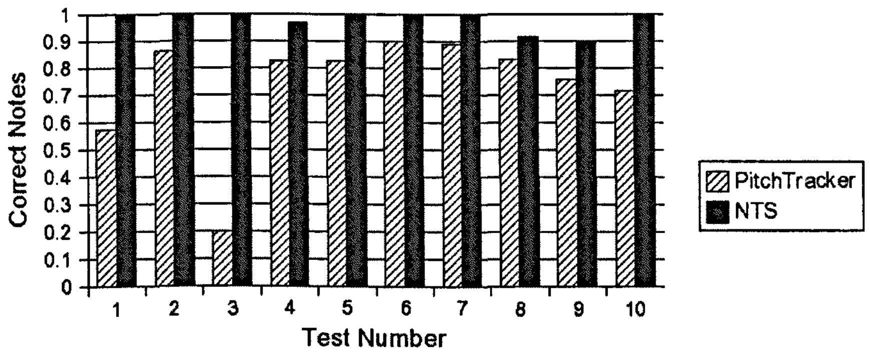


Figure 48: Normalised results for the 10 sytem tests.

	PitchTracker				NTS			
Test No.	C	W	M	E	C	W	M	E
1	4	1	2	0	7	0	0	0
2	25	0	4	0	29	0	0	0
3	1	0	4	0	5	0	0	0
4	24	2	3	0	28	1	0	1
5	24	0	5	0	29	0	0	0
6	9	0	1	0	10	0	0	0
7	8	1	0	0	9	0	0	0
8	20	2	2	3	22	2	0	0
9	22	1	6	1	26	3	0	0
10	5	1	1	0	7	0	0	0

The results of Test Three, a series of five repeated notes of the same pitch, verify the earlier statement that systems which rely on the "steady state" of a note will always underperform when compared to systems which rely on note onsets. Figure 48 shows a direct comparison of the number of correct notes found by both systems. The results were normalised to allow for between test comparison.

When rating performance, the error types assume a level of seriousness. A wrong note is the least severe error as it shows that the detection system has found a note in an appropriate place but was unable to ascertain its correct pitch. It is opined that when considering extra and missing error types, the detection of extra notes is the lesser of two evils (as long as extra notes are not detected to the detriment of genuine notes). That is, a false positive is better than a false negative.

These results show that the Note Tracking System is capable of the identification and subsequent tracking of notes.

7.5 Real Instrument Test Cases

Recordings of an oboe, a violin and a cello played by professional musicians inside an anechoic chamber were made using an AKG C414 B-ULS condensor microphone, set to a cardioid response pattern, onto DAT at 48000Hz. The microphone was positioned approximately 80cm away from the instrument(s) being recorded. In the case of the oboe, the microphone was positioned below the instrument in line with its bell. Test cases were played as consistently[44] as possible in spite of the unnatural conditions inside the anechoic chamber. All three musicians independently commented on the intimacy with their instruments created by the chamber.

Both the violinist and the oboist, whilst having a break from recording, took the opportunity to confirm that it was their perception of the sound that had changed rather than their playing. The test cases used in this section are taken from these recordings.

7.5.1 Oboe Test Cases

The oboe was the first instrument to be recorded. Consequently there are fewer test cases. When presented with a note (approximately 3 seconds long) played on an oboe, the NTS produces the following onset results:

```
TN 1
nN 0
38764 38352 235878 242176 5
20 21 38352 242176
39 40 38896 237568
60 61 38912 234496
80 81 38864 233728
100 101 38800 231424
```

Figure 50 shows where the NTS has identified the start of the note. The first line indicates the earliest starting individual harmonic and the second line indicates where the average of the harmonic's start points occurs.

[44]The tests included repetitive playing of scales at dynamic levels from *pp* to *ff*.

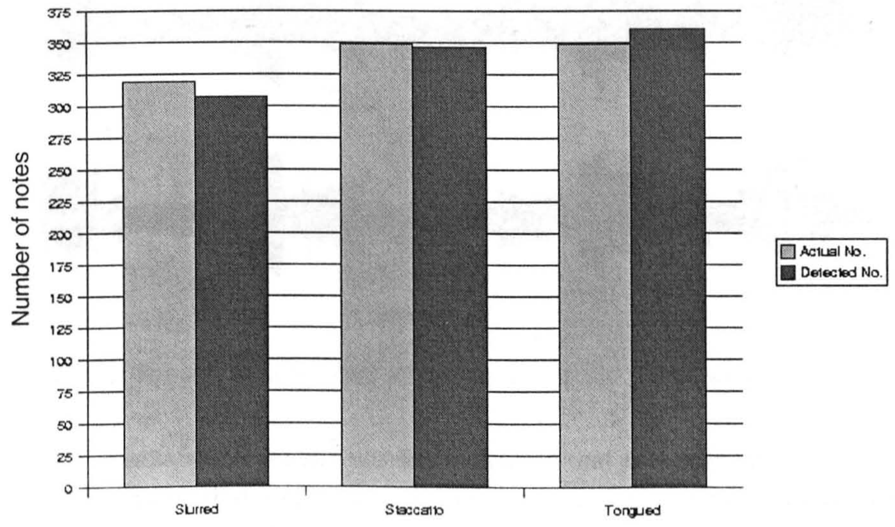


Figure 49: Results of note onset detection for a scales played on an oboe using different methods.

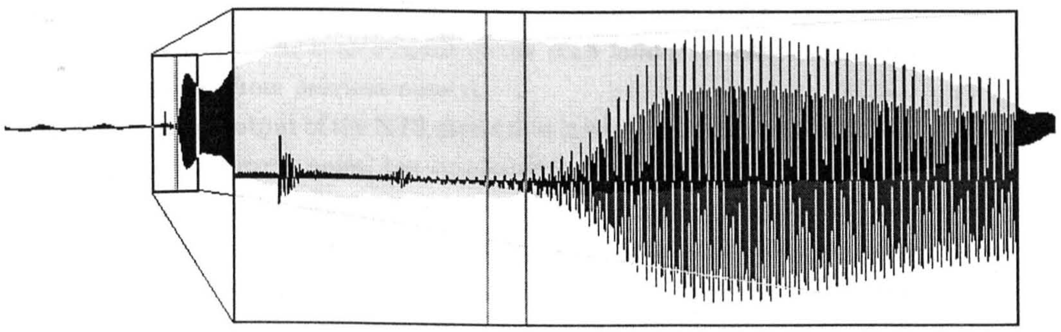


Figure 50: The onset point of a note played on an oboe.

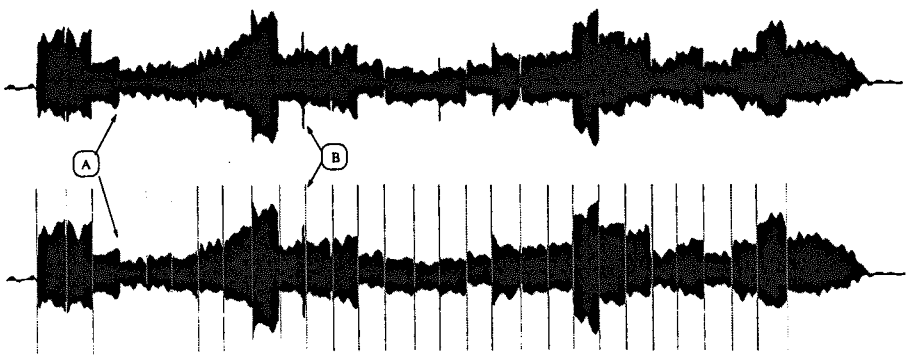


Figure 51: Results of note onset detection for a scale played at *pp* on an oboe.

The results above contain a crude indication that this note has been played with vibrato in that each harmonic has been tracked through two adjacent bins. This is proved when the calculated frequency is examined and shows the modulation of the frequency of the note. This note was used as the example in Section 6.2, Figure 37.

Figure 49 shows how the NTS has performed when processing files that each contain 29 note scales performed on an oboe. Recordings where made from *ppp* - *fff*, the entire dynamic range of the oboe. Figure 51 is an example of why the system has under-detected the amount of notes for slurred scales. This scale was was played at the dynamic level of *pp*. The slurred playing style causes each note to run into the next, making onset detection non-trivial. The error, highlighted by marker A, is as a result of the NTS failing to detect the death point of harmonics from previous note(s).

The debugging output of the NTS shows that it successfully found a potential note onset at the correct point, but discarded it because it was deemed to be a false onset due to it containing harmonics which matched an existing note. Point B has been highlighted to draw attention to the spike in the amplitude envelope. This spike occurs at the end of a note and is not a note onset. In this case the NTS detects the growth in harmonic energy at this point, and correctly identifies it as belonging to an already existing note.

The results for staccato notes are expected to be due to the very fast rise and fall time of the notes; each note in a scale lasts approximately 70m Seconds. Despite this short time, the system was able to detect the onset of 90% of notes in a staccato scale played played at *ppp*. The significance of this is that it is far harder to produce a full bodied note when playing an oboe *ppp*. A human listener, aware that they are listening to a scale played at a certain speed, will anticipate the next note, both in terms of when it should occur and what note it should be, thus giving a contextual advantage over the computer.

7.5.2 Violin Test Cases

A greater number of violin test cases were recorded compared to the oboe.

7.5.2.1 Single legato notes The NTS gave a wide variance in performance when presented with single legato notes (approximately three seconds long) played at different dynamic levels, using both up and down bows, on each open string. The table below shows the number of notes found by the NTS:

String:	G		D		A		E	
	Bow direction (Down/Up)							
Dynamic	D	U	D	U	D	U	D	U
pp	5	11	5	2	6	2	3	1
p	1	6	1	1	1	1	1	1
mp	2	5	1	1	1	1	1	1
mf	1	4	2	1	0	1	1	1
f	2	1	1	3	1	1	2	2
ff	1	6	2	2	1	1	1	5

The variance in results for the G string can be partly explained by the physical shape of the violin and the way it is held. The G string is located on the opposite side of the bridge to the bowing arm. The combination of bow angle and the requirement to play quietly has resulted in the bow bouncing on the string[45]. Bounces of the bow cause a change in dynamic level which causes the NTS to detect repeated notes. Other repeated notes are caused by the efficiency of the harmonic tracking routine assigning a stop point to a harmonic which has been absent from a few frames of data. Methods for overcoming this problem are discussed in Section 10.2. Despite these problems the NTS has still managed to identify and track 56% of the notes presented to it.

[45]This is particularly true for the down bows because bow control levels diminish from the heel to the tip.

7.5.2.2 A series of repeated notes Using open strings a series of 16 repeated notes were recorded at different dynamic levels. This presents a non-trivial test for the NTS as each note overlaps the following note. The note repition rate was approximately 5Hz. The number of notes found by the NTS is shown below:

	String			
Dynamic	G	D	A	E
pp	15	17	17	15
p	16	18	14	14
mp	15	17	15	12
mf	18	14	14	11
f	17	18	17	11
ff	12	15	16	14

Speed of playing and dynamic level account for the mssing notes on the E string. When listened to the louder test cases are particularly "scratchy" meaning the string has not been allowed to settle and vibrate in a steady manner. For the reasons dicussed at the end of Chapter Five, these results could be misleading in that they convey no information as to whether the identified notes are correct. The comparator could only be used to verify these results if a MIDI file with similar timings was generated as the evaluation of repeated notes is only possible when timing information is used.

7.5.2.3 Scales and arpeggios The scale of G major 3 Octaves (43 notes) and its arpeggio (19 notes) were played at two dynamic levels (pp and ff) using a variety of bowing styles. The number of notes found by the NTS for each scenario is shown below:

	Scale		Arpeggio	
Playing style	*pp*	*ff*	*pp*	*ff*
Legato	no data	38	21	20
Staccato	43	43	19	19
Slurred	41	43	19	21
Plucked	40	40	18	19

The results for slurred bowing are particularly impressive when the speed the scale was played (43 notes in just over 7 seconds ~ 6 notes a second) is taken into account. Visual inspection of the slurred-*ff* result showed that of the 43 notes reported, 41 were correct and two were false detections meaning that two genuine notes had been missed.

As a scale contains sequential notes of differing pitch it is possible to generate a Midi file of the scale and use a pitch only comparator to evaluate the frequency detection performance of the NTS. Comparison showed that for the staccato-*ff* scale there were 25 correct notes and 18 wrong notes. Examination of this disappointing result showed that of the 18 wrong notes, 12 were octave errors. Further investigation showed that whilst the fundamental frequency was presented to the pitch extraction routine it was not selected. With the correction of this error, the system would therefore report 37 correct notes and 6 wrong notes. The remaing wrong notes are as a result of the NTS failing to actually detect the fundamental frequency. In these cases it was found to have a very low level of spectral energy compared to higher order harmonics.

As the aim of this work is gestural extraction, this pitch extraction error remains the subject of further work.

7.5.3 Cello Test Cases

The least number of recordings were made for the cello due to the cellist's busy schedule and time constraints. The first seven notes of a scale were played in a number of different playing styles using different dynamic levels. The following table shows the number of notes detected for each test case:

	Dynamic	
Playing style	*p*	*f*
Legato	7	7
Staccato	7	7
Spiccato	7	9
Plucked	7	8

The extra notes are caused by the NTS prematurely detecting the end of note harmonics.

The following table shows the number of notes found by the NTS when presented with the scale C major 3 Octaves (43 notes) and its arpeggio (19 notes) played at two dynamic levels (pp and ff) using a variety of bowing styles:

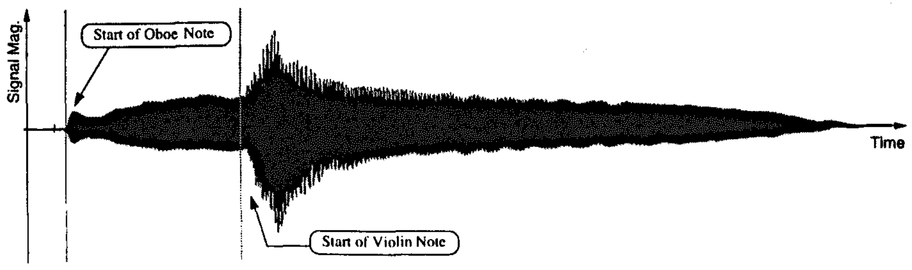


Figure 52: Simultaneous oboe and violin (plucked) notes and their start points.

	Scale		Arpeggio	
Playing style	*p*	*f*	*p*	*f*
Legato	no data	no data	26	no data
Staccato	43	43	19	20
Slurred	73	73	32	33
Plucked	43	44	no data	no data

7.5.4 Polyphonic Test Cases

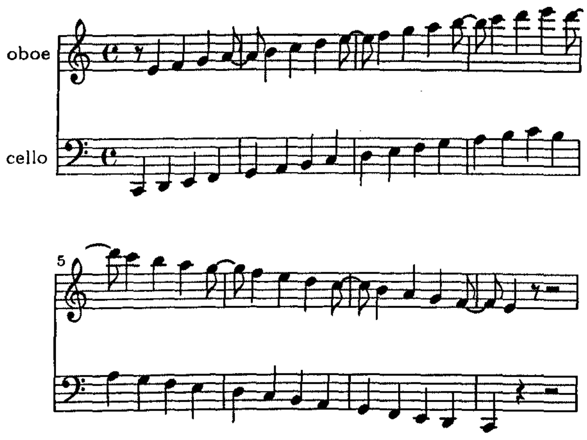
7.5.4.1 Oboe and Violin These two notes were mixed so that the notes overlap, as shown in Figure 52. The NTS produces the following results:

```
TN 2
nN 0
39126 38352 236953 246272 5
20 21 38352 233984
39 40 39408 234752
60 61 39424 235776
80 81 39408 233984
100 101 39040 246272
nN 1
129563 129504 238994 248576 7
12 13 129792 248576
24 25 129504 233984
31 32 129520 247296
50 51 129536 233984
74 75 129536 236288
87 88 129536 236032
114 115 129520 236800
```

The system finds the same number of harmonics for the oboe (see results in Section 7.5.1) and finds two less for the violin (compared to analysis of the same violin note played in isolation). This is because when the signal is mixed the resultant wave is normalised - thus weaker harmonics become even weaker, making them more difficult to detect. The detection of the oboe notes' harmonic death points is affected by the presence of the release stage of the simultaneously sounding note.

7.5.4.2 Oboe and Cello

The following music was played using different combinations of style and dynamics.



A two octave scale contains twenty nine notes.

Test	Dynamic	Oboe Gesture	Cello Gesture	Total Notes	Detected Notes
1	P	Staccato	Staccato	58	56
2	F	Staccato	Staccato	58	57
3	P	Slurred	Slurred	58	66
4	F	Slurred	Slurred	58	61
5	P	Tongued	Legato	58	66
6	P	Slurred	Plucked	58	53
7	F	Slurred	Plucked	58	54

As expected the staccato results are the most accurate. These results show that the NTS can track notes in a realistic polyphonic scenario. Further analysis is required to determine the breakdown of correctly detected notes on a per instrument basis.

7.5.5 Evaluation of Results

- The ten comparator based tests shows that the NTS outperforms the PitchTracker.
- The NTS correctly identifies 97% of the notes presented to it.
- The PitchTracker identifies 80% of the notes presented to it.
- The NTS copes with "difficult" test cases that by nature the PitchTracker cannot detect.
- The NTS performs well with polyphonic sources.
- Certain playing styles on real instruments presented the NTS with data that deceived its note tracking routines. Improvements are recommended and discussed in Section 10.2.

The overall results show that the NTS is an appropriate system to take forward into the gestural analysis arena.

8 Gestural Analysis

This section shows that it is possible to extract information from audio signals which leads to the identification of a gesture. The work presented in Sections 8.1 & 8.2 are exemplars for further work.

8.1 String Instrument Bow Gestures.

This section considers different aspects of a note from a stringed instrument which can be used to extract gestural information relevant to it.

8.1.1 Plucked and Bowed notes

Table 9 provides a summary of the basic string instrument excitation gestures and it can be seen that plucked and staccato notes have the most similar discriminating features. Figure 53 shows the measurements from which the following information can be calculated (Poli *et al.*[72] performed similar measurements on the amplitude envelope of a note, rather than its harmonics):

- Position, in time, of the occurence of the peak of the harmonic[46]
- The gradient of the growth of the harmonic.

In addition the NTS provides information regarding the number of (in)harmonics present in a note. Figure 54 shows spectral plots for the beginning of plucked and bowed (spiccato) notes. As Figure 54 shows, the bowed note contains a number of short-lived (in)harmonics which are not present in the plucked note.

It is anticipated that these three pieces of information will provide a means of distinguishing not between these two gestures alone, but between the four gesture types shown in Table 9.

8.1.1.1 Extraction of violin excitation gestures Sample populations of 43 staccato, 42 slurred, 36 legato and 41 plucked violin notes were used to determine the mean value (corrected for statistical outliers) of each discriminator for each gesture type. The NTS was modified so that having performed the second sweep of audio data, it would analyse the start of each note and compare it to the average values for each gesture type. An accumulative measure of absolute difference was used to determine the similarity of a note to a given

[46]This is the only discriminator which is common to those used by Martin and Kim[58] in their musical instrument identification system.

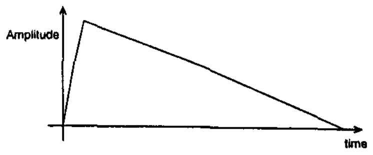
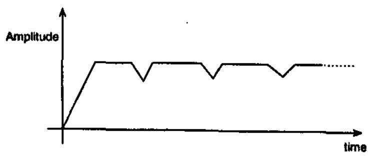
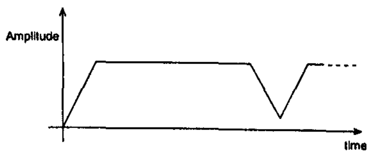
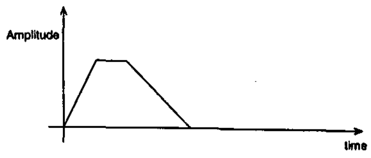
Gesture	Discriminating Features	Ideal Amplitude Waveform
Pluck	• Fast growth of note • Gradual decay • No bow noise	
Slurred	• Slower growth rate compared to a plucked note • Gradual decay similar to plucked note • Bow noise present	
Legato	• Initial growth rate similar to a slurred note • subsequent growth rates different from initial growth • Fast decay of note • More bow noise in first note than subsequent notes	
Staccato	• As legato but much shorter. • discernable gap between notes	

Table 9: A summary of string instrument excitation gestures.

Test Type	Gesture	Staccato	Slurred	Legato	Plucked	Unknown	Total
S-*p*	Staccato	19	**22**	0	1	1	43
S-*f*	Staccato	**27**	16	0	0	0	43
S-*p*	Slurred	9	**22**	0	10	0	41
S-*f*	Slurred	13	**21**	0	8	1	43
S-*f*	Legato	1	3	**31**	1	2	38
A-*p*	Staccato	**10**	9	0	0	0	19
A-*f*	Staccato	**15**	4	0	0	0	19
A-*p*	Slurred	6	**9**	0	4	0	19
A-*f*	Slurred	**11**	4	0	6	0	21
A-*p*	Legato	4	3	**13**	1	0	21
A-*f*	Legato	**8**	3	**8**	1	0	20
S-*p*	Plucked	1	3	0	**35**	1	40
S-*f*	Plucked	0	2	0	**38**	0	40
A-*p*	Plucked	0	3	0	**12**	0	15
A-*f*	Plucked	0	1	0	**17**	1	19

Table 10: Showing the results for automatic detection of the type of excitation gesture used to play a violin.

gesture type. Table 10 shows the results for a plucked and bowed violin scales (G major three octaves, 43 notes - denoted by an 'S') and arpeggios (G major 3 octaves, 19 notes - denoted by an 'A').

The primary aim of this gestural extraction method was to enable the NTS to differentiate bowed and plucked notes. For the plucked note tests, 89% of notes were correctly identified as the result of a pluck excitation gesture. For the bowed note tests, 89% of notes were correctly identified as the result of a bowed excitation gesture.

The most suprising result is the similarity bewteen staccato and slurred gesture types. When staccato notes deviate from the average measurements they are judged to be more like slurred notes, whereas when slurred notes deviate from their average type they exhibit characteristics associated with both staccato and plucked gesture types. Another interesting result is that when plucked notes deviate from their average type they are judged to be more like slurred notes. The reason for this is discussed below.

In terms of the bowing gestures the sytem is most able to distinguish legato bowing. The violinist was asked to play legato notes using as much bow as possible. The build up of sound from a fast moving bow coupled with the direction changing preparatory action of the wrist, has resulted in the average peak magnitude postion for the legato gesture type to be up to nine times later than values recorded for other gesture types.

Gesture identification could be further improved by sub-setting the characteristics of certain gesture types. Table 9 shows that the onset of the initial note in a series played with either legato or slurred bowing will contain gestural information different from that of subsequent notes. When a series of legato notes are played on the same string notes following a change in bow direction contain less onset noise as the string is already vibrating.

The first[47] note in a series of slurred notes will exhibit similar onset characteristics to other bowing styles. Subsequent notes played by the same bow stroke will not contain bow onset noise. This accounts for the judged similarity between plucked and slurred notes. The note that immediately follows a change of bow direction will exhibit similar characteristics to legato notes. The results in Table 10 support this notion.

These results would display higher levels of discrimination if test notes had been hand picked to ensure that the NTS was evaluating a correctly identified note. Twenty four individually recorded staccato notes[48] played on open strings at different dynamic levels were analysed by the NTS system. Checks were made to ensure that each note had been properly identified. The gestural discrimination routine reported that seventeen notes were produced using staccato bowing and the remaining seven using legato bowing. This result is significant as at first it seemed to be contrary to the patterns shown in Table 10 as a staccato note is never identified as a legato note. However, when the seven "legato" notes were listened to, it was found that they sounded like legato notes, despite the specific instructions given to the violinst.

These results show that note detection techniques which depend upon "steady state" note harmonics would miss the vital information before point B in Figure 54. This significance of this point is discussed in Section 10.4.

8.1.1.2 Extraction of cello excitation gestures Recordings of the cello were analysed using the NTS without making any modifications to the average discriminator values for each gestural type. Table 11 shows the results of these tests. The gestural extraction routine has succesfully identified staccato and plucked test cases. In addition to the result for the plucked C major scale, it was noted emprically that plucked notes were consistently recognised as slurred

[47]In other words, the string to be played is at rest.

[48]The instruction given to the violinist was "Accented - attack the string"

Test Type	Gesture	Staccato	Slurred	Legato	Plucked	Unknown	Total
7 ascending notes	Staccato	7	0	0	0	0	7
7 ascending notes	Legato	2	4	0	1	0	7
7 ascending notes	Plucked	0	0	0	7	0	7
C major 3 octaves	Plucked	0	10	0	34	0	44
C major 3 octaves	Staccato	39	3	0	0	1	43

Table 11: Showing the results for automatic detection of the type of excitation gesture used to play a cello.

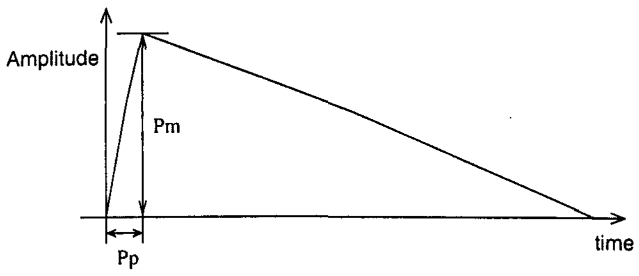


Figure 53: Showing the measurement of different harmonic attributes.

notes. Unfortunately further legato test cases were unavailable.

8.1.1.3 Extraction of cello gestures from polyphonic recordings This section tests whether the NTS is capable of correctly detecting plucked, staccato or slurred notes played on a cello whilst an oboe holds a note for the duration of the cello notes. In each case the oboe was judged to be using a "slurred" playing style. The results shown in Table 12 show that the NTS can correctly process gestural information from a polyphonic signal. The last test is particularly encouraging in that the seven ascending notes were scored to be played as two staccato notes followed by three slurred notes finishing the sequence with two further staccato notes. The NTS dectected gestures in the order shown in Table 13.

8.2 Oboe Key Presses

The correct position in time of a key press is essential to produce a "clean" note. When playing a series of notes in a non-legato sytle, changes in fingering should ideally occur in the interval between notes. If the change in fingering occurs too

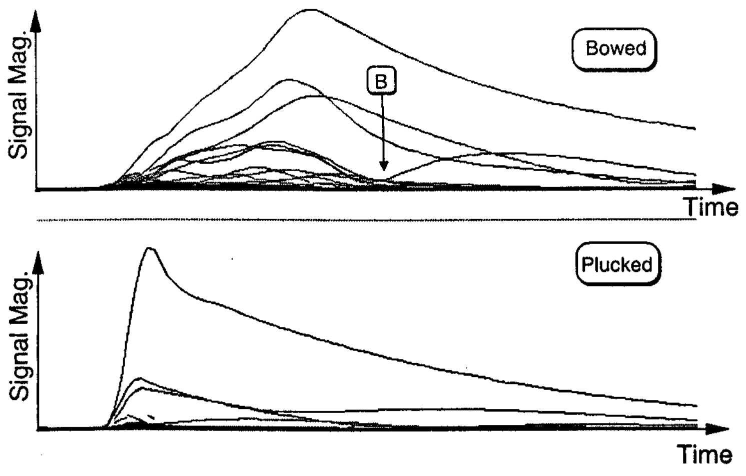


Figure 54: Showing the difference between a bowed (spiccato) and plucked note.

Test Description	Gesture	Staccato	Slurred	Legato	Plucked	Unknown	Total
Oboe:1 long note cello:7 ascending notes	Staccato	6	2	0	0	0	8
Oboe:1 long note cello:7 ascending notes	Plucked	1	2	0	5	0	8
Oboe:1 long note cello:7 ascending notes	Slurred	1	7	0	4	0	12
Oboe:1 long note cello:7 ascending notes	Slurred/ Staccato	4	2	0	2	0	8

Table 12: Showing the results for automatic detection of the type of excitation gesture used to play a cello.

Note	NTS output
1	Slurred (oboe)
2	Staccato
3	Staccato
4	Slurred
5	Plucked
6	Plucked
7	Staccato
8	Staccato

Table 13: Showing the order of detected gestures.

early then the end of the currently sounding note will be clipped. Changing the fingering too late results in the harmonics of the note taking longer to establish themselves. A key press "click" in the correct place can often be heard. Its presence is the equivalent of the screech heard when changing position on a steel strung guitar, or the actual noise of the bow heard in the upper registers of a violin. The click is heard and considered to be a member of the set of sounds belonging to an oboe.

Extraction of this key click gesture could lead to the implementation of a practice aid whereby the "cleanness" of notes could be indicated.

8.2.1 Analysis of the key press

As well as being very low in spectral energy, key press events are very short. The following table shows manual timings taken from audible key presses between the notes of a staccato scale:

Click number	Length (ms)	Click number	Length (ms)
1	23.22	7	25.76
2	10.16	8	14.51
3	8.71	9	15.24
4	19.8	10	12.7
5	16.33	11	9.07
6	15.96	12	7.26
Average:		14.89 ms	

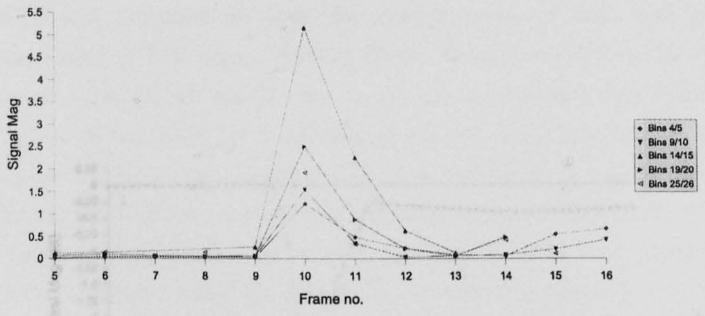


Figure 55: Showing the spectral content of the first click in Figure 11.

Observation of a heavily zoomed in waveform shows that values for the longer clicks are due to a flam[49] like effect of two separate clicks.

Figure 11 in Chapter four showed an isolated key press event that took place before the note. Similar key presses can be observed between isolated notes in a staccato scale. Therefore it is the actual press of the key that generates the sound of the click, which resonates in the body of the oboe[50]. Thus each click has a percussive like pitch which is relative to the key pressed. When performing FFT analysis of a waveform there is usually a compromise between between frequency and temporal resolution. However, due to the nature of a key press, this compromise is an advantage. A short time frame is needed in order to detect the click. The wide frequency range of each bin ensures that the spectral energy of the click contributes to a smaller number of bins, rather than being spread across many bins.

Figure 55 shows a plot of LoPs constructed to cover the section of the wave containing the first click in Figure 11. Figure 56 shows the plot of a LoP corresponding to the interval between two notes, from a tongued scale, which does not contain a click from a key press.

Figure 57 shows the plot of a LoP which covers the period between two notes. The key click appears appears as a spike between the notes. The data used in Figure 57 comes from LoPs created from data that was generated using a frame size of 512 bins.

[49]A flam is defined by Buddy Rich[75] as "*a principle (large) note, preceded by a grace note*".

[50]String players make use of a similar effect, by firmly striking a string with an appropiate finger, to check their tuning before playing a note.

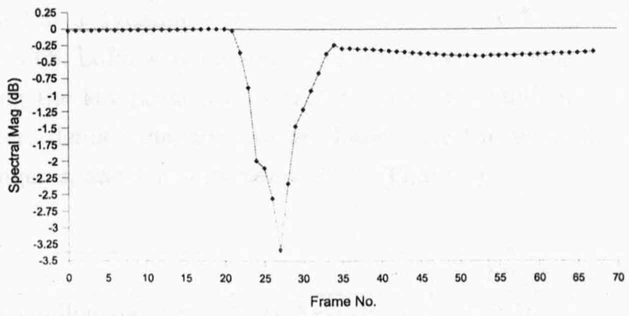


Figure 56: Showing a gap between notes with no key press click

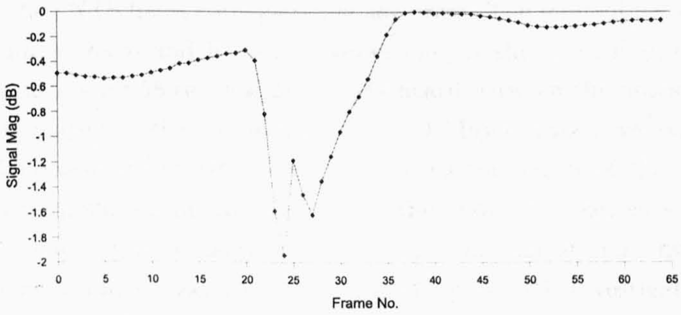


Figure 57: Showing the appearance of a key press click between two notes.

The NTS was modified so that the second sweep of data was performed with a frame size of 512 bins. Having found the peak magnitude of a given note harmonic, the search routine for a key press click searches back towards the start point of the note for a turning point. A recording of the scale of D Major tongued at a dynamic level of *ppp* over two octaves was analysed using the modified NTS. LoPs were created using note start and stop data which were searched using the key press click search routine. It found the following clicks (Capital letters denote the first octave, lower case the second, and Đ top D. Px=Press finger x, and Lx=Lift finger x. T=Thumb):

		Fingering Difference	
Note number following click	Note Transition	Left Hand	Right Hand
0	D	T, P1,P2,P3	P1,P2,P3
1	D→E	-	L3
14	c#→Đ	-	P2, L1
18	a→g	-	P1
21	e→d	T, P3	-
23	C#→B	P2	-
25	A→G	-	P1
26	G→F#	-	P2

Apart from the D→E transition of note one, each key press click identified by the modified NTS has a corresponding key press. The interval between notes D (note number zero) and E (note number one) is shown in Figure 58. The click shown in Figure 58 can also be clearly heard between the notes when the waveform is played. Other recordings of the D Major scale were examined to rule out the possibility of extraneous noise being the source of the click. The click arises from the actual construction of the oboe. The keywork system of the oboe is made up of a complex lever and rod mechanism (the evolution of which is described by Goosens and Roxburgh[42, p.17-27]). Investigation of this mechansim showed that keywork system enables finger 2 of the right hand to hold down a pad at the back of the oboe. When finger 3 is lifted to play the E, the mechanism held in place by finger 2 prevents the key associated with finger 3 from returning to its natural resting place, resulting in a metallic click as it hits the mechanism of the oboe. This "click" would obviously not be present if the scale had been played on a classical oboe which has no keywork.

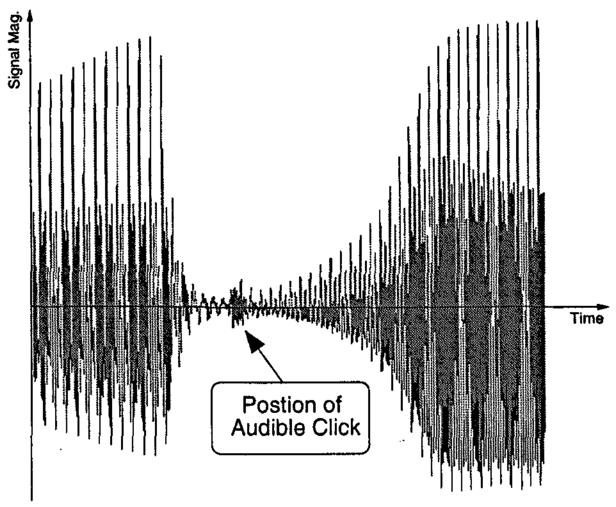


Figure 58: A click between two notes.

The detection of key presses inbetween notes is an indication of good coordination between the hands and mouth. The above table is not exhaustive as it does not include early key presses as shown in Figure 12, Section 4.1.1.1. In some instances the key press is gentle enough to go undetected. Further work is required to determine whether a classical oboe would actually produce the required auditory infromation.

The current implementation contains a built in level of robustness that prevents noise peaks from being identified as key clicks. LoPs are only searched for a key press event if they exhibit growth characteristic of the start of a note.

8.2.2 Towards the detection of an early key press

An early key press is characterised by a change in frequency at the end of a note. The method for calculating the instantaneous frequency of a spectral peak (described in Chapter six) relies on the assumption that only one harmonic contributes to the spectral peak in a given bin. As the magnitude of a harmonic decreases this assumption breaks down and the spectral peak in a bin becomes affected by noise and artefacts. The outcome of this is that for signals of low magnitude the calculated frequency is incorrect. Thus the calcluated frequency cannot be used to detect an early key press.

Figure 59 shows two LoPs that were constructed from data that was gener-

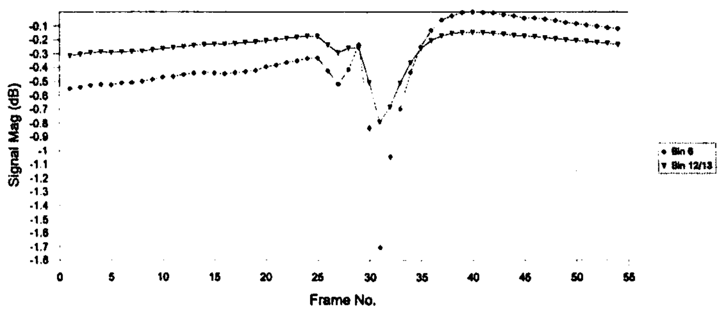


Figure 59: The effect of a key press click has on the decay of the harmonics of a note.

ated using a frame size of 512 bins. The characteristic spike associated with a key press occurs in the decay of the harmonics of the note. It has been observed that this spike often occurs in conjunction with a bin hop. Thus, the presence of the key press spike and a bin hop are two factors that would enable detection of early key press events. A third factor would be to look for a key press spike (which belongs to the second note) occuring in the release section of the first note.

Such events would be detectable on both modern and classical oboes.

8.2.3 Chapter conclusions

- Initial tests indicate that one set of generic discriminators can be constructed for gestures common to the violin and the cello.
- Oboe key presses have successfully been detected. Analysis of a recording of the D major scale showed that extracted key presses coincided with finger changes throughout the scale.

9 Conclusion

The aim of this research was to extract gestural information from musical audio signals. Work centred around the development of a Note Tracking System (NTS) which was modelled on features of the human auditory system. This approach has produced an effective method of mining note attributes. Proposed definitions for note features were justified by their continued use throughout this research.

The comparator designed to validate the output of this NTS was first itself validated using simple tests analgous to music and found to be appropriate for this use.

The output of the NTS was then validated using this bespoke comparator and in almost all cases the NTS was accurate in determination of pitch and timing. It outperformed published PitchTracker software in all test cases. The vast amount of note information it can command makes it a suitable system for retrieval of information about gestures that influence sound.

Gestures have been defined in a logical manner identifying the pertinent type of gestures for this work.

The oboe key press detection method shows that it is possible to extract articulatory key press timing information from an acoustic signal. This method could be developed as a tool to improve musical technique.

The system also has the potential to determine in real situations the manner in which a note has been played on a stringed instrument. Initial steps were taken to find generic discriminators that would work for both the violin and the cello that would permit a system to identify the type of gesture used. Preliminary tests show that not only was the system able to distinguish between plucked and bowed notes, it was also able, in the case of bowed notes, to determine the type of bowing used. This indicates that a set of generic discriminators does exist.

The system can extract string and oboe based gestures with a promising degree of success, showing that it is possible to extract gestures from musical audio signals.

10 Further work

10.1 Comparator

10.1.1 Efficiency

The functional requirement of this comparator to compare a given error with the remainder of the string it occurs in, results in computational overhead. In a formal language context, lookahead functionality could be legitimately restricted using an abstract top-down contextual approach. Text would be grouped according to the abstraction levels provided by paragraphs, sentences, words and characters. Thus the length of the lookahead window for a given level would be bound by the level of abstraction above it. Each level of abstraction is separated by different delimiters: paragraphs by carriage returns, sentences by punctation marks and words by spaces.

Comparision would begin at the top level comparing paragraphs with paragraphs. If there is a difference each level of abstraction would be used to determine if the error was caused by a missing paragraph, sentence, word or character. For example, if a character was missing from a word, information from the comparison of words would be used to restrict the lookahead distance when finding the missing character(s) of the word.

The equivalent in a musical context would be movements, phrases, bars and notes. Difficulties would arise in terms of musical representation: for example, determining where a phrase begins and ends.

In order to make comparisons at each level of abstraction, it would be necessary to perform some form of encoding operation to generate a unique tag (based on content and order) for each item within a level. The letters in a word would be used to generate a tag for each word. The tags from the words in a sentence would be used to generate the sentence tag, and so on.

10.1.2 Musical representation

Development of a musical type language would enable the comparator to process scores and performances containing musical objects such as chords or embelishments. Using the unique tag concept from the previous section, chords could be compared without the need for the complex parallel and sequential temporal structures as described by Heijink *et al*[47].

10.1.3 Timing

The problems associated with note length timing show that the MIDI file format is not ideal basis for note length comparisons. This is due to the manner in which the note-off event is used. In the original MIDI file it is used to indicate the point at which a note (if it has not already done so) can begin its decay. The comparison MIDI file uses the note-off event to indicate the actual end of a note. Some sort of compensation function is also needed to account for instruments (e.g. the piano) when note length is proportional to pitch.

10.2 Note Tracking system

10.2.1 Adaptive streaming

When a note is found the onset slope for each harmonic could be characterised and used to distinguish between dynamic fluctuation of a harmonic and a genuine repeated note. This would result in a dramatic reduction in the number of falsely detected notes.

10.2.2 Heuristic Partial Tracking

At present, gaps in a LoP are considered to be an indication of the end point of a partial. A better system would take the "strength" of the partial into account, whereby a strong partial would be judged to be able to sustain itself over a gap in a LoP. The maximum gap length would be proportional to the strength of the partial. Factors that contribute to the strength of the partial would include the number of items in the LoP before the gap and the gradient of the LoP leading up to the gap. This would reduce the number of falsely detected onsets which arise from gaps in LoPs which prematurely end the life of a partial.

10.3 Gestural Extraction

10.3.1 Oboe gestures

This thesis has laid a foundation upon which further research can be built. At present the oboe key click extraction routine only uses note start information and tracks backwards to find key clicks. Further work would provide the routine with the means of using both note end and start points so that it can analyse the gap between notes and find:

- Early key presses which cause the pitch of the next note to be heard before the current note has finished sounding.
- Key presses which occur inbetween notes.
- Late key presses which occur after the onset of the note. Such presses have been empirically observed.

Such work would also provide information that would go towards extracting articulatory oboe gestures (e.g. legato/staccato playing styles) that are the equivalent of the string instrument gestures. A new set of discriminators would be needed; it is invisaged that necessary information would be provided by:

- the note onset;
- note length;
- inter-note interval.

Detection of the note onset would be particulary important when distinguishing between tongued and slurred playing styles.

10.4 String instrument gestures

Distinctions drawn between different gestures used to sound these instruments can be improved by including more discriminators in the analysis. The NTS sytem provides details of the end points of the harmonics of a note. This information can be used to calculate:

- Inter-note intervals and inter-note onsets.
- The gradient from the peak magnitude position of a harmonic to its end point.
- The peak magnitude position of a harmonic relative to its end point.

Implementation of the above may not improve the accuracy of cello gesture identification and would demonstrate that the fuller sound of the cello does not easily conform to note parameters established from violin notes. Such a result would justify the need for separate discriminator values for the violin and cello.

Further work is needed to investigate point B in Figure 54. A number of short lived (in)harmonics come to an end at this point, though one rises from this point. Given that the note in question was played using spiccato bowing, point B may be the point at which the bow was lifted from the string. The detection of this gesture would provide a further discriminator for the distinction of bow types.

It is suggested that examination of the magnitude of the (in)harmonics that are seen to rise and fall from the onset to point B would provide information concerning the amount of bow force exerted to create the note, thus emulating the bow force measurements made by Askenfelt[3].

Further work is also required to investigate whether the oboe key press routines can detect lack of coordination in string players when using legato bowing. Failure to change the left hand fingering in time with a change of bow direction would result in the same clipping effect at the end of a note.

References

[1] Gnu lilypond - the music typesetter. http://www.lilypond.org/.

[2] V. Algazi, R. uda, R. Morrison, and D. Thompson. Structural composition and decomposition of hrtfs. Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, pp103-106, 2001.

[3] Anders Askenfelt. Measurement of bow motion and bow force in violin playing. *Journal of the Acoustical Society of America*, 80(4):1007–1015, October 1986.

[4] J. Backus. *The Acoustical Foundations of Music.* W.W. Norton & Company Inc, New York, 1969. p 85.

[5] A. H. Benade. *Fundamentals of Musical Acoustics.* Dove Publications Inc, 31 East 2nd Street, Mineola, N.Y., 2nd edition, 1990. pp 223-225.

[6] Kenneth W. Berger. Some factors in the recognition of timber. *The Journal of the Acoustical Society of America*, 36(10):1888–1891, 1964.

[7] Uzay Bora, Selmin Tufan, and Semih Bilgen. A tool for comparison of piano performances. *Journal of New Music Research*, 29(1):85–99, 2000.

[8] Robert S. Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[9] Annelies Braffort, Rachid Gherbi, Sylvie Gibet, James Richardson, and Daniel Teil, editors. *Gesture-Based Communication in Human-Computer Interaction, International Gesture Workshop, GW'99, Gif-sur-Yvette, France, March 17-19, 1999, Proceedings*, volume 1739 of *Lecture Notes in Computer Science.* Springer, 1999.

[10] A. Bregman. *Auditory Scene Analysis.* Cambridge: MIT Press, 1990.

[11] Roberto Bresin and Giovanni Umberto Battel. 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.

[12] Judith Brown and Miller Puckette. A high resolution fundamental frequency determination based on phase changes of the fourier transform. *Journal of the Acoustical Society of America*, 94(2):662–667, 1993.

[13] Claude Cadoz, Annie Luciani, and Jean Loup Florens. Cordis-anima: A modeling and simulation system for sound and image sythesis - the general formalism. *Computer Music Journal*, 17(1):19–29, 1993.

[14] Antonio Camurri, Shuji Hashimoto, Matteo Ricchetti, Andrea Ricci, Kenji Suzuki, Riccardo Trocca, and Gualtiero Volpe. Eyesweb: Toward gesture and affect recognition in interactive dance and music systems. *Computer Music Journal*, 24(1):57–69, 2000.

[15] D. Cirotteau, D. Fober, S. Letz, and Y. Orlarey. Un pitchtracker monophonique. *Actes des Journes d'Informatique Musicale JIM2001*, pages 217–233, 2001.

[16] Jean claude Risset and David L. Wessel. *The Psychology of Music*, chapter 5: Exploration of Timbre by analysis and synthesis, page 118. Academic press, 2nd edition, 1999.

[17] Martin P. Cooke and Daniel P.W. Ellis. The auditory organization of speech in listeners and machines. Technical Report TR-98-016, International Computer Science Institute, June 1998.

[18] D. Cooper and N. Bailey. Perceptually smooth timbral guides by state-space analysis of phase vocoder parameters. *Computer Music Journal*, 24(1):32–42, 2000.

[19] Sofia Dahl. The playing of an accent - preliminary observations from the temporal and kinematic analysis of percussionists. *Journal of New Music Research*, 29(3):225–233, 2000.

[20] Roger B. Dannenberg. Music understanding by computer. *Carnegie Mellon School of Computer Science*, pages 19–28, 1987/1988.

[21] Roger B. Dannenberg. *Music, Language, Speech, and Brain*, chapter Recent Work in Real-Time music Understanding by Computer. Macmillan, 1991.

[22] C.J. Darwin. Perceiving vowels in the presence of another sound: Constraints on formant perception. *The Journal of the Acoustical Society of America*, 76(6):1636–1647, 1984.

[23] C.J. Darwin and Valter Ciocca. Grouping in pitch perception: Effects of onset asynchrony and ear of presentation of a mistuned component. *The Journal of the Acoustical Society of America*, 91(6):33–81, 1992.

[24] D. Deutsch, editor. *The Psychology of Music.* Academic Press Inc. (London) Ltd, 24/28 Oval Road, London, NW1 7DX, 1982. pp 4-6.

[25] Diana Deutsch, editor. *the Psychology of Music*, chapter 5, Exploration of Timbre by analysis and synthesis. Academic Press, 2 edition, 1999.

[26] Diana Deutsch, editor. *the Psychology of Music*, chapter 4, The Perception of Musical Tones. Academic Press, 2 edition, 1999.

[27] Roberto Dillon. Classifying musical performance by statistical analysis of audio cues. *Journal of New Music Research*, 32(3):327–332, 2003.

[28] Diana Treffry (Editorial Director), editor. *Collins Concise Dictionary.* Harper Collins, 4th edition, 1999.

[29] Simon Dixon. Extraction of musical performance parameters from audio data. In *IEEE Pacific-Rim Conference on Multimedia*, pages 42–45, 2000.

[30] Simon Dixon. On the computer recognition of solo piano music. *Mikropolyphonie*, (6), 2000.

[31] Simon Dixon. Automatic extraction of tempo and beat from expressive performances. *Journal of New Music Research*, 30(1), 2001.

[32] Shlomo Dubnov. Emotion - is it measurable? In *KANSEI - The Technology of Emotion, AIMI International Workshop*, July 1997.

[33] Chris Duxbury, Mark Sandler, and Mike Davies. A hybrid approach to musical note onset detection. In *Proceedings of the 5th International Conference on Digital Audio Effects*, Hamburg, Germany, September 2002.

[34] A. Eronen and A. Klapuri. Musical instrument recognition using cepstral coefficients and temporal features. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2000.

[35] Paul R. Farnsworth. *The social Psychology of Music*, chapter Chapter Four: Melody, page 57. the Iowà State University Press, 1969.

[36] Harvey Fletcher, E. Donnell Blackman, and O. Norman Geertsen. Quality of violin, viola, 'cello, and bass-viol tones.i. *The Journal of the Acoustical Society of America*, 37(5):851–863, 1965.

[37] Harvey Fletcher and Larry C. Sanders. Quality of violin vibrato tones. *The Journal of the Acoustical Society of America*, 41(6):1534–1544, 1967.

[38] A. Fraser and I. Fujunaga. Toward real-time recognition of acoustic musical instruments. In *Proceedings of the International Computer Music Conference*, 1999.

[39] Daniel J. Freed and William L. Martens. Deriving psychophysical relations for timbre. In *Proceedings of the 1986 International Computer Music Conference.*, pages 393–405. San Francisco: International Computer Music Association, 1986.

[40] Anders Friberg, Johan Sundberg, and Lars Frydén. Music from motion: Sound level envelopes of tones expressing human locomotion. *Journal of New Music Research*, 29(3):199–210, 2000.

[41] Alf Gabrielsson. *Generative Processes in Music*, chapter 2. Timing in music performance and its relations to music experience, pages 27–51. Clarendon Press, 1988.

[42] Leon Goosens and Edwin Roxburgh. *Yehudi Menuhin Music Guides: Oboe.* MacDonald and Jane's, 1977.

[43] John W. Gordon. The perceptual attack time of musical tones. *Journal of the Acoustical Society of America*, 82(1):88–105, July 1987.

[44] J. Grey. *An Exploration of Musical Timbre.* PhD thesis, Stanford University, 1975.

[45] John M. Grey and James A. Moorer. Perceptual evaluations of synthesized musical instrument tones. *The Journal of the Acoustical Society of America*, 62(2):454–462, 1977.

[46] Harry H. Hall. Sound analysis. *The Journal of the Acoustical Society of America*, 8:257–262, 1937.

[47] Hank Heijink, Peter Desain, Henkjan Honing, and Luke Windsor. Make me a match: An evaluation of different approaches to score-performance matching. *Computer Music Journal*, 24(1):43–56, 2000.

[48] H. Helmholtz. *On The Sensations Of Tone As A Physiological Basis For The Theory Of Music.* Longmans, Green and Co., 3 edition, 1895. page 127.

[49] Barbara S. Kisilevsky, LiHui Pang, and Sylvia M.J. Hains. Maturation of human fetal responses to airborne sound in low- and high-risk fetuses. *The Journal of Early Human Development*, 58:179–195, 2000.

[50] A. Klapuri. Sound onset detection by applying psychoacoustic knowledge. In *Proc. IEEE Int. Conf. Acoust., Speech, and Signal Proc. (ICASSP)*, volume 6, pages 3089–3092, 1999.

[51] Anssi Klapuri. Automatic transcription of music. Master's thesis, Tampere University of Technology, Department of Information Technology, November 1997.

[52] Katrin Krumbholz, Rod D. Patterson, Andrea Nobbe, and Hugo Fastl. Microsecond temporal resolution in monaural hearing without spectral cues? *The Journal of the Acoustical Society of America*, 113(5):2790–2800, May 2003.

[53] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady*, 10(8):707–710, February 1966.

[54] Matija Marolt. Networks of adaptive oscillators for partial tracking and transcription of music recordings. *Journal of New Music Research*, 33(1):49–59, 2004.

[55] Teresa Marrin and Rosalind Picard. Proceedings of the international computer music conference. In *The "Conductor's Jacket": A Device for Recording Expressive Muscial Gestures*, October 1998.

[56] K. Martin. *Sound-Source Recognition: A theory and computational model.* PhD thesis, Massachusetts Institute of Technology, June 1999.

[57] K. D. Martin. Automatic transcription of simple polyphonic music: Robust front end processing. Technical Report 399, Massachusetts Institute of Technology, The Media Laboratory, December 1996.

[58] Keith D. Martin and Youngmoo E. Kim. Musical instrument identification: A pattern-recognition approach. In *Presented at the 136th meeting of the Acoustical Society of America*, October 1998.

[59] G.A. Miller and G. A. Heise. The trill threshold. *Journal of the Acoustical Society of America*, 22(5):637–638, September 1950.

[60] Dirk Moelants. Statistical analysis of written and performed music. a study of compositional principles and problems of coordination and expression in "punctual" serial music. *Journal of New Music Research*, 29(1):37–60, 2000.

[61] Brian C. J. Moore, Brian R. Glasberg, and Robert W. Peters. Relative dominance of individual partials in determining the pitch of complex tones. *The Journal of the Acoustical Society of America*, 77(5):1853–1860, May 1985.

[62] Brian C.J. Moore. *An Introduction to the psychology of Hearing.* Academic Press, 5th edition, 2003.

[63] James A. Moorer. On the segmentation and analysis of continuous musical sound by digital computer. Technical report, Center for Computer Research in music and Acoustics, Department of Music, Stanford University, May 1975.

[64] Joesph Derek Morrison and Jean-Marie Adrien. Mosaic: A framework for modal synthesis. *Computer Music Journal*, 17(1):45–56, 1993.

[65] Eric Métois. *Musical Sound Information. Musical Gestures and Embedding Synthesis.* PhD thesis, Massachusetss Institute of Technology, February 1997.

[66] Stefan Müller. Computer-aided musical performance with the distributed rubato environment. *Journal of New Music Research*, 31(3):233–237, 2003.

[67] Stefan Müller and Guerino Mazzola. The extraction of expressive shaping in performance. *Computer Music Journal*, 27(1):47–58, 2003.

[68] A.W. Nolle and C.P.Boner. The initial transients of organ pipes. *The Journal of the Acoustical Society of America*, 13:149–155, October 1941.

[69] Jeremy Pickens, Juan Pablo Bello, Giuliano Monti, Mark Sandler, Tim Crawford, Matthew Dovey, and Don Byrd. Polyphonic score retrieval using polyphonic audio queries: A harmonic modeling approach. *Journal of New Music Research*, 32(2):223–236, 2003.

[70] R. Plomp. The ear as a frequency analyzer. *The Journal of the Acoustical Society of America*, 36(9):1628–1636, September 1964.

[71] R. Plomp. *Aspects of tone sensation : a psychophysical study.* London New York : Academic Press, 1976.

[72] Giovanni De Poli, Antonio Rodà, and Alvsie Vidolin. Note-by-note analysis of the influence of expressive intentions and musical structure in violin performance. *Journal of New Music Research*, 27(3):293–321, 1998.

[73] Rudolf A. Rasch. *Generative Processes in Music. The Psychology of Performance, Improvisation and Composition.*, chapter 4. Timing and synchronization in ensemble performance. Oxford Science Publications, 1988.

[74] Edward M. Reingold. *Algorithms and Theory of computation handbook*, chapter Algorithm Design and Analysis Techniques, pages 16–18. CRC Press, 1999.

[75] Buddy Rich. *Buddy Rich's Modern Interpretation of Snare Drum Rudiments.* Embassy Music Corporation, 1942.

[76] E.G. Richardson. The transient tones of wind instruments. *The Journal of the Acoustical Society of America*, 26(4):960–962, November 1954.

[77] Don A. Ronken. Monaural detection of a phase difference between clicks. *The Journal of the Acoustical Society of America*, 1970.

[78] Evelyn Rothwell. *Oboe Technique.* Oxford University Press, 2nd edition, 1962.

[79] Stanley Sadie, editor. *The Violin Family*, chapter Two. Macmillan Press, London, 1989.

[80] E.L. Saldahna and John F. Corso. Timbre cues and the identification of musical instruments. *The Journal of the Acoustical Society of America*, 36:2021–2026, October 1964.

[81] Sylviane Sapir. Gestural control of digital audio environments. *Journal of New Music Research*, 31(2):119–129, 2002.

[82] D. Scheirer. Extracting expressive performance information from recorded music. Master's thesis, Massachusetts Institute of Technology, September 1995.

[83] M.R. Schroeder. Complementarity of sound buildup and decay. *The Journal of the Acoustical Society of America*, 40(3):549–551, 1966.

[84] Carl Seashore. *Psychology of Music.* McGraw-Hill Book Company, 1st edition, 1938.

[85] E. A. G. Shaw. Transformation of sound pressure level from the free field to the eardrum in the horizontal plane. *The Journal of the Acoustical Society of America*, 1974.

[86] Günter Sohler. Günter's midi compiler. http://mitglied.lycos.de/gsohler/linux.

[87] Robin Stowell. *Violin Technique and Performance Practice in the Late Eighteenth and Early Nineteenth Centuries.* Cambridge University Press, 1985.

[88] Johan Sundberg. Four years of research on music and motion. *Journal of New Music Research*, 29(3):183–185, 2000.

[89] Timidity. http://www.goice.co.jp/member/mo/timidity/.

[90] Dan Trueman and Perry cook. Bossa: The deconstructed violin reconstructed. *Journal of New Music Research*, 29(2):121–130, 2000.

[91] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, (64):100–118, 1985.

[92] Esko Ukkonen and Derick Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, November 1993.

[93] B. Vercoe. The synthetic performer in the context of live performance. In *Proceedings of the 1984 International Computer Music Conference.*, pages 275–278. San Francisco: International Computer Music Association, 1984.

[94] P. Walmsley, S. Godsill, and P Rayner. Polyphonic pitch tracking using joint bayesian estimation of multiple frame parameters. In *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, Oct 1999.

[95] Paul J. Walmsley, Simon J Godsill, and Pete J.W. Rayner. Polyphonic pitch tracking using joint bayesian estimation of multiple frame parameters. In *Proceedings of the IEEE Worksop in Applications of Signal Processing to Audio and Acoustics*, Oct 1999.

[96] F. Winckel. *Music, sound and sensation A modern exposition.* Dover publications inc. New York, 1967.

[97] R. Young. *McGraw-Hill Encyclopedia of Science and Technology*, volume 11, chapter Musical Acoustics, page 581. McGraw-Hill, 8 edition, 1997.

A Appendix - A

Quantitative Measurement of the Reliability of Automatic Pitch Detectors based on "Performance Distance"

Nicholas J Bailey[*] Jered Bolton[†] Damien Cirotteau[‡]

April 27, 2005

Abstract

A continuing problem in music technology is that of automatic transcription of music from an audio recording. Currently, this process is only even moderately reliable for monophonic music. The testing of such a process is problematic, as it requires either a huge investment of man-hours in comparing transcriptions with the original recordings, or otherwise can only produce anecdotal evidence of the transcriber's accuracy, being based on a small sample set. We propose a formal, semantically based automatic method of detecting transcription errors, wherein a large corpus of music may be presented to the automatic transcriber and the results analysed without human intervention. The new algorithm is used to produce an error analysis of a mature pitch-to-midi system.

1 Introduction

Pitch-to-MIDI conversion, or more generally, an automatic method for music transcription, is one of the holy grails of music technology. Various attempts have been adopted, with different goals and hence different requirements being placed on the converter. For example, extraction of melodies from a database rarely requires that the octave of a note be identified correctly, and in fact hardly depends upon the absolute pitch value being identified at all [GLCS95]. Automatic accompanists which listen to a soloist in order to generate synchronisiation information require somewhat more accurate pitch identification depending upon their algorithmic sophistication[DM88], but the octave is still relatively insignificant. Transcription programs designed to notate melodies, such as automatic amanuensis, certain musicological tools or plagiarism detectors require a much greater degree of accuracy.

In comparing the performance of such systems, it would be beneficial to be able to apply a standard metric which could be used to determine a somewhat standardised figure of merit (FoM). Although the problem will always be genre-specific to an extent, the very poor performance of existing programs, even with

[*]Centre for Music Technology, The University of Glasgow
[†]Centre for Music Technology, The University of Glasgow
[‡]CSC - DEI, University of Padua

relatively noise-free monophonic sources, makes the useful range of measurement fall within the scope of relatively easily constructed tools. We propose an algorithm which can be used to calculate a musically significant FoM repeatably and fairly based upon the presentation of a fixed and possibly very large corpus of works to the candidate transcription algorithms.

The algorithm has been tested using the Pitch-to-Midi converter from Grame [CFLO01], described in Section 2, as an example. The experimental technique adopted is described in Sections 3 & 4. Section 5 presents an algorithm developed to detect errors in recorded pitch.

2 The Grame Pitch-to-MIDI Converter

This Pitch-to-MIDI Converter (or pitchtracker) is based on an enhanced phase vocoder [DCM00], permitting the accurate determination of partial frequencies. A maximum-likelihood function is then applied in order to extract the best candidate fundamental. We will briefly present the algorithm, its intended application and some empirically observed strengths and weaknesses.

2.1 Algorithm

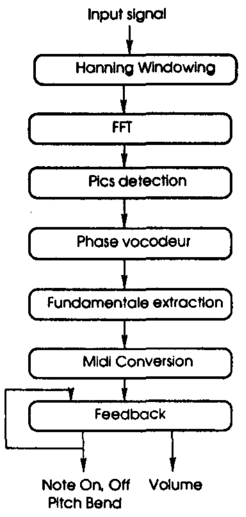


Figure 1: The pitchtracker architecture

The first step of the algorithm extracts the significant partials of the signal. Typically, after a noise gate which allows us to keep only significant signals, a Hanning window is applied and the spectrum is computed with a traditional FFT. The local maxima are then extracted and taken into consideration if greater than 3% of the global maximum of the spectrum. The precision in frequency of those partials is not as good as we would like. The traditional trade-off between time and frequency gives us a separation of 86.13 Hz between two consecutive bins if we work with a buffer of 512 points (11ms at a sampling rate of 44.1 kHz). This is a quite strong, and well-known, limitation of frequency domain analysis: if we want a good precision in frequency we have a low precision in time.

To reduce the severity of this limitation the signal derivative is used in order to obtain a higher precision for the partials found in the spectrum. This technique is due to Marchand [DCM00], and proceeds as follows:

Suppose a perfect audio signal (composed only from sinusoidal partials and without noise).

$$a(t) = \sum_{p=1}^{P} a_p(t) \cos(\varphi_p(t)) \tag{1}$$

with

$$\varphi_p(t) = \varphi_p(0) + 2\pi \int \omega_0 t f_p(u) du \tag{2}$$

From (1) and (2):

$$\frac{da(t)}{dt} = \sum_{p=1}^{P} 2\pi f_p(t) a_p(t) \cos\left(\varphi_p(t) - \frac{\pi}{2}\right) \tag{3}$$

The signal is discrete so let DFT^k be the spectrum of the *DFT* of the k-th derivative of the signal.

$$\text{DFT}^k[m] = \frac{1}{N} \left| \sum_{n=0}^{N-1} w[n] \frac{d^k a}{dt^k}[n] e^{-j\frac{2\pi}{N}nm} \right| \tag{4}$$

Once the precise partials have been determined, the best candidate fundamental must be selected. A *maximum likelihood* function such as in [IB98], is used. For each partial, the value (5) is computed with f being the accurate frequency of the partial, p being the other components of the signal and P the number of partial of the signal.

$$\Gamma(f) = \sum_{p=1}^{P} O_p(f) Y_p(f) \tag{5}$$

Y_p is a triangular function centered on the computed peak. This factor is the tolerance within a peak that is considered as a partial of a certain note. The nearer p is close to a multiple of f, the highest the value of Y_p, and if p is too far from nf then Y is null.

$$\begin{aligned} Y_p(f) &= \frac{nf - h(p)}{h(p) - f_{\text{min}}} + 1 & f_{\text{min}} \leq nf \leq h(p) \\ &= \frac{f_{\text{max}} - nf}{f_{\text{max}} - h(p)} & h(p) \leq nf \leq f_{\text{max}} \\ &= 0 & \text{otherwise} \end{aligned} \tag{6}$$

with $n \in [2, P]$.

As lower partials have a greater importance than higher ones in the formation of a note, lower partial are awarded a higher higher score, O_p:

$$O_p(f) = \frac{0.9}{i - 0.1} \tag{7}$$

The partial which maximises $\Gamma(f)$ is considered as the fundamental. The frequency is converted to a MIDI note and a pitchbend value which corresponds to the deviation between the theoretical value of the MIDI note and the detected frequency.

A tunable stability threshold is then applied: a fundamental has to be detected x consecutive times in order to be considered as a consistent note. The accuracy of the system is then $x * w$, with w being the size of the window. We note that we need a small value for x to obtain a good accuracy, but small a x produces more errors during the attack. Typically $x = 3$ gives a good compromise between accuracy and reliability, but it has to be tuned depending on the instrument and the size of the window. Careful tuning helps the system to only detect stable notes, resulting in MIDI note data output which corresponds with the wave file input.

3 Detecting Errors in Pitch

In order that a large database be available to perform the tests, monophonic output is generated by a synthesiser from an existing MIDI file. The generating audio stream can be fed directly to the audio input of the pitch-to-midi converter under test, for example using a named pipe in a Posix-compliant operating system; the output from the converter may then be compared with the original MIDI file using the proposed pitch-error-detection algorithm. Error detection algorithms are presented with two sequences of tokens representing notes. The first is the reference sequence ("the score"), and the second is the output from the candidate pitch-to-MIDI converter ("the performance"). Two fundamental requirements of an error-detection algorithm are:

1. that it reliably detects error categories identified in Table 1; and
2. that errors are reported in a manner which maximises the number of matches between the score and its performance.

The second requirement effectively gives the benefit of doubt to the performer. That is, it is expected that they will attempt to correctly perform as many notes as possible.

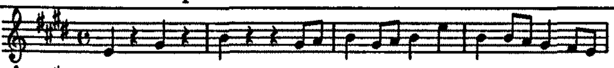
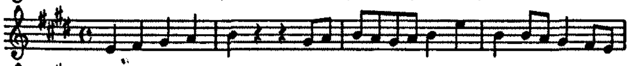
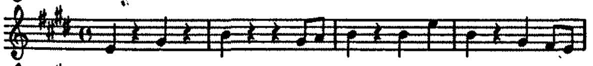
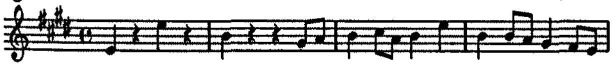
Fault	Musical Example
Original	
Extra Notes ⊕	
Missing Notes ⊖	
Wrong Notes ⊗	

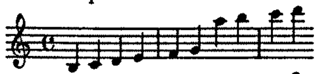
Table 1: Types of errors detected by the proposed algorithm

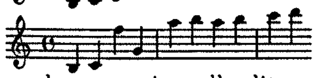
Identification of errors relies on the assumption that a sequence of one or more mismatches are followed by "correct" notes. When a mismatch occurs the realignment of the score and performance allows the error type(s) to be inferred.

The errors presented in Table 1 correspond to the editing operations used by Ukkonen[Ukk85] in his work on string matching. These editing operations, based on those formulated by Levenshtein[Lev66], are used to determine the edit distance (or minimum cost) of converting string A into string B. Comparison of pitch information is closely related to string matching, but not identical to it.

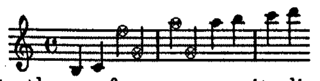
4 From Edit-Distance to "Performance Distance"

The purpose of string matching is to find the most efficient sequence of editing operations so that string B can be modified to match string A. However, in a musical context the purpose of string matching is to determine how string B (the performance) deviates from string A (the score). This provides a means of measuring the accuracy of the performance (which in the context of this paper relates to the performance of the pitchtracker). The phrase "Performance Distance" (PD) is used to decribe editing operations that relate to score deviations rather than the shortest edit distance. The outcome of this approach is that the PD is not necessarily the same as the (shortest) edit distance, which in some cases can misrepresent the performance.

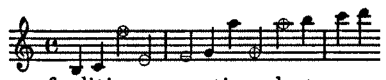
Score 

Performance 

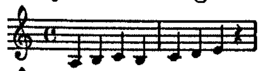
In the above example, assuming all edit operations have an equal cost, the edit distance is:

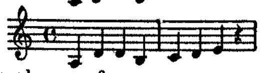
Edit Distance 

This misrepresents the performance as it eliminates the correctly performed G4. The PD of the above is:

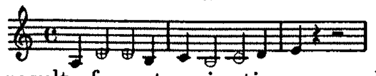
Performance Distance 

This is more costly in terms of editing operations but preserves the correctly performed G4. Bora *et al.* present a tool specifically designed for comparison of performances on a midi keyboard[BTB00]. Their system judges a realignment position to be correct as long it and the next pair of notes in the score and performance match. This, coupled with an unexplained *"if...then...else..."* ordering of error type checking (which imposes an order of precedence on error types), results in their system finding false matches. For example:

Score 

Performance 

It is obvious that the performance contains two wrong notes in beats two and three of the first bar. Due to the ordering of error checking and the requirement that only two matches indicate successful identification of an error, the system presented by Bora *et al.* would find:

Bora *et al.* 

This outcome results from terminating error checking as soon as a realignment position that conforms to their matching criteria is found; the possibility of two wrong notes is not considered. Due to a restriction imposed by a threshold number of (consecutive) errors allowed, their system does not guarantee the successful comparison of a score and its performance.

As well as reviewing other comparison systems, Heijink *et al.*[HDHW00] present a comparator which uses a similar approach to Ukkonen: an error and the identity of its type are evaluated in isolation. Each error provides a number of alternative "paths" which represent different combinations of matches and errors. The path with the least number of errors (i.e. smallest edit distance) is deemed to be correct, meaning that the edit, rather than performance, distance is found.

Vercoe[Ver84] and Dannenberg[Dan91] are generally considered to be the pioneers of real-time computerised score performance matching. The system described by Dannenberg[Dan88] is based on an offline matching system which uses similar techniques to that of Ukkonen. Instead of calculating an array of edit costs, Dannenberg populates an array with values that correspond to the Longest Common Substring (LCS) for any given position. Despite the use of LCSs, Dannenberg's system does not find the PD, as the "correct" path through the array is chosen in terms of efficiency resulting in the edit, rather than performance, distance.

The detection of the PD is dependent on finding the combination of tokens which best represent the performance. The sequential nature of musical notes presented to a comparator dictates that the definition of error token combinations adhere to a set of semantic rules.

4.1 Incorporation of Error-Reporting Semantics

An algorithm based on the mismatch of tokens across a limited window-length can be confounded by choosing a test case where it is unclear within the scope of the lookahead which candidate error is "correct". Furthermore, it is easy to contrive an example which causes algorithms which find the edit distance to misrepresent the performance by combining the effect of several different errors within the scope tested. In order to overcome these limitations, it is necessary to take combinations of errors into account by assessing each *combination* of errors before declaring the winning candidate.

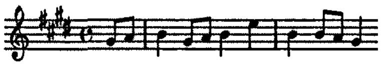
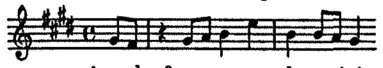
4.1.1 Error Combination Semantics

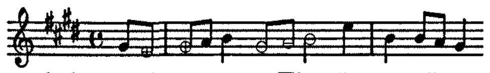
While employing the notation presented in Table 1, $\ominus\oplus$ and $\oplus\ominus$ are semantically different, both error combinations are the equivalent of $\otimes$. The existence of $\otimes$ means that it is impossible for both $\oplus$ and $\ominus$ to appear consecutively in the same (multiple) error definition. It is also impossible for $\otimes$ to appear immediately after $\oplus$ or $\ominus$ in an error combination. In terms of semantics the only valid error definitions and combinations are:

- zero or more $\otimes$ immediately followed by zero or more $\oplus$
- zero or more $\otimes$ immediately followed by zero or more $\ominus$

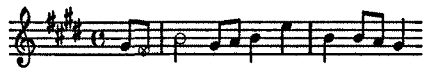
Errors are separated by correct (matching) characters.

4.1.2 An Example of Error Combination

Suppose that when presented with  , the performer played  clearly making an error in the upbeat and then pausing before resynchronising in the second beat of the first full bar. Algorithms which fail to evaluate error combinations misattribute this error as:

No error combinations | 

This a clear misrepresentation of the performance. The "correct" account of the error is:

With error combinations | 

4.2 Assessment of Token Combinations

Algorithm 1 shows how an iterative scoring approach is used to find a combination of tokens which adheres to the prescribed error combination semantics and most appropriately describes the performance with respect to the score. The notation $A \sqsubseteq B$ is used to indicate that list A is a subsequence of B, which is to say that B contains the whole of A in order with zero or more additional tokens interspersed.

Lists of token combinations are rated according to the number of matches they contain. The list which produces the highest number of matches is judged to be the winner. The consequence of this approach is that the entire score and performance must be evaluated. Failure to do so removes the certainty of finding the "correct" error candidate.

5 Implementation in Haskell

We present a reference implementation in Haskell[HF92, HJK+92], an object-oriented, polymorphically typed, lazy, purely functional language. If the type *Pitch* has been defined which represents score and performance tokens appropriately, the function *report* returns a comparison between score and performance, both being a list of *Pitch*.

```
dataPerfToken = CorrectPitch
              | MissingPitch
              | InsertedPitch
              | WrongPitchPitch
                deriving(Eq, Show)
report :: [Pitch] → [Pitch] → [PerfToken]
report score perf =
   errSeqToPerfTokens score perf (fst (getError score perf))
```

The return value is a list of type *PerfToken*, which can represent both pitch values and error types.

```
errSeqToPerfTokens :: [Pitch] → [Pitch] → [Error] → [PerfToken]
errSeqToPerfTokens (s : rscore) (p : rperf) (e : rerr)
```

```
case e of
  (Omitted, n)      →  — skip n notes in the score
        map (λ m → Missing m) (take n (s : rscore)) ++
        errSeqToPerfTokens (drop (n − 1) rscore) (p : rperf) rerr
  (Extra, n)        →  — skip n notes in the performance
        map (λ i → Inserted i) (take n (p : rperf)) ++
        errSeqToPerfTokens (s : rscore) (drop (n − 1) rperf) rerr
  (Incorrect, n)    →  — indicate notes in error
        map(λ(a, b) → Wrong a b)
            (zip (take n (s : rscore)) (take n (p : rperf))) ++
            errSeqToPerfTokens (drop (n − 1) rscore) (drop (n − 1) rperf) rerr
  (NotAnError, n) →  — score and performance agree
        map (λ c → Correct c) (take n (s : rscore)) ++
        errSeqToPerfTokens (drop (n − 1) rscore) (drop (n − 1) rperf) rerr
```

If both arguments are non-null, and their first tokens match, the result is a *Correct* token followed by the result of applying the algorithm to the rest of the score and performance.

If the first tokens fail to match, *errSeqToPerfTokens* is invoked to determine the type(s) and extent of the error. The appropriate information is prepended to the result, the required tokens dropped from the score and performance token list, and the *getError* function is invoked recursively to continue processing.

Three termination conditions must be supplied because there is no guarantee that the score and performance will be of the same length

```
errSeqToPerfTokens (s : rscore) [] e = Missing s  :  errSeqToPerfTokens rscore [] e
errSeqToPerfTokens [] (p : rperf) e  = Inserted p  :  errSeqToPerfTokens [] rperf
errSeqToPerfTokens [] [] _           = []
```

errSeqToPerfTokens applies corruption tests to the remainder of the score and performance by invoking *getError*.

```
getError ::[Pitch] → [Pitch] → ([Error], Int)
— An empty score matches an empty performance
— but with a match score of 0
getError [] [] =([], 0)
— If the score/performance runs out, all remaining notes are
— extra/omitted and the match scores 0 points
getError [] perf  =  ([(Extra, lp)], 0)
    where
      lp  =  length perf
getError score []  =  ([(Omitted, ls)], 0)
    where
      ls  =  length score
— Otherwise, recurse to find best error description
getError score perf =selectMaxPoints (tryAllCases score perf)
```

We introduce the following types to represent the type of error and its extent:

```
data ErrorType = Omitted | Extra | Incorrect | NotAnError
                    deriving (Eq, Show)
type Error       = (ErrorType, Int)
```

A tuple containing a list of errors and a rating *([Error], Int)* is returned. *selectMaxPoints* is defined thus:

```
selectMaxPoints               :: [([Error], Int)] → ([Error], Int)
selectMaxPoints (st : points) = foldl maxPoints st points
  where
    maxPoints :: ([Error], Int) → ([Error], Int) → ([Error], Int)
    maxPoints e1@((firste1,_) : _, e1count) e2@((firste2,_) : _, e2count)
       | e1count > e2count = e1
      — Semantics demand that preference be given to error
      — sequences beginning (Wrong, _).
      — Humans prefer to say "Wrong x should be y" rather
      — than "y was omitted then x inserted", even though
      — the transformations are equivalent.
       | e1count == e2count =
          if firste1 == Incorrect then e1
          else if firste2 == Incorrect then e2
          else e2
       | otherwise          = e2
```

The candidate lists of token combinations are formed by recursing through the top-level *getError* function testing *all possible* combinations of each candidate token (*Omitted*, *Extra*, *Wrong* and *NotAnError*). This results in the code presented in Figure 2.

tryAllCases iterates from the end to the beginning of the score and performance, returning the likelihood rating for each combination (list) of tokens. Since this could result in a string of errors of the same type, for example `[(Omitted,1),(Omitted,1),...], match_score)`, the infix operator <> is defined which folds all such consecutive instances of the same error into the first tuple of the list.

Finally, a *summary* function is also provided which writes out the results from *report* in a more compact form.

5.1 Results

The test examples from Table 1 were presented to the comparator with the following results.

```
Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
MusicTypes.hs
Report.hs
Report> summary (report [E,Gsharp,B,Gsharp,A,B,Gsharp,
A,B,E,B,B,A,Gsharp,Fsharp,E] [E,Fsharp,Gsharp,A,B,Gsharp,
A,B,A,Gsharp,A,B,E,B,B,A,Gsharp,Fsharp,E])
"E+Gsharp+BGsharpAB+GsharpABEBBAGsharpFsharpE"
Report> summary (report [E,Gsharp,B,Gsharp,A,B,Gsharp,
A,B,E,B,B,A,Gsharp,Fsharp,E] [E,Gsharp,B,Gsharp,A,B,B,
E,B,Gsharp,Fsharp,E])
"EGsharpBGsharpAB--BEB--GsharpFsharpE"
Report> summary (report [E,Gsharp,B,Gsharp,A,B,Gsharp,
A,B,E,B,B,A,Gsharp,Fsharp,E] [E,E,B,Gsharp,A,B,Csharp,
— Make the exhaustive test for each offset and each error.

tryAllCases          :: Int → [Pitch] → [Pitch] → [([Error], Int)]
tryAllCases _[]      = []
tryAllCases[] _      = []
tryAllCases score perf =
  let
    omittedErr = tryOmitted score perf
    extraErr   = tryExtra score perf
    wrongErr   = tryWrong score perf
    notErr     = tryNotAnError score perf
  in
  in
  [notErr, wrongErr, omittedErr, extraErr]
— Assuming a token omitted/extra/wrong/correct, the following
— counts the maximum possible number of matching tokens
— by comparing score and performance lists.

— To test for an omitted note in the performance, skip notes in
— the score and count how many match.
tryOmitted                   :: [Pitch] → [Pitch] → ([Error], Int)
tryOmitted (s : rscore) perf =((Omitted, 1) <> getError rscore perf)
— Testing for extra notes in performance is the reverse of the above
tryExtra                     :: [Pitch] → [Pitch] → ([Error], Int)
tryExtra score (p : rperf) =((Extra, 1) <> getError score rperf)
— Check the conseqence of there being a wrong tokens by ignoring the
— first token of the score and performance
tryWrong ::[Pitch] → [Pitch] → ([Error], Int)
— Suggesting a trailing score or performance is a
— "wrong note" is penalised
tryWrong [] _                  = ([(Incorrect, 1)], −1)
tryWrong _ []                  = ([(Incorrect, 1)], −1)
tryWrong (s : rscore) (p : rperf) = ((Incorrect, 1) <> getError rscore rperf)
— Test for a correct tokens. Suggestions that non-matching tokens
— are correct are penalised.
tryNotAnError                        ::[Pitch] → [Pitch] → ([Error], Int)
tryNotAnError (s : rscore) (p : rperf)
  | s == p     = ((NotAnError, 1) <> (errors, count + 1)
  | otherwise = ([(NotAnError, 1)], −1)
where
  (errors, count) = getError rscore rperf
— Permit an error to collapse into the list if it is prepended to
— an error of the same sort
infix 5 <>
(<>) :: Error → ([Error], Int) → ([Error], Int)
newErr <> ([], matches) = ([newErr], matches)
newErr@(neType, neLen) <> (oldErrs@((fstEType, fstELen) : rErrs), matches)
  | neType == fstEType = ((neType, neLen + fstELen) : rErrs, matches)
  | otherwise          = (newErr : oldErrs, matches)
```

Figure 2: The Main Body of the Error Description Algorithm

```
A,B,E,B,B,A,Gsharp,Fsharp,E])
"E/BGsharpAB/ABEBBAGsharpFsharpE"
```

In each case, extra notes ('+'), missing notes ('-') and wrong notes ('/') have been correctly identified, and musical semantics are correctly taken into account.

5.2 Efficiency Considerations

The presented error detection algorithm is unconstrained by local window-size considerations, but is extremely poorly conditioned with respect to complexity. Each candidate error that is tested potentially results in a recursive call to the *getError* function which typically results in identical expressions being evaluated many times. Consequently, the complexity of performing error matches on strings of notes length n is $O(n^3)$ which is entirely unacceptable even for very small score and performance fragments such as those included in this paper. The example quoted above required just over a minute and a half of processor time on a Motorola G4-series PowerPC!

```
Report> summary (report [Gsharp,A,B,Gsharp,A,B,E,B,B,A,Gsharp]
                        [Gsharp,Fsharp,Gsharp,A,B,E,B,B,A,Gsharp])
"Gsharp/-GsharpABEBBAGsharp"
Report> :q
[Leaving Hugs]

real    1m35.662s
user    1m35.350s
sys     0m1.180s
```

A much improved version of the code has been implemented in C which unwraps one level of recursion into an iteration, and caches results from the recursive examination using a hash table. The resulting joining of the search tree is similar in efficacy to method used in [HDHW00]. Compared with the Haskell prototype, the resulting C code is less transparent and less problem-oriented. It is not presented here, but along with with the complete code of the Haskell program is available for download at http://cmt.gla.ac.uk/Software/software.html.

6 Error Analysis Results

6.1 Intended Applications and Empirical Observations

GRAME's pitchtracker has been developed mainly for interactive music. Efficient computing strategies were used in order to provide a low computing cost allowing real time analyses. Furthermore the high capacity of integration of this converter allows off-line analysis, and automatic part transcription (although this is not yet implemented). Different applications have been realised (under Linux, MacOS and Windows) with this library such as a MidiShare driver, some modules for block-diagram based environments (Max/MSP and EyesWeb), and stand alone applications.

The GRAME pitchtracker has been tested in concert mainly with flute and voices. Other live tests (in studio) have been done with 'cello, guitar, trumpet and didgeridoo. All of them but didgeridoo (obviously) have produced good results with accurate tuning, especially voices and flute, probably because more experiments have been conducted with those instruments. In the middle register of the flute, some errors were observed, typically the lower octave being recognised during the attack of the note. Since the conversion was used in an interactive piece to trigger MIDI-controlled events, the mis-identification does therefore pose a problem for the composer and performer (wrong events started, for example). Analysing the spectrum of the medium register of the flute, it was noticed that the attack was always one octave lower. This limitation was overcome in practice by increasing the stability-time threshold.

'Cello, guitar and trumpet produced good results too even if more errors were detected. The low registers of the 'cello and the guitar were not very well recognised: mainly mistaken notes and octave errors. Rapid phrases in trumpet were hardly recognised due to noisy attack and too weak accuracy of the system.

One of the main weakness of this pitchtracker is imposed upon it by its real-time nature. Off-line processing can detect stable notes and then adjust the timings: once it finds a stable note, it can decide that the preceding attack was part of that note and then adjust the timing in order to match as best as possible the played sequence. With a real-time process, such a procedure is not possible. Every slice has to be characterised on the fly. In consequence, attacks are hardly detected because of their unvoiced structure. Trying to detect the attack exposes us to mistaken notes. Therefore, we have to deal with a trade-off between accurate timing and strength of the recognition.

6.2 Pitch Tracker Tests

Ten test scores were created such that each score provided a typical example of a difficult note detection scenario. Table 2 shows the results for each test.

Each test was performed with the PitchTracker set to use a buffer of 512 points, except test three which used 1024 points. The test results reflect the realtime nature of the pitch tracker. For example, tests 3 and 6 show that the pitch tracker consistently failed to detect consecutive repeated notes. It also failed to detect the very high register notes of tests 1, 2, 4, 5, 7 and 9. This again is consistent with the form of the pitch tracker in that note harmonics were too short to satisfy the pitch tracker's criteria for what constitutes an established note.

The comparator presented here guarantees a correct comparison of a score and performance without the constraints of a finite and arbitrary window length. Semantics of musical performance are also taken into account. With a reliable and automatic method of assessment of accuracy for pitch trackers, the comparator could be used in conjuction with a genetic algorithm to repeatedly test the pitch tracker using different parameters allowing its settings to be optimised for a given instrument.

References

[BTB00] Uzay Bora, Selmin Tufan, and Semih Bilgen. A tool for comparison of piano performances. *Journal of New Music Research*, 29(1):85–99, 2000.

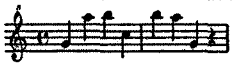
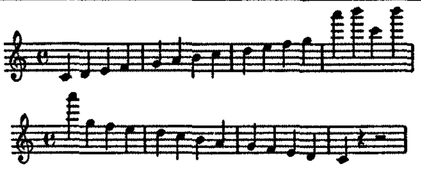
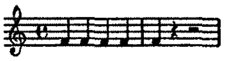
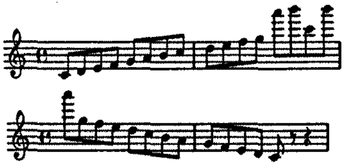
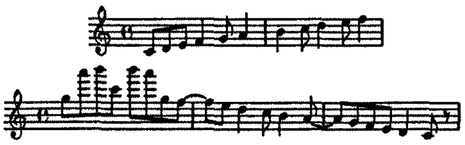
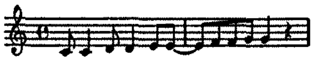
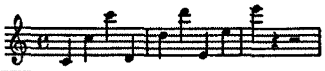
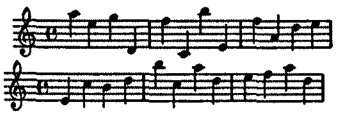
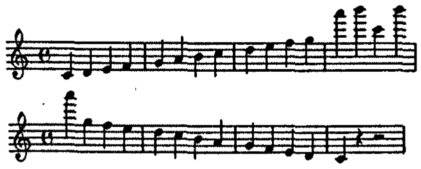
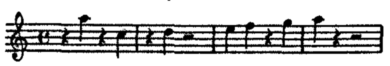
Test No.	Score	Correct	⊗	⊖	⊕
1		4	1	2	0
2		25	0	4	0
3		1	0	4	0
4		24	2	3	0
5		24	0	5	0
6		4	0	6	0
7		7	0	2	0
8		20	2	2	3
9		22	1	6	1
10		5	1	1	0

Table 2: Test Results for the GRAME Pitch Tracker

[CFLO01] D. Cirotteau, D. Fober, S. Letz, and Y. Orlarey. Un pitchtracker monophonique. In IMEB, editor, *Actes des Journes d'Informatique Musicale JIM2001, Bourges*, pages 217–223, 2001.

[Dan88] Roger B. Dannenberg. Music understanding by computer. *Carnegie Mellon School of Computer Science*, pages 19–28, 1987/1988.

[Dan91] Roger B. Dannenberg. *Music, Language, Speech, and Brain*, chapter Recent Work in Real-Time music Understanding by Computer. Macmillan, 1991.

[DCM00] M. Desainte-Catherine and S. Marchand. High precision Fourier analysis Using Signal Derivatives. *Journal of the Audio Engineering Society*, 48(7/8):654–667, July/August 2000.

[DM88] Dannenberg and Mukaino. New techniques for enhanced quality of computer accompaniment. In *Pro- ceedings of the International Computer Music Conference*, pages 243–249. Computer Music Association, September 1988.

[GLCS95] Asif Ghias, Jonathan Logan, David Chamberlin, and Brian C. Smith. Query by humming -- musical information retrieval in an audio database. In *ACM Multimedia 95 — Electron Proceedings*, 1995.

[HDHW00] Hank Heijink, Peter Desain, Henkjan Honing, and Luke Windsor. Make me a match: An evaluation of different approaches to score-performance matching. *Computer Music Journal*, 24(1):43–56, 2000.

[HF92] Paul Hudak and Joseph H Fasal. A gentle introduction to haskell. *ACM SIGPLAN Notices*, 27(5):1–52, May 1992.

[HJK+92] Paul Hudak, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, John Peterson, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Mara M. Guzmn, Kevin Hammond, and John Hughes. Report on the programming language haskell, v1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[IB98] O. Izmirli and S. Bilgen. Multiple Fondamental Tracking for Polyphonc Note Recognition. In *Recherches et applications en informatique musicale*, pages 305–314. Paris, hermes edition, 1998.

[Lev66] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady*, 10(8):707–710, February 1966.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, (64):100–118, 1985.

[Ver84] B. Vercoe. The synthetic performer in the context of live performance. In *Proceedings of the 1984 International Computer Music Conference.*, pages 275–278. San Francisco: International Computer Music Association, 1984.

Algorithm 1 Determine the combination of tokens which leads to the maximum number of matches between the score and performance

```
Require: Score $S_{1...m}$, Performance $P_{1...n}$
Ensure: Analysis $R = \{x|x_p = S_q \vee x_p = \oplus \vee x_p = \ominus \vee x_p = \otimes\}*, R \sqsupseteq S$
  $s \leftarrow 1$
  $p \leftarrow 1$
  while $S_{s...} \neq \epsilon \wedge P_{p...} \neq \epsilon$ do
    if $S_s = P_p$ then
      Append $S_s$ to $R$
      Increment $s$ and $p$ by 1
    else Mismatch detected
      for $n \in 1 \ldots$(max(m-q, n-p)) do
        for $e \in$ (Missing, Extra, Wrong) do
          Evaluate rating $r$ assuming error $e$ lasting $n$ input tokens
          Error $E \leftarrow (e, n)|_{r\ \mathbf{max}}$
        end for
      end for
      Now perform corrections in the light of highest-scoring error combination
      if $e =$ Missing then tokens missing from Performance
        Append $n$ $\ominus$ to $R$
        Increment $s$ by $n$
      else if $e =$ Extra then extra tokens in Performance
        Append $n$ $\oplus$ to $R$
        Increment $p$ by $n$
      else if $e =$ Wrong then Performance and Score do not agree
        Append $n$ $\otimes$ to $R$
        Increment $p$ and $s$ by $n$
      else if $e =$ Wrong,Missing then Performance and Score do not agree
      and tokens are missing from Performance
        Append $n$ $\otimes$ and $m$ $\ominus$ to $R$
        Increment $p$ by $m$ and $s$ by $n + m$
      else if $e =$ Wrong,Extra then Performance and Score do not agree and
      extra tokens are in Performance
        Append $n$ $\otimes$ and $m$ $\oplus$ to $R$
        Increment $p$ by $m + n$ and $s$ by $n$
      end if
    end if
  end while
```

B Appendix - B

Towards Real Time Score Performance Matching Using Musical Performance Distance.

Jered Bolton and Nicholas J. Bailey

29th April 2005

Abstract

Real time performance tracking systems are usually based on offline score performance comparators. Such a derivation leads to a compromise of performance which can result in the derived system failing to properly track a performance. Tracking systems can be improved by factoring in performance error models. However, su ch errors are not necessarily made by the performer but are as a result of the actual approach used to compare a score and performance. We re-examine the assumption that a straight forward string matching algorithm can adequately represent a performance (and in doing so examine the semantics of tokens used for the representation of a performance) and present a new algorithm which gives a musically fair representation of a performance and works without modification in both real time and offline environments.

Introduction

Comparison systems, both offline and online (realtime), which compare a score with a performance use techniques from the field of string matching. The three editing operations of wrong (denoted $\otimes$), missing ($\ominus$) and extra ($\oplus$) characters correspond directly with possible performance errors.

Existing comparators Bora *et al.* (2000); Dannenberg (1991); Heijink *et al.* (2000); Pardo and Birmingham (2002) make the assumption that algorithms designed for string matching are appropriate for use in a musical context.

In a bid to improve performance Pardo and Birmingham (2002) extend their performance follower by modeling typcial transcription errors and exploiting timing information. Their work assumes that the underlying comparison algorithms are providing a correct representation of the performance.

Performance errors are detected by exploiting the assumption that following an error(s), correct notes will be performed. A system determines the best way of realigning a performance with its score by finding "correct" notes which match the score. It is well known that although a recursive approach appears to be intuitively correct, it is grossly inefficient. Instead the dynamic programming method of using a two dimensional array or matrix, populated using certain rules, has emerged as an efficient way of tackling the problem (Levenshtein (1966); Ukkonen (1985)).

There are two schools of thought as to how the matrix should be populated: edit cost and Longest Common Substring (LCS). In the case of the former, the value of each location is calculated according to the "cost" of converting (if necessary) the performance into the score, whereas for the latter, each location is filled with an integer representing the LCS for the available substrings.

The use of LCS provides a certain amount of context for editing operations. Context allows the identification of a given error to have repurcussions for other errors. This advantage is lost when a matrix is populated purely in terms of the cost of character substitution. The use of LCS by Dannenberg (1987/1988) positions his comparator as being the best in terms of musical representation. It is for this reason that his comparator will be used as a starting point for investigating whether string matching techniques are appropriate in a musical context.

Longest Common Substrings

For a matrix M of size s, p (where s and p are the lengths of the score and performance respectively), Dannenberg (1987/1988) adheres to the normal procedure of finding an optimum path (which corresponds to the shortest, or most efficient, edit distance) through the matrix by starting at $M_{s,p}$ and tracking backwards to $M_{0,0}$. He populates his matrix row by row, starting at $M_{0,0}$ filling each location with an integer representing the LCS for the available substrings. This results in a system that finds the longest common substring with respect to the the start of the score but does not easily lend itself to real time processing.

The consequence of this are twofold. Firstly, errors are identified in a manner which does not account for any impact a given decision may have on the system's ability to correctly identify future errors. Whilst this does not prevent Dannenberg's method from finding an edit distance, it does allow for the performance to be misrepresented. Secondly, because the system commences evaluation at $M_{s,p}$ or from the end of a look-ahead region if windowing is being used, the path finding routine must first process unperformed notes in the score before reaching actual performance data. This also automatically results in an outcome which is not musically relevant. Consider the following snapshot of a performance:

```
      Score = abdefg
Performance = abc
```

The performer has misenterpreted the score as a simple scale, clearly misplaying the third note. The matrix for this stage is shown in Table 1. Starting at $M_{s,p}$ forces the comparator to produce the following output:

	Score						
		a	b	d	e	f	g
Performance	a	1	1	1	1	1	1
	b	1	2	2	2	2	2
	c	1	2	2	2	2	2

Table 1: A LCS populated array for comparison of a score and performance.

```
        Output = ab⊖⊖⊖⊗
```

Thus the system has misrepresented the performance.

Dannenberg acknowledges this problem by reporting "strange behaviour", which also arises from a "wrong" note matching a note much later on in the score. This problem is discussed in the Future Common Longest Substrings section on page 186.

Error Semantics

Error semantics are discussed in depth in Bailey *et al.* (2005) and will only be summarised here. The possible errors a performer can make (when only taking sequential order into account) can be defined as: 1) zero or more ⊗ immediately followed by zero or more ⊕; 2) zero or more ⊗ immediately followed by zero or more ⊖.

LCS and the performance distance

A correct representation of the performance must maximise the number of (musically) "correct" notes that *follow* an error.

The prima facie approach would be to process the score and performance backwards (by starting at $M_{s,p}$) as the "correct" notes that follow an error will (by default) be processed first.

However, this is not the case. The path finding algorithm makes use of information provided by a grid population routine. Such routines work forwards through the score and performance, populating the grid with the results of the comparisons it makes. Consequently LCSs primarily consist of notes that precede an error. By definition all notes leading up to the first error are correct. In terms of error identification the information provided by:

```
          Score = abcdefghijklmno
    Performance = abcdefghijklXno
```

is the same as:

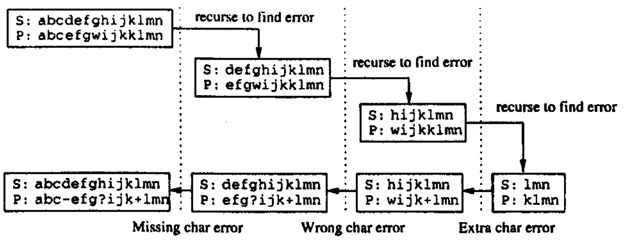


Figure 1: A simplified recursive comparator identifying the three error types.

```
      Score = mno
Performance = Xno
```

Hence, errors are processed in the order in which they are made, preventing the system from taking the knock-on effect identifying an error can have on future (virtual) errors into account.

Virtual error(s) occur when a comparison system finds a non-ideal alignment position. Such positions match only a few notes before another "virtual" error is found, i.e. one that has arisen as a result of the non-ideal realignment position rather than performer error.

Whilst systems which use LCSs can guarantee to find an edit distance (usually the most efficient) there is no guarantee that it is musically relevent, as such systems do no seek to maximise the number of correct notes that follow an error(s). Thus the performance distance (PD) is the edit distance which maximises the number of musically relevent matches following an error, irrespetive of efficiency.

Although grossly inefficient, a recursive implementation, as shown in Figure 1, helps to explain why errors must be processed in reverse. The occurrence of an error causes recursion to take place. Thus the last error is identified first meaning this result is taken into account when identifying the second and first errors. The same outcome can be achieved using a matrix if the direction of the matrix population and path finding routines is reversed.

Future Common Longest Substrings

The matrix is still populated row by row, but the routine commences at at $M_{s,p}$; a resulting matrix is shown in Table 2. Consequently the path finding routine starts at $M_{0,0}$ resulting in a semantically correct outcome and better efficiency as the path finding routine can terminate as soon as the end of the performance is reached.

The path corresponding to the PD is found by starting at $M_{0,0}$ and obeying the rules shown in Algorithm 1 (for S = the score, P = the performance and $s = 0$, $p = 0$).

		Score					
		a	b	d	e	f	g
Performance	a	2	1	0	0	0	0
	b	1	1	0	0	0	0
	c	0	0	0	0	0	0

Table 2: Showing a FLCS populated array for comparison of a score and performance.

```
if $S_o = P_c$ then $s$++, $p$++, else
    1. if $M_{s,p+1} = M_{s+1,p+1} = M_{s+1,p}$ then increment $s$ & $p$, else
    2. if $M_{s,p+1} > M_{s+1,p+1}$ and $M_{s,p+1} > M_{s+1,p}$ then increment $p$, else
    3. if $M_{s+1,p} > M_{s+1,p+1}$ and $M_{s+1,p} > M_{s,p+1}$ then increment $s$, else
    4. if $M_{s,p+1} = M_{s+1,p}$ then increment $s$ or $p$
```

Algorithm 1: Path Finding Rules

Items 1-3 of Algorithm 1 represent the identification of wrong, extra and missing error types respectively. Item 4 is a special case for catching ambiguous error situations where the choice between a missing or extra error type is arbitrary for an isolated error and dependent on the previous error type for multiple errors. Thus errors are processed in reverse allowing the system to find a path through the grid which corresponds to the performance distance. Whilst this approach solves Dannenberg's "strange behaviour" problem of a "wrong" note causing the system to immediately jump to the end of the score, the same symptoms caused by a "wrong" note appearing later on in a score are still present.

The use of FLCS guarantees the detection of a PD which is semantically correct. However, it can result in an outcome that is musically unlikely. Factors which contribute to a such an outcome include the instrument used for the performance, tempo, complexity of score and the number and combination of repeated notes within the score:

```
      Score = abdefgfcdefg
Performance = acdefg
```

The PD of the above is:

```
PD = a⊖⊖⊖⊖⊖⊖cdefg
```

The comparator has ensured that the maximum number of matches has been found between the performance and the score. From a musical perspective, why would a performer jump six notes in the score? It is more likely that the performer having noticed their mistake, stops playing. The Musical Performance Distance (MPD) is therefore:

```
MPD = a⊗defg⊖ ⊖ ⊖ ⊖ ⊖⊖
```

The comparator in giving the benefit of doubt to the performer assumes that any note played by the performer will be from the score. The consequence of this is that a wrong note can only be immediately identified if it is not part of the remaining score. In the example above, if the performer had continued playing past the second "f" then the comparator would have realigned itself, resulting in the first "c" being recognised as a wrong note.

Dannenberg addresses this problem by modifying his array populating algorithm (in a manner that he acknowledges "seems to work best in practice") such that the matrix contains localised common substring information, potentially removing the resulting edit distance even further from the PD. It also prevents the system from correctly spotting that the performer has genuinely jumped a number of notes. Dannenberg's modification appears to make the assumption that the performer will not jump any notes, which is musically unsound. Semantically, wrong notes are better than extra or missing notes, but this modification is done at the expense of overall functionality.

Semantic Extensions

Wrong notes are musically more acceptable than missing or wrong notes. A performance containing wrong notes preserves the rhythmic (and to a certain extent, musical) structure of the score, which is lost when notes are (inadvertently) added or missed by the performer. This is also borne out by considering that the route through the matrix for a wrong note is the same as a correct note. Ascribing a wrong note with same value as a correct note when determining the length of a FLCS, effectively gives precedence to a wrong note error over a "correct" note that occurs much later having missed many notes in the score. The problem of knowing which notes are "wrong" when populating the matrix is overcome by exploiting the dependence matching systems have on the "correct" notes that follow the occurrence of an error.

Blip Sequencing

A series of "blips" are calculated on a separate matrix and occur at locations where the score and performance match. The MPD is realised by adding the blip sequence matrix to the FLCS matrix which modifies the route taken by the path finding routine. Additional rules, shown in Algorithm 2, and code to enforce the semantic rules, are added to the path finding routine, to cater for new scenarios created by the blip sequences.

Table 3 shows a matrix before addition of the blip sequence matrix (Table 4). Table 5 shows the blip sequences of Table 4 added to Table 3. This method

FLCS												
	a	b	d	e	f	g	f	c	d	e	f	g
a	$\boxed{6}$	5	5	5	5	5	5	5	4	3	2	1
c	5	$\boxed{5}$	$\boxed{5}$	$\boxed{5}$	$\boxed{5}$	$\boxed{5}$	$\boxed{5}$	$\boxed{5}$	4	3	2	1
d	4	4	4	4	4	4	4	4	$\boxed{4}$	3	2	1
e	3	3	3	3	3	3	3	3	3	$\boxed{3}$	2	1
f	2	2	2	2	2	2	2	2	2	2	$\boxed{2}$	1
g	1	1	1	1	1	1	1	1	1	1	1	$\boxed{1}$

Table 3: An FLCS showing the PD.

Blip Sequences												
	a	b	d	e	f	g	f	c	d	e	f	g
a	1	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	1	0	0	0	0
d	0	0	1	0	0	0	0	0	1	0	0	0
e	0	0	0	1	0	0	0	0	0	1	0	0
f	0	0	0	0	1	0	1	0	0	0	1	0
g	0	0	0	0	0	1	0	0	0	0	0	1

Table 4: The blip sequences for Table 3.

FLCS with added Blip sequences												
	a	b	d	e	f	g	f	c	d	e	f	g
a	$\boxed{7}$	5	5	5	5	5	5	5	4	3	2	1
c	5	$\boxed{5}$	5	5	5	5	5	6	4	3	2	1
d	4	4	$\boxed{5}$	4	4	4	4	4	5	3	2	1
e	3	3	3	$\boxed{4}$	3	3	3	3	3	4	2	1
f	2	2	2	2	$\boxed{3}$	2	3	2	2	2	3	1
g	1	1	1	1	1	$\boxed{2}$	1	1	1	1	1	2

Table 5: An FLCS showing the MPD.

```
if $M_{s+1,p+1} > M_{s+1,p}$ and $M_{s+1,p+1} > M_{s,p+1}$ then increment $s$ & $p$
if $M_{s,p+1} = M_{s+1,p+1}$ then increment $s$ & $p$
```

Algorithm 2: Additional path finding rules.

preserves the FLCSs meaning the system's ability to find a musical PD is not compromised.

Test Applet

A Java applet using the techniques described is available at:
http://cmt.gla.ac.uk/Software/software.html

Further Efficiency Considerations

Efficiency is further improved by delaying the use of the matrix until the occurrence of an error. A simple comparator with no error identifcation capability is used to compare the score and performance until an error occurs. At this point comparison operations are passed to the matrix routines. This prevents known "correct" notes from being included in the matrix which, depending on the position of the first error, can leading to considerable savings in computational overhead.

Conclusion

Having highlighted the short comings of using standard string comparison techniques in a musical context, we have presented techniques that overcome the problems encountered as a result of requirements imposed by such a context. These techniques can be used to provide semantically correct representations of a performance that allows for accurate score matching, leading to the implementation of, for example, an auto-accompanient system.

References

Nicholas J Bailey *et al.* (2005). Quantitative measurement of the reliability of automatic pitch detectors based on "performance distance". *In Press.*

Uzay Bora *et al.* (2000). A tool for comparison of piano performances. *Journal of New Music Research*, 29(1):pages 85–99.

Roger B. Dannenberg (1987/1988). Music understanding by computer. *Carnegie Mellon School of Computer Science*, pages 19–28.

Roger B. Dannenberg (1991). *Music, Language, Speech, and Brain,* Macmillan, chapter Recent Work in Real-Time music Understanding by Computer.

Hank Heijink *et al.* (2000). Make me a match: An evaluation of different approaches to score-performance matching. *Computer Music Journal,* 24(1):pages 43–56.

V.I. Levenshtein (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady,* 10(8):pages 707–710.

Bryan Pardo *et al.* (2002). Improved score following for acoustic performances. In *ICMC.* pages 262–265.

Esko Ukkonen (1985). Algorithms for approximate string matching. *Information and Control,* (64):pages 100–118.