AUTOMATIC MUSIC TRANSCRIPTION

BERK EKİM PAŞMAKOĞLU

IŞIK UNIVERSITY

2010

AUTOMATIC MUSIC TRANSCRIPTION

BERK EKİM PAŞMAKOĞLU

B.S., Computer Engineering, Işık University, 2007

Submitted to the Graduate School of Science and Engineering
in partial fulfillment of the requirement for the degree of
Master of Science
in
Computer Engineering

IŞIK UNIVERSITY
2010

IŞIK UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

AUTOMATIC MUSIC TRANSCRIPTION

BERK EKİM PAŞMAKOĞLU

APPROVED BY:

Prof. Ercan SOLAK (Thesis Supervisor)	(Işık University)	_____
Assist. Prof. Boray TEK	(Işık University)	_____
Assist. Prof. Ümit GÜZ	(Işık University)	_____

APPROVAL DATE:

AUTOMATIC MUSIC TRANSCRIPTION

Abstract

Computational music research is spread out of the world in many fields. One of these fields is automatic music transcription. During this thesis, we concentrated on the detection of music notes inside an audio signal. We decided to work on a percussive instrument i.e. piano because percussive onset can be relatively more easier to detect than other types of onset. We benefitted from the signal processing techniques like FFT, low-pass filtering and the statistical methods like Hinkley's CUSUM algorithm and linear regression. We proposed a transcription algorithm applied to a synthetically created audio data which was formed by the notes of middle octave and first five note value types. The algorithm transcribes the music scores with an average accuracy of 96,7 using the tuned parameters.

AUTOMATIC MUSIC TRANSCRIPTION

Özet

Bilişimsel müzik araştırması bir çok alanda dünyaya yayılmıştır. Bu alanlardan biri de özdevimli müzik çevriyazımıdır. Bu tez sırasında, bir ses iminin içerisindeki müzik notalarının algılanması üzerine yoğunlaştık. Bir vurmalı müzik aleti olan piano üzerine çalışmaya karar verdik çünkü vurmalı nota başlangıçlarının algılanılması diğer nota başlangıç tiplerine göre göreceli olarak daha kolaydır. Hızlı Fourier Dönüşüm'ü ve alçak geçirgen süzgeci gibi im işleme tekniklerinden ve Hinkley'in CUSUM algoritması ve doğrusal regresyon gibi sayımlama yöntemlerinden faydalandık. Orta oktav notalarından ve ilk beş nota değer türlerinden oluşan bireşimsel olarak yaratılmış bir ses verisine uygulanan bir algoritma teklif ettik. Algoritma müzik parçalarını ayarlanmış değiştirgeler kullanarak ortalama yüzde 96,7 bir doğrulukla yazılı biçime dönüştürmektedir.

Acknowledgements

My kindest and deepest thanks go to Prof. Ercan Solak. During the time I have been working on my thesis, he spent its time educating and guiding me by giving handy suggestions and keeping my thesis on an appropriate way. I would also like to thank him about his patience for my endless questions on the next step and my irrelevant interrogations about the subject of research.

I would like to thank Assist. Prof. Boray Tek, who helped immensely re-organizing the methodology of my thesis by addressing many questions and giving invaluable advice.

I would like to thank Assist. Prof. Ümit Güz for his recommendations of Savitzky–Golay smoothing filter offered for the segmentation of the note values smaller than sixteenth note. I also thank him for its advice about the part related to the previously done works that enrich my thesis.

This work would also not be possible without personal support from several people. I would like to thank all my mates in computer and electrical engineering departments. They helped me so much. I am very grateful for their examination substitutions. To Sevgi Dikmen, I owe a special debt, for her aids and advices at the thesis redaction.

Finally, I would like to thank my parents for their emotional support over the years and for everything they did for myself. They always encouraged me to go on. I would extremely like to thank Özlem Aslan for her kindest and gentlest sublime love, her amity and for keeping me going through times when it seemed like this project would never end. It is to you that this work is dedicated.

Table of Contents

List of Tables

List of Figures

List of Symbols

$Blocks$	n-by-2 matrix denotes the remaining attacks after progression
$b_{m,n}$	Squared error sum for K' notes of a test file
C_q	Centroid of a group
c	Constant used to tune the elimination threshold
d	Number of features
$disType$	Number of distinct note value types
$E(g\|\mathrm{X})$	Empirical error of the model for a given data set
Env	Amplitude Envelope of signal
e_i	Experimental error due to pitch estimation
f	Frequency
f'	Strongest frequency obtained by power spectrum analysis
f_0	Fundamental frequency
f_{note}	Lowest frequency for a pitched note
f_S	Sampling frequency
$g(\boldsymbol{x})$	Discriminant function which best represents the data , the model
H	Threshold used in Page-Hinkley stoping rule
$H_{LPF}(j\omega)$	Frequency response of a lowpass filter
h	Threshold value used in spurious attacks elimination
$IEnv$	Index array for amplitude envelope
ja	Jump amount of window
K	Number of segmented notes from a test file
K'	Number of remaining notes after spurious attacks eliminated
k	Number of approximation points
$leftB$	Peripheral boundary limits a group at left
l_i	Attack lenghth of ith note
lnd	Longest note duration

ltf	Look-up table frequency
$MapDur$	Mapped note durations
Max	Index array for local maxima
$MaxVal$	Corresponding smoothed envelope values for note attack
Min	Index array for local minima
$MinVal$	Corresponding smoothed envelope values for note finish
m	Median of attack lengths
$NAround$	Notes durations standing in a group peripheral boundaries
$NDur$	Calculated note durations
$NewInd$	Non-spurious attacks' indices after elimination
$NFFT$	Length of the Fast Fourier Transform
N_n	Minimum cumulative sum value used in Page-Hinkley stoping rule
$NName$	Array consisting note names of all K' notes
$Notes$	Look-up table for note names
$nbits$	Number of bits per sample used to encode the data in wave file
new_s_i	ith frame of the signal $s[n]$
new_t_i	ith frame of time vector $t[n]$
$Octaves$	12-by-9 matrix carrying 108 notes' pitches
$Onset$	Index array for physical onsets
$OnsetVal$	Corresponding smoothed envelope values for note onset
$Pitch$	Pitch array of all remaining K' notes
$periPer$	Scalar constant used to compute peripheral boundaries
$rightB$	Peripheral boundary limits a group at right
r^t	Second feature of the data set
$SEnv$	Smoothed version of the amplitude envelope
$S\{n\}$	Slope array
$s[n]$	Discrete-time representation of musical signal
T	Period
T_{note}	Period of for a pitched note with lowest frequency
$tempC$	Pseudo centroid of a group
$t[n]$	Time vector
Δt	Delta-time ticks for an event
U_M	Cumulative sums

$\boldsymbol{u}$	Index vector whose values denote where an onset detected
μ_0	Mean of the current data
$\boldsymbol{v}$	Index vector whose values denote where a local maximum detected
v_M	Jump magnitude
$\boldsymbol{w}$	Index vector whose values denote where a local minimum detected
ω	Angular frequency
w_1, w_0	Linear model parameters
ws	Window size used in amplitude envelope construction
ω_{cutoff}	Cutoff frequency
$\mathbf{X}$	Data set used in linear regression
x^t	First feature of the data set
Y	Output of FFT, the transform
y_i	Current slope value used in Hinkley' CUSUM

List of Abbreviations

$ASCII$	American Standard Code for Information Interchange
$CUSUM$	Hinkley's Cumulative Sum
DEC	Decimal
FFT	Fast Fourier Transform
HEX	Hexadecimal
IIR	Infinite Impulse Response Filter
LPF	Lowpass Filter
$MIDI$	Musical Instrument Digital Interface
$SMPTE$	Society of Motion Picture and Television Engineers

Chapter 1
Introduction

Computational music research is a well known and frequently studied area, especially after 70's. A lot of aspects of this field have been studied over the last three decades. Among the topics are music genre similarity, onset detection, audio classification, key-finding, beat-space segmentation, tempo and beat estimation. Automatic music transcription is also one of these areas.

Automatic transcription of music is the extraction of music events i.e. music notes for a given audio signal [1]. The transcription system usually concerns western monophonic or polyphonic music [4]. We can separate the task into monophonic and polyphonic transcriptions [2, 3]. Pieces which have only one instrument playing one note at a time are monophonic. Usually more than one instruments can be present in a piece of music by playing several notes at a time. In this case, polyphonic transcription aims to identify which instruments are played and transcribe which notes are played for a length of time [2]. In other words, main focus is to estimate the multiple fundamental frequencies (f_0) of several concurrent audio signals [3].

A technique used for monophonic transcription is the time-domain autocorrelation [4, 5]. For an N-length sample frame of an audio signal, autocorrelation measures the similarity between shifted versions of the waveform. The delay of the highest peak gives the period of the waveform. From the period, we obtain the pitch of the signal. That's why monophonic transcription is sometimes called pitch tracking [2]. The Fast Fourier Transform is used to compute the calculation of autocorrelation of a windowed signal [4]. A new, accurate, simple, fast and robust $f_0$estimator based on autocorrelation is introduced in [6]. Another spectral location type fundamental frequency estimation method is presented in [7].

Another method used for monophonic transcription is spectrum-autocorrelation [3]. The underlying idea of this technique is the spectral magnitude periodicity of harmonic audio signals [8]. The interval between the frequency components (harmonics) is investigated. The intervals may not remain constant. Nevertheless, they are still more stable than the locations of the components because they shift cumulatively [9]. With this method, even the sounds that have inharmonic characteristics allow to estimate f_0 [3].

1.1 Aim of Thesis

In this thesis, we would like to transcript the music scores from a given audio signal whose content is only constituted from the music notes. The pieces we work on do not contain any chords, rests between notes, or other type of music events. We create the test pieces with a program that we have written. In addition, we work only percussive instruments i.e. piano whose onset nature eases the detection of notes. Moreover, we try to find abrupt changes in the amplitude envelope of audio signal. Some of these sudden changes correspond to music notes. Overview of the complete system can be seen in Figure 1.1.

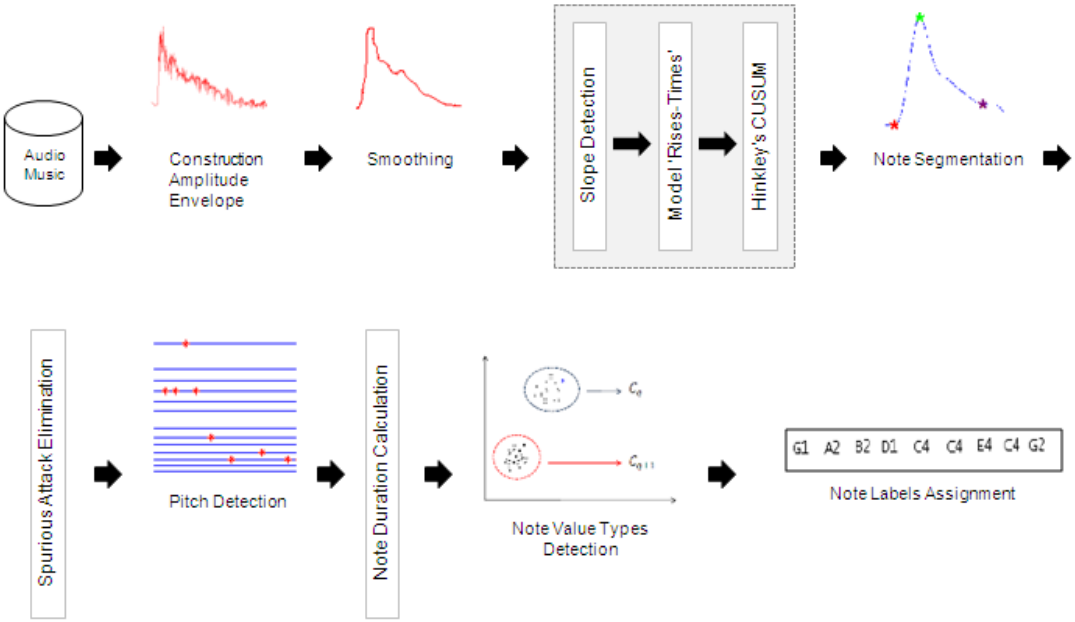

Figure 1.1: Overview of the system.

First, we smooth the envelope waveform by applying a low-pass filter. Then, we calculate slopes for the smoothed envelope by using linear regression. Afterward, we compute cumulative sums for slopes. Envelope, slopes and cumulative sums are together used in note segmentation. There may exist some spurious material in the segmented notes. We eliminate them by using a median filter. Segmented notes contain onset information where a percussive note begins in music. We detect the pitch of note by applying Fast Fourier Transform between successive note onsets. In the same manner, we estimate the note durations from the test data. In addition, we map note value types to the note durations by sorting them in ascending order in the time scale. Then, we assign the note labels to the notes detected. Finally, we compare the outcome with the original data in order to compute accuracy of our method.

1.2 Musical Background

1.2.1 What is music ?

Answering this question is not an easy task to do. Maybe a proper and usual dictionary definition about that is not acceptable for everyone. We can explain it by the actions we made when we are totally or partialy entitled to it. For example, we do it when we sing, play or compose, also we feel it when we hear, and of course we enjoy it, at least we try. More specific definition can be that music is a succession of sound tones arranged in a specific rhythm [10].

The thing that keeps in touch with us about music is just a sound, indeed. From its birth to its perception, it is a totally physical series of events. A musical instrument excites a vibration, and then this vibrating thing passes through a milieu, which is usually air, at the end it arrives to the human ear. After many operations conducted over it, the human auditory system make us perceive sound. It resembles meal prepared with plenty of ingredients such instruments, notes, scales, rhythm, tempo, intervals, pitch, frequency etc... We will investigate some of these ingredients in a different point of view as being the content appeared in a digital music and the way of their meaningful interpretations.

1.2.2 Sound

Sound is a wave propagating through the air from a source to destination [10]. Its vibrations travel and spread out around everywhere until they will be faded out. During this journey, vibrations sometimes change the milieu and encounter different elements of environment, for instance human beings. We can feel these propagating vibrations if we put our hands on music playing speaker. They travel from the speaker through the air to our hands.

To explain more precisely the motion of sound, we can call the aids of two important concepts which are time and displacement. To make a good observation, we usually need the method of comparison that necessitates a solid reference. As being a reference, time points to a measurable and observable acting way of the sound. In a given amount of time, we may observe that the sound wave starts getting an increasing value, after some amount of duration reaches its maximum and then returns back to its original starting point (see Figure 1.2). We may well call this motion as oscillation. All these parts of this motion can be identified or referenced in time scale. The corresponding time intervals of each parts show us the displacement of the sound wave. In the cartesian plane, displacement points to the y-ordinate and time refers to x-axis.

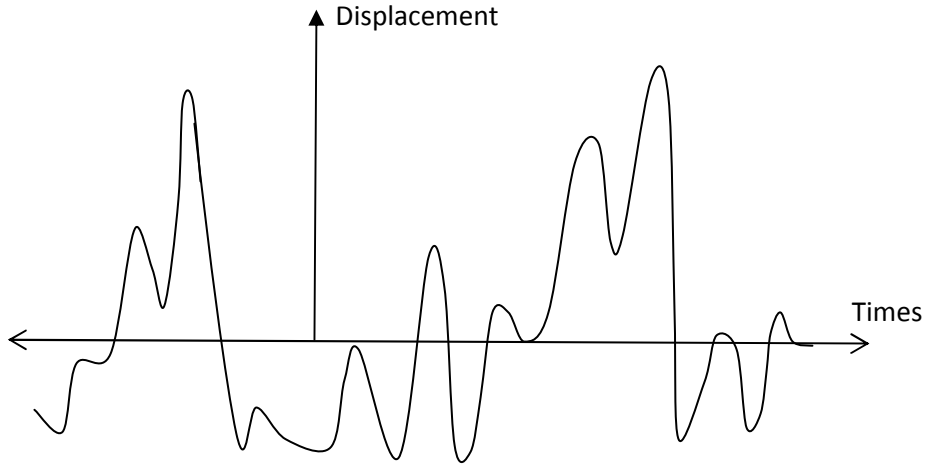

Figure 1.2: Oscillating sound waves and. displacement as a function of time .

1.2.3 Frequency, Period and Amplitude

Sometimes sound vibrations repeats themselves in a regular period of time (see Figure 1.3). For a given unit of time, there exists a number of vibration [10]. Normally, they are measured per second and standard unit is Hertz which means one vibration per second. When we denote 440 Hz, actually, we try to explain there exist 440 vibrations per second [11, 12].

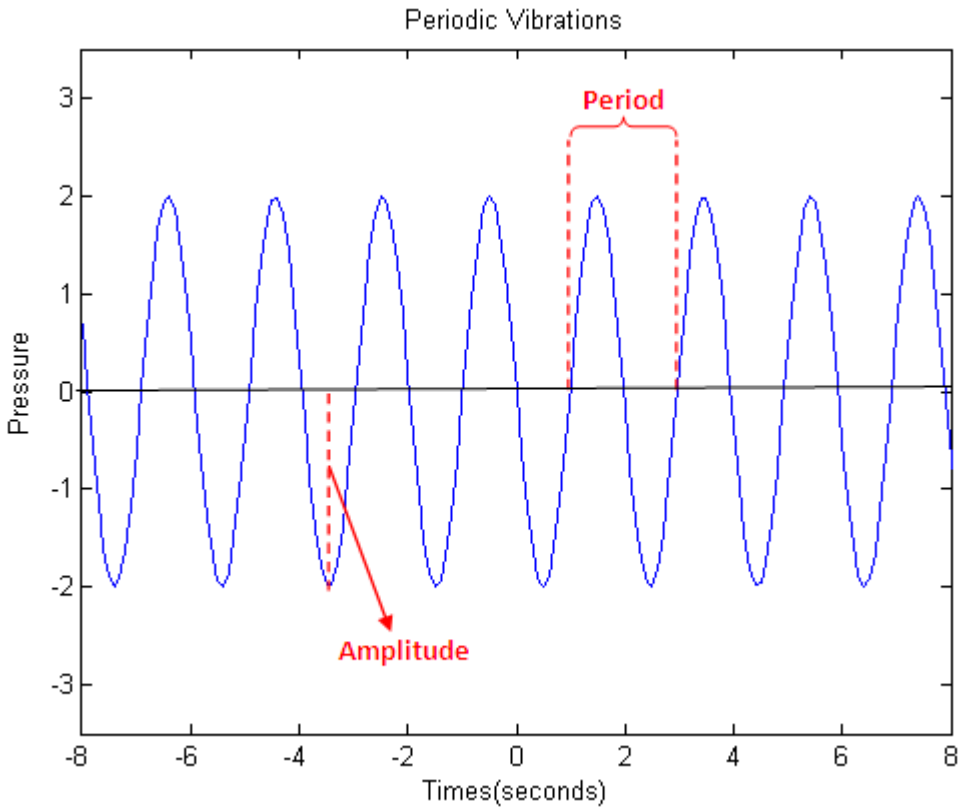

Figure 1.3: Periodic waveform.

If we mathematically express the relationship between period and frequency, we will notice that period is denoted as the inverse of frequency

$$T = \frac{1}{f} \ in\ seconds \tag{1.1}$$

where T is period and frequency is expressed by f. For this section, finally we introduce the concept of amplitude that we often utilize throughout the text. Amplitude is the maximum displacement of the vibration from the x-axis (see Figure 1.3).

1.2.4 Musical Note and Pitch

In the music theory, pitch is very essential in every sense [10]. First of all, musical notes are distinguished from each other according to their pitch values. In addition, their representation, insertion in the musical staff and transcription vary depending on this value [11]. Thus, it is important to explain these further.

1.2.4.1 Pitch

A tone is a sound that is played at a specific pitch. Pitch describes the specific frequency of a tone. Frequency of the tone is a measurement of how fast the sound wave propagates in the air [11]. Faster vibrations mean higher pitches. For example, middle A[1] or in the scientific pitch notation[2] A4 has a pitch of 440 Hz on the piano. The tones that have higher frequency pitch are called higher-pitched and the lower counterpart is lower-pitched [10, 11]. In the case of piano, it is clearly noticeable that from left to right sound becomes from less bass to more treble and the frequency keeps always increasing. Moreover, physical attributes of instruments can have an effect on the pitch distribution. Physically larger instruments usually produce lower-pitched tones, whereas smaller instruments produce higher-picthed tones [10]. This is because bigger instruments move more air than the smaller ones do, and more air means a lower pitch [11]. This is the why a flute produces higher notes than a tuba and why the thin strings on a guitar are higher pitched than the thick strings [11].

1.2.4.2 Musical Note

While composing music we can not use frequencies or in a more proper terminology, note pitches to write music because whole piece would be constituted from a bunch of numbers such as 587 Hz, 659Hz, 783Hz, etc... This will be very incomprehensible and prevents performing. Therefore, we need an easier way to designate tones that is

[1] Standard pitch (or tuning fork) is middle A. Its pitch is 440 Hz. All other notes are pitche according to this note.

[2] Scientific pitch notation serves to identify the specific pitch by placing a number after the note name. The lowest C on a grand piano is denotated by C1. The next C is C2 and so on. Middle C is C4 in this notation.

using literal notes for each pitch. The nomenclature of specific musical pitches uses the first seven letters of the alphabet A, B, C, D, E, F, and G. For instance, the note A always refers to a specific frequency in an octave. We will see details in the Section 1.5 later.

In music theory, to represent note pitches in a better and visual way, a graphical interface was invented: Musical staff has five lines and four spaces that each of them represent a specific pitch [11]. However this is not enough to identify all the notes pitches of a single instrument. The concept of clef is so vital because the pitches are determined by the type of clef at the beginning of the staff. A clef fixes the position of a single pitch in the staff and then from its position it is now possible to manage other pitches' places [10, 11]. For instance, the treble clef is positioned just above middle C(C4). The treble clef fixes the pitch G which is the second line on the staff. That is why this clef is also called as the G clef (see Figure 1.4).

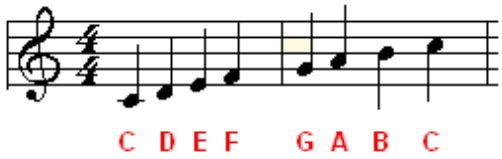

Figure 1.4: From middle C(C4) to an octave higher C(C5), the notes are placed inside the staff with the treble clef.

1.2.4.3 Note Value Types

There are seven main note value types which are whole, half, quarter, eighth, sixteenth, thirty-second and lastly sixty-fourth [11]. The context of value type is the duration of each single note type. Among them there is a simple mathematical relation that each note is, indeed, the half of the previous note type. By dividing the largest note seven times into two, we can reach the note value type of the smallest 64th note. See Figure 1.5 that summarizes first five of these value types.

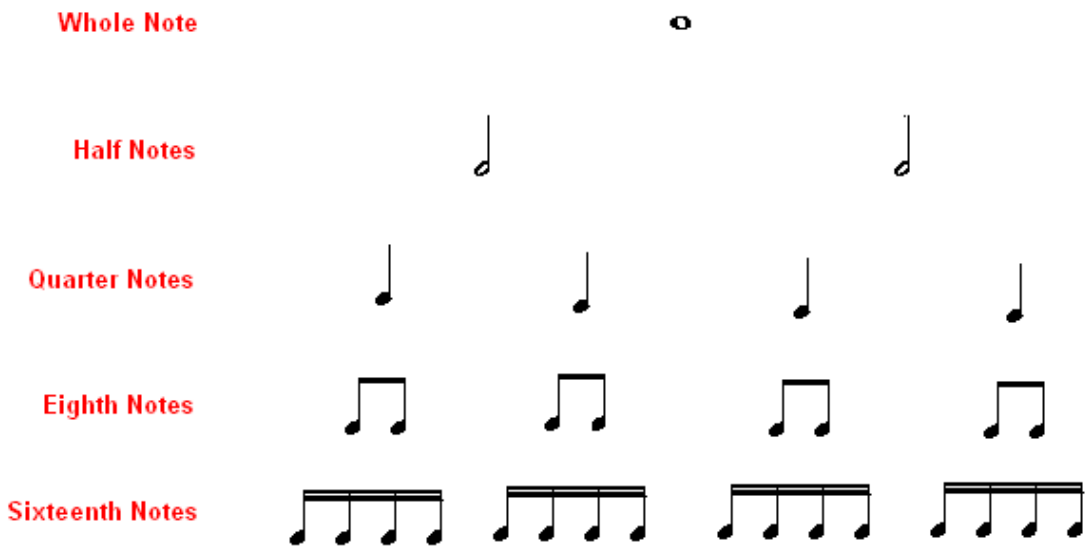

Figure 1.5: Five note value types in a hierarchical order from largest to smallest.

First of all, the whole note endures all along the measure as the name implies. Its representation on the staff is an only empty and oval notehead without a stem or flag attached. For example, a measure of 4/4 time, there will be the one whole note that fits the measure. This means a whole note's duration is equal to the sum of the four beats durations. In Section 2.7.1, we will see the concept of beat in details.

Secondly, when we divide the whole note, we get exactly the half note. Because a whole note lasts a whole measure, the half note lasts a half measure. This means a half note duration is equal to the sum of two beats durations. Thus, we can put two half notes in a measure. Its representation on the staff looks like the whole note but a stem[3] accompanies to the notehead this time.

Lastly, as being the half of the half note, a quarter note endures an only beat, we can fit four quarter notes in a measure if the time signature is 4/4. Its representation on the staff likes a half note with the notehead filled completely.

[3] If the notehead is on or above the third(middle) line of the staff, then the stem should point down from the notehead. If the notehead is below the third line of the staff, then the stem should point up from the notehead. See Figure 1-3's last two notes.

1.2.5 Intervals

There are some intervals between notes. If we can symbolize each note distinctly with a specific and constant frequency, then the ratio of these frequencies yields note interval. Let's have a look at the example of two notes having the frequency values 200Hz and 300Hz respectively. The interval ratio is 2 to 3. The table summarizes the ratio change for a given octave.

Table 1.1: Ratio change in an octave.

Note	Frequency	Semitones	Ratio
C3	130,81	0	1,0
C3#	138,59	1	1,05947558
D3	146,83	2	1,12246770
D3#	155,56	3	1,18920572
E3	164,81	4	1,25991897
F3	174,61	5	1,33483679
F3#	185	6	1,41426497
G3	196	7	1,49835640
G3#	207,65	8	1,58741686
A3	220	9	1,68182861
A3#	233,08	10	1,78182096
B3	246,94	11	1,88777616
C4	261,63	12	2,00007647

In the tunng system of western music, each octave has twelve equally "well-tempered" notes which means that an octave is divided into twelve equal semitones. The expression well-tempered refers to the fact that all the semitones are the same ratio [10]. Now, we have to answer the question of what is a semitone. A semitione is, indeed, such an interval between each note, the previous and the next, that gets always the same ratio of $\sqrt[12]{2}$ [13, 11]. Each such an interval is named as semitones. In addition, the total interval consisting of twelve semitones corresponds to a total frequency ratio of exactly 2 which is defined as octave (see Table 1.1).

In Table 1.1, first column shows the names of notes in the scientific notation. Second column shows the frequency values of each note. Third column shows the

corresponding semitone number for the current octave and last column shows the ratio of each semitone's frequency divided by the first semitone's frequency.

1.2.6 Scales

In Western music and pop music (pop songs) scales are so important and determines the characteristic of the piece's composition [13]. They describe which combination of notes will be used while composing the music. We can understand the importance of scale from the insist of frequent use of "diatonic scale" which dominated Western Music for more than 500 years [10, 13].

A music scale is a sequence of music notes within an octave from which a music piece is composed [13]. For instance, diatonic scale is a scale with 7 notes out of 12 notes. From the previous section, we know that an interval of 12 semitones yields an octave. Therefore, with this termonology, we can use the word semitone instead of saying notes. In diatonic scale, there exist seven semitones. These seven semitones-notes- in diatonic scale are C, D, E, F, G, A, B. These are also called "scale tones".

All scales start on one note and end on that same note one octave higher [11]. For example, every C scale starts on C and ends on C. The starting note gives the scale its name. In addition, diatonic scale has seven modes with each one using a different note as the starting note. Major scale and minor scale are the two most widely used modes of diatonic scale. In this scale, the major scale starts from the semitone C and the minor scale starts from the semitone A. Let's investigate the instance of C Major and A Minor scales. C Major scale consists of the semitones C, D, E, F, G, A, B and C(one octave higher), respectively. A Minor scale consists of the semitones A, B, C, D, E, F, G, A(one octave higher), respectively (see Figure 1.6). We will return back to C Major scale in the Chapter 2 when we discuss preparation of data with MIDI.

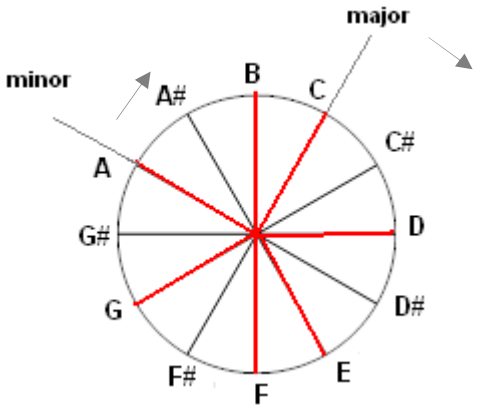

Figure 1.6: Red lines show diatonic scale semitones.

1.2.7 Rhythm of the Music

Rhythm is the task of counting. We count the beats of music [10, 11]. Each beat endures an amount of time. A group of a number of beats constitute the concept of measure. At the total, all the measures in a music piece must be equal each other. This is a rule in the composition of music. Each measure is exposed in the lines of a bar. Musicaly, all written notes are divided into bars. Each bar has the same duration. We can use both the words of bar and measure interchangeably.

1.2.7.1 Time Signature

The bar duration is designated by a time signature. The most common time signature mostly used in nearly all pop, jazz, rock and any kind of similar mainstream musical genres is 4/4 time signature. Beside this, there also exist some other mostly used such time signatures like 6/8, 3/4, 2/2, 12/8, 9/8, etc... [11]. Notation resembles to a fraction however it is not the case. Actually, a number is placed on top of another number. In overall, a time signature serves to signify that how many beats are there in a measure and what kind of note is used for representing any of these beats. The top number is the former and the bottom number is the latter. For instance, in the case of a time signature 4/4, in deed, a measure holds the equivalent of four quarter notes. The first 4 tells us that the duration of each bar or measure is divided into 4 beats. The second 4 in the signature specifies the length of the note. So a duration of a bar or measure is an equivalent of the total duration of four quarter notes per bar (see Figure 1.7).

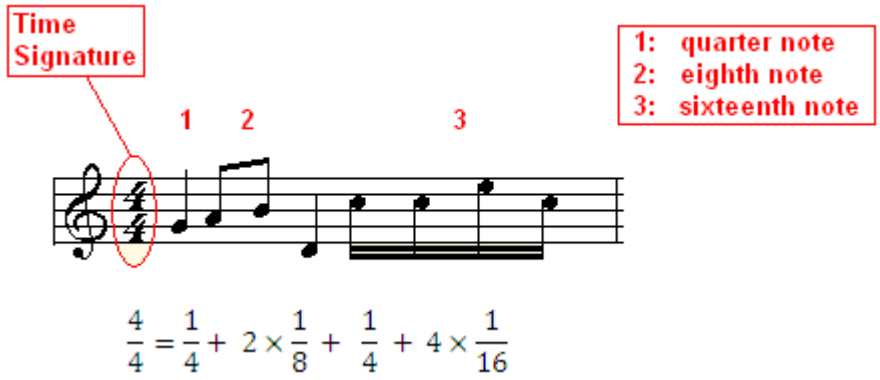

Figure 1.7: Valid bar of three kind of notes generated via Anvil Studio[4].

As a summary, because we have four beats in a measure, the top number in the time signature is a four and because the beat is a quarter note, the bottom number is a four. Moreover, using other note types for the beat is also common, i.e. eighth note in classical music [11], as well as the number of beats per bar may be supposed to be different i.e: 3, 8, 9. When we have an eighth note time signature likewise the examples of 3/8 and 6/8, a beat is an eighth note now. A measure has three eighth notes in 3/8 time signature.

1.2.7.2 Tempo

The tempo refers to the number of beats per unit of time. Normally the unit of time is minutes, so tempo is given as beats per minute, abbreviated as bpm [10]. For example, when we say that tempo is 80 with a time signature 4/4, this means that there will be 80 beats per minute or 80 quarter notes if all notes were played in this type.

What will happen when the tempo is changed to 100 with the same time signature? The definition of the length of any note is now changed, either. This time, we will play 100 beats per minute. If a beat is a quarter note as usual, now the length of quarter note was shortened a little bit because we have to play more notes than the former case but the unit of time still remains as the same before. To be able to satisfy this condition, every note has to be more quickier than before. Lastly, tempo often

[4] In Chapter 2, we explained how we used this programme in details.

varies gradually during the performance of a musical piece however we assume that in our study there is no such a change.

1.3 Organization of Thesis

The rest of the thesis is organized as follows:

Chapter 2 - Computer Music and MIDI This chapter provides an introduction to MIDI. We have used MIDI to create randomly generated music pieces with given runtime parameters. We test our algorithm provided in the Chapter 4 with these pieces. We explain MIDI file formats, MIDI events, variable length reading/writing with some details. We explain parts of MIDI specification that we have implemented.

Chapter 3 - Automatic Music Transcription In this chapter, we explain how automatic music transcription is performed. We introduce the techniques we used by supporting figures and tables. The topics mentioned are signal information retrieval, construction of amplitude envelope of signal, smoothing the envelope, slope detection, the Rise-Times Model, CUSUM algorithm, note segmentation, spurious attack elimination, pitch detection, note durations calculations, note value types detection, and note labels assignment.

Chapter 4 - Proposed Transcription Algorithm The techniques given in the Chapter 3 are not enough to produce a good understanding of how we could make automatic transcription. That is why we provide a pseudocode at first, then we explain each part with details supported in our implementation.

Chapter 5 - Results Each implemented part of the proposed algorithm in the previous chapter are investigated in details. We provide input/output snapshots of main implementation and data creator program. We review the details of each part of implementation by giving the rationale. We comment on runtime parameters, variable initial values, data structures used, and results. Finally, we explain our performance measurements for the tested data.

Chapter 6 – Conclusion In the final chapter, we summarize what we have done so far. We recap our methods and tools that we used. We discuss the results we obtained from our experiments. Lastly, we mention the future work to do.

Chapter 2
Computer Music and MIDI

After the electronic devices were involved so much in every day life, usual and traditional habits about any discipline have been changed. Such a discipline influenced so much from the new technological changes is music, of course. Through the end of 60's the electronic music appeared and spread out of the world [14]. The electronic instruments started to be used by the musicians instead of the old traditional ones that used to be played from centuries. Many manufacturers also appeared to supply the demand. In addition, during the middle of 80's, the use of personal computers tremendeously increased [14]. Accordingly, the use and need for software and multimedia applications rised, as well. Music was and still is one of the supporting pillars of multimedia applications. In such case of frequent use and need, there was a hole in that reproducibility, maintainability and exchange of any sort of sound or musical pieces was nearly impossible or required professional equipment or devices that synthesize musical audio information which requires both music theory and audio signal processing knowledge at the same time.

In such an environment, a consortium of two associations[5] came together and declared a standard interface describing musical events digitally that serve to exchange music. This standardized interface was MIDI which is the abbreviation of Musical Instrument Digital Interface [15]. MIDI is not about signal, sound, music or anything else. It imitates musical events and stores them in text-based manner in the standard midi file format. The digital music or sound we hear is sythtesised by the MIDI compatible sound card [16]. There are some application software that enables to create and perform music encoded by midi at the background with a handy graphical user interface that conceals the MIDI use at the back. These type of

[5] MIDI Manufacturers Association in Los Angeles from USA and Association of Musical Electronic Industry in Tokyo from Japan.

programs are called sequencer because they put in order the text-based MIDI messages as a sequence of events. We used such a sequencer Anvil Studio[6] for data validation and virtualization.

For testing our algorithm presented in the Chapter 6, we needed some randomly created wave file whose content can be manipulated in a programmatic approach by using runtime parameters. For this reason, we have written a small scale sequencer that can read and write midi events and also create an output MIDI file. We coded it using C programming language. We can consider the whole application as a little library constituted from 22 headers. In the rest of this chapter we will explain the MIDI file structure and some important aspects of MIDI standard that we coded in our program.

2.1 MIDI File Format

A MIDI file is a container for the text-based music events. With the players supporting MIDI on personal computers and other devices e.g. synthesizer, we can even play it and hear its content whenever we want. The file has a .mid extension. MIDI files are written bytes after bytes. Every bytes's content is an element of Extended ASCII table [17]. Every data piece that a sequencer process are manipulated in hexadecimal format while reading from and writing to a file. Every MIDI file has two important partitions: Header and Track chunks [18]. There can be only one header chunk in a MIDI file while the presence of numereous track chunks is possible at the same time in the same file.

2.1.1 Header Chunk

This is the beginning of the file. All MIDI files first 14 bytes is this header. Header is processed before all to provide information about how to process and execute the rest of the file content. Its constant structure has 14 bytes sized. The header itself is identified by a string marker which is "4D 54 68 64". These four bytes can be converted to a literal expression by using ASCII Table which produces "MThd" [19].

[6] http://www.anvilstudio.com/

This string is the same in all MIDI file as a standard. The next four bytes represent the size of the rest of the header chunk. Again, it is the same for all MIDI files, and is always designated by the literal expression “00 00 00 06”. Actually, this means that after read this part, we will encounter new information sized 6 bytes along. These new information are file format, number of tracks in file and the number of delta-time ticks per quarter note. These are all represented by two bytes.

Firstly, the file format can be one of the three options: single-track, multiple synchronous track or multiple asynchronous track. These are enumerated from 0 to 2. In our program, we use the second format type (enumerated by 1). Secondly, The number of tracks field is to warn MIDI sequencer that there will a job to define for a number of tracks. Because each track is delimited by a beginning and ending event, the number of tracks is very informative about how many delimiter will be encountered while playing. Lastly, number of delta-time ticks is time-base for the randomly created data. This number of ticks symbolize the duration of a quarter note in any tempo. We will give detailed explanations about this topic in the next sections of this chapter. See Figure 2.1 for a snapshot of our program.

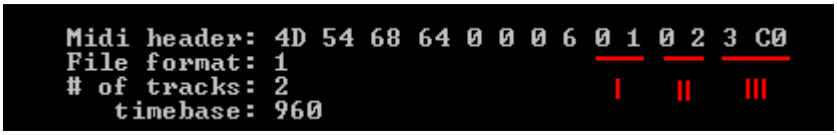

Figure 2.1: The header part for the file *generic-data.mid*.

On the first line of the snapshot, we see the hexadecimal header information. A byte of information can be represented by two characters at most. For instance, the two bytes “ 3 CO” is an hexadecimal number read in the file. The most significant digit is the left most character “3” which is written by using only a character. Mathematically, we can express the red roman number three of figure by conversion formula as following, $(3C0)_{\mathrm{HEX}} = (960)_{\mathrm{DEC}}$. This is a good example for the purpose to show that some part of MIDI has fixed length recording structure and some part has variable length recording structure. Header chunk is absolutely structured a fixed length record. The red roman numbers I and II of figure can be written as “1” and “2” in variable length recording because decimal numbers 1 and 2 are also hexadecimal numbers, either. In fact, with two bytes, we can create a largest

number in hexadecimal format as “FF FF”. However, in variable length recording, it has no limit e.g. FF FF FF FF FF FF... FF 7F and so on [17, 19]. See the Section 2.1.3 for detailed explanation.

2.1.2 Track Chunk

Track chunk consists of every kind of information about musical events directly or undirectly related. Its first eight bytes are constant and the rest may vary according to data in use. It begins with a four bytes long delimeter expression “4D 54 72 6B” whose translation is “MTrk”, literally [17, 18]. Moreover, next four bytes serves to express the data size of current track in terms of bytes. The events of a track chunk data can be separated into three main groups which are midi events, system exclusive events and meta events. For example, one of them is a meta-event and is responsible for finishing a track chunk. The finishing delimeter of a MIDI track which is fixed for all every track and expressed by the four bytes string “0 FF 2F 0”. To ease the understanding, see Figure 2.2 for the track chunk snapshot of our program.

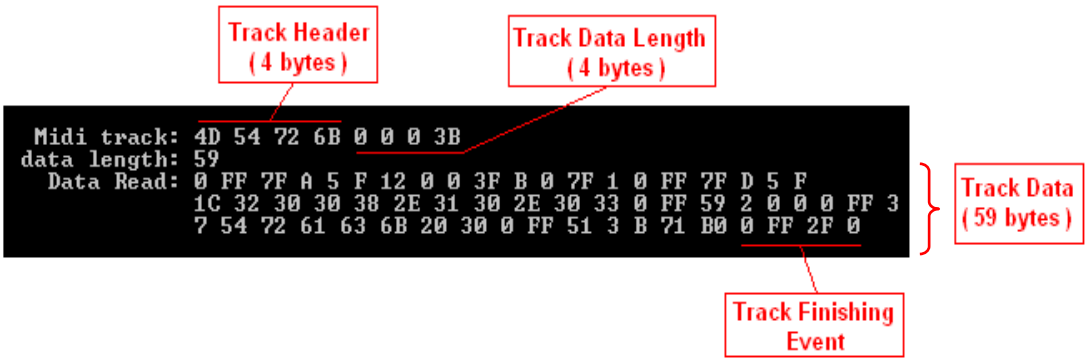

Figure 2.2: Track chunk snapshot provided with 59 bytes data.

In Figure 2.2, the notion of event is introduced, basically. The figure’s first line summarizes the fixed division of the track chunk. The second four bytes is the data length represented in hexadecimally. The conversion can be shown as $(3B)_{HEX} = (59)_{DEC}$. After the byte “3B”, the data is coming next and constituted from the accumulation of numerous different events. The finishing event is fixed and appeared as the last four ones out of 59 bytes. Finally, we use two track chunks: one is used to provide some meta events which are not related about MIDI and another chunk serves to make sequence of the midi events for the piece which is being created. This

second chunck can also contain meta events at the same time. Using two chunks was a design of our program. In this way, we simplified the algorithm and divided the event coding according to its context. In examples of the manuals we investigated, the selection of using more than two track chunks is also available.

2.1.3 Variable Length Reading and Writing

In the standard of MIDI, some numbers are structured as they have the most significant bit number seven as clear in their last byte while the rest of the byte series having their most significant bit number seven set. An example can be more explanatory using a single byte, we can write the largest number hexadecimally as "7F" which is equal to 127 and $(01111111)_{\text{BIN}}$ decimally and binarily, in given order. If we would like to generate the decimal number 128 in this manner, we will not use the hexadecimal equivalent as "80" [18]. In this way, this will cause a fault because this is a single byte representation and so that the bit seven is set. Instead, we will use two bytes in which the last byte will be "00" and the first byte will be "81" hexadecimally [19, 20]. To understand how we got these two numbers, see the Section 2.1.3.1. In Table 2.1, we provide some numbers generated as the variable length quantities in order to get familiar with. The decimal and hexadecimal equivalent are given before in order to explain clearly which number was trying to be expressed. We have written two procedures in our program for both reading and writing these types of numbers [17]. Their pseudocodes are provided in Appendix A. The most important area of use is expressing the delta times preceding every type of events in track chunk data. See the Section 2.2 for this.

Table 2.1: The variable length representation of some numbers.

Decimal	Hexadecimal	Variable-Length Representation
0	00	00
127	7F	7F
128	80	81 00
240	F0	81 70
480	01 E0	83 60
960	03 C0	87 40
8192	20 00	C0 00

2.1.3.1 Variable Length Writing

For a given integer number i.e. 960, we first make a bitwise AND operation with the fixed operand "0x7F [7]" and save the result in an integer variable, say R. Even the number is larger than 127, the bitwise AND operation yields an output whose bit seven will always be clear. Such a R's current value is 64(0x40) and this is the case. Then, we will make a bitwise RIGHT-SHIFT operation for 960 seven times whose meaning is dividing a number by 128. This operation comes up with a solution of 7 which is saved up in an integer variable say L. If L is greater than zero, a loop has started and we will make a LEFT-SHIFT operation for R eight times whose meaning is multiplicating a number by 256. This operation results in 16384(0X4000) for R and will be saved up in an integer variable say K. Next, we will make another AND operationd for L with the operand 0x7F and we get L again as a result. This intermediate step is saved up in an integer variable say I. Then, I is bitwise OR'ed by the operand 0x80 and this operation gives 0x87 as a result. Moreover, 0x87 is bitwise OR'ed with K and we get 0x4087. This is the current value of the data to write. We overwrite the value of L to the input number. We make a RIGHT-SHIFT operation for the new value of input number seven times. The result is assigned to L, and then L is checked with its new value as a condition of loop. The operation result yields zeros, as intended.. The loop is broken because the result is not greater than zero. If not, the same operations will be repeated. Before entering the loop, if the value of L is not greater than zero, the value of data to write will be found by only making a bitwise AND operation for the given input number by the operand 0x7F.

At the moment, we have only the data value to write in MIDI file. How can we write it down ? In a loop again, we first write the value of the data to write as one byte character inside a file by using putc() function of C programming language. This is an interesting property that C programming language's printing procedures can write a large number as one-byte character in a file by mapping it to ASCII table. For example, 0x4087(16519) will be written in one byte character representation as 0x87 in the file. Then, we make a bitwise AND operation for the data to write with the operand 0x80. If the result is greater than zero, the value of data to write will be

[7] This is the representation of hexadecimal numbers in C programming language environment. The literal prefix can be written as 0x or 0X. Both are the same thing.

bitwise RIGHT-SHIFT'ed eight times. As a result, we get 0x80 and the value is shifted by giving 64 as a result, decimally. 64 is also written in the file. By the way, writing 64 or 0x40 is the same thing because both of them correspond to the same element in ASCII table. Finally, 64 is bitwise AND'ed with the operand 0x80 and the loop is now broken because this operation yields zero. All the variable length writing is now completely accomplished. See Appendix A.1 for its pseudocode.

2.1.3.2 Variable Length Reading

First of all, we read an hexadecimal number in the file. We assign the value read to the variable V. We check whether V's bit seven is set or not by making a bitwise AND operation with 0x80 and we assigned the result to the variable R. If R is greater than one, we make following operations: First, we make bitwise AND operation for R with 0x7F. The result is assigned to both R and V. Second, we multiplicate V by 128 and result is assigned to L. Then, we read a new hexadecimal number in the file. The number is assigned to Y. We make bitwise AND operation for Y with 0x7F. The result is saved in R, this time. R and L is added and the result is assigned to V. Lastly, we make bitwise AND operation for Y with 0x80. If the result is greater than one we return back to the beginning of the second step. Otherwise, we return V as an output of the procedure. Before entering the first step, if R is not greater than one, we make bitwise AND operation for V with 0x7F and the result is assigned to V itself. Finally, we return V as an output of the procedure. See Appendix A.2 for its pseudocode.

2.2 The Events in MIDI

Events make the real job in MIDI. They are responsible for data exchanging, device controlling, time synchronizing, music events handling, and storing meta-event information. They reside only in the track chunk that's why we can call them as track events, either. We can generalize the events under three main categories which were mentioned in the Section 2.1.2. Before investigating these three event types separately, we would like to introduce the structure of it. An event has two internal subdivisions: the first one is delta-time and the second is event (any type). The first

subdivision can not be absent in any type of events because it is mandatory. However second division can vary according to the context. We can formulate events as

$$Track\ Data = \bigcup_{t=1}^{N} Track\ Event \tag{2.1}$$

and where a track event can be shown as

$$Track\ Event = \Delta t \langle\ midi \mid system\ exclusive \mid meta \rangle \tag{2.2}$$

where Δt is delta-time and $\langle\ midi \mid system\ exclusive \mid meta \rangle$ significates that after a delta-time only one type of events occur [16, 17]. The delta-time is a variable length quantity. It designates the amount of delta-time ticks for an event. On other words, it is the duration of an event in terms of SMPTE times [17, 18]. In addition, we did not implemented the system exclusive event type which was out of the context of our program that is why we skip this event type and did not give any explanation about it.

2.2.1 The Track Chunk MIDI Events

A midi event is a set of 4 pieces of information. These pieces are status byte, channel byte, data byte 1 and data byte 2, respectively. Its size is three bytes because first two bytes are merged into one byte. The status byte determines the MIDI message type. Its size is one byte whose most significant bit is one. It takes values like 0x80, 0x90, 0xA0, 0xB0, 0xC0, 0xD0, and 0xE0, respectively [21]. It is noticeable that only first most significant four bits are used in a status byte. In addition, the channel byte determines which channel will be used to receive and send the midi messages to a midi compatible device e.g sound card. There are 16 channels in MIDI. This amount can be enumerated from 0 to 15 [21]. In contrast with the status byte, the least significant four bytes are used in channel byte, this time. Thus, it clear that in both status and channel bytes, there are a total number of eight bits that are not used. To avoid redundacy, MIDI standard merged these two byte into one byte information. For instance, when we see an event like 0x93, we will easily understand that a note on event occurs and it is transmitted over the fourth channel. Other two bytes content can vary according to the status byte in use. However, in most cases, the first data

byte carries information about notes or we can say that it takes the number of midi note. The second data byte carries information about the velocity of the related note or we can say that it takes the volume intensity of the related note. We show the types of midi events that we have implemented in Table 2.2.

Table 2.2: The midi event types implemented in our program.

Status | Channel	Data Byte 1	Data Byte 2	Explanation
9x	Note number	Velocity	Note on event
8x	Note number	Velocity	Note off event
Cx	Instrument number	Not available	Control change event

We can summarize all the midi event types that we implemented with an example of our program. In Figure 2.3, we have all three mentioned midi event types. Program change event is responsible for the selection of instrument that will be used in the creation MIDI file. By default, our instrument is grand piano which is determined as zero in MIDI and we provided it as the third zero in the little yellow rectangle. This event has delta-time of zero which is shown as the first zero in the same rectangle. Generally, all the zero valued event are related to specify the properties of file out of the context of music events, directly. In addition, all such events are happenned and processed synchronously at the same time because the MIDI standard makes it possible.

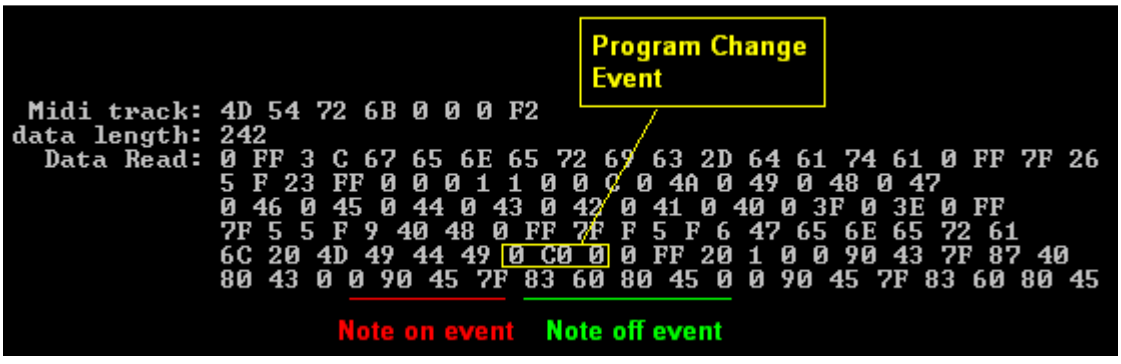

Figure 2.3: Three types of midi events for the second track chunk of the file *generic-data.mid*.

Note on and note off event are colored in red and green in the figure. Pay attention that the note on event's delta-time is zero for the midi note number 45 with the intesity 127 which is maximum. This event tells the sequencer to play the midi note 45 instantly. This process was now executed and has just finished. But a note is being

played, right now. This event keeps going on until the note off event comes take place after “83 60” delta-time ticks. The first two bytes are about the delta-time of note off event which is written in variable length recording format. The midi note 45 is the same with the one in note on event, as expected. However, the intensity of volume is zero that makes the note silent or totally terminated after the delta-time duration finishes. To conclude, we can say that every note begins with a note on event and finish with a note off event. This mechanism seems to the switch of an electic circuit.

2.2.2 The Meta Events

The meta event type carries complementary information which is not related directly to music or MIDI. There exist many meta events differing from each other but they have a common property that all of them start with the pattern “FF” which serves to distinguish that type of event from other two main types of events. Its structure can be formulated as

$$meta\ event = \Delta t\ (FF, identifier, data\ length, data\ ) \qquad (2.3)$$

where identifier defines the unique types of meta event [18, 21]. All identifiers are represented by one byte. The data length field serves to designate that there will be a data for this meta event with the given specific size. The sequencer process the data according to the given size. If it was not properly set, the track structure can be collapsed. The data part of each different meta event varies accordingly the context of it. We give detailed explanations about the meta events that we implemented[8], later. Table 2.3 recaps those events.

[8] There exist some other meta events that we implemented but not provided because we did not needed to use, later.

Table 2.3: The meta events type that we implemented.

Pattern	Identifier	Data Length	Data	Explanation
FF	0x03	N bytes	Text	Track name
FF	0x20	1 byte	Channel number	Channel prefix
FF	0x2F	0 bytes	Not available	End of track
FF	0x51	3 bytes	Time code	Tempo
FF	0x7F	N bytes	Text	Sequencer
FF	0x59	2 bytes	Scale information	Key Signature

We can summarize all the meta event types that we implemented with an example of our program (see Figure 2.4).

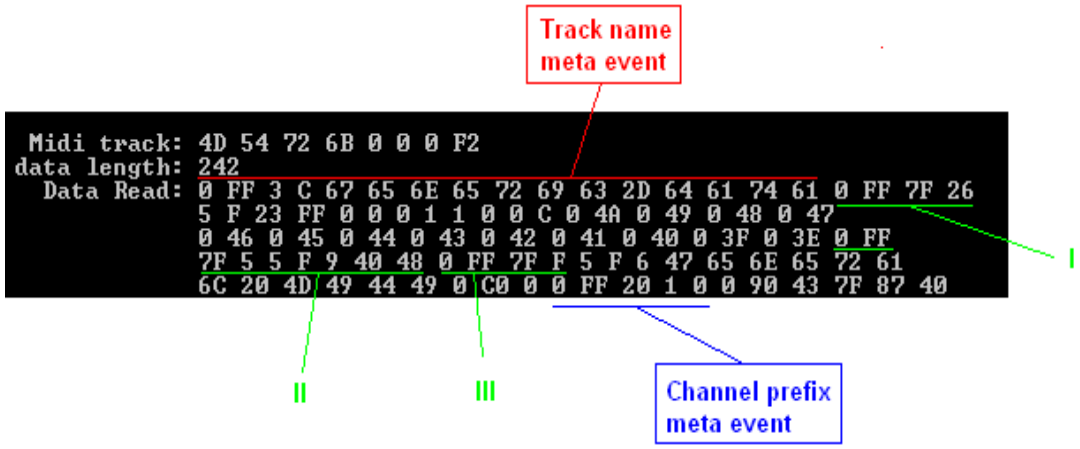

Figure 2.4: Track name, channel prefix and sequencer specific meta event used in the second track chunk.

In Figure 2.4, we see a track name meta event color in red which give the name "generic-data" to the second track chunk. Three sequencer specific meta events are colored in green and enumerated with roman numbers. They carry the information required for a midi file to be played in midi compatible players. For instance, the third sequencer specific event describes for which standard the MIDI file was prepared for. Its output is "General MIDI". Lastly, channel prefix event provides the information of which channel is in use. Its next event is a midi note on event that has not a channel information. In this way, channel prefix event serves to cope with a lack of information.

Chapter 3
Automatic Music Transcription

3.1 Getting Input Wav File

At the first spot, we have taken an acoustic music signal as an input which was required to be a .wav file for our implementation. The content of it carries only the information of one single instrument during the whole the record. The input wav file was prepared in the aid of a midi-creator tool which was also written by me using C programming language. In fact, this program codes a midi file content according to the given runtime arguments and produces a .mid extended output file. Usually, we create randomly generated 50 midi notes for testing purpose which come from the variety of nine different octaves of the acoustic piano. So, we have 108 possible notes every time we generate a note. The midi standard describes musical events in a text-based manner to aid exchange of musical interpretation between different musical instruments and different musicians. It provides a globally accepted interface for the devices processing the midi codes. As a result, we have only a raw data at the current phase of implementation. Indeed, we need to convert it into wave form via using a tool named ‘winamp’ which is a popular program used by end-users to listen to various audio music formats. We have pre-set this program not to play any audio format; instead we arranged it to convert the given input file to the wave equivalent by using an interior tool which is called ‘Nullsoft Disk Writer’. When the play is over, it outputs a wav file into the destination folder. At last, after passed various initial steps, the input of our main program is now ready to be tested. Before to conclude this part, we would like to tell about a file that is created by our data creator program concurrently it prepares the real test wave file. This text-based second output file can be considered as a ground-truth data which is used in validation. It carries all fully qualified note labels which belong to the notes created by the program. This file is named with the name of the test data file in order to establish

direct access while calculating the accuracy of algorithm after the last step. For instance, *test-5-50-notes.wav* emerges with *test-5-50-notes.txt*. The nomenclature is used for this text file except the file extension which must be ".txt", this time. Every such a file has a common part except its content. Every file begins with a starting label "scores: " which indicates that music scores are saved in this file. Then, an integer number which significates the number of music scores will be given after itself. In our data files, this number is fifty. A fragment belonging to the very beginnings of the file *test-5-50-notes.txt*. can be like that: "scores: 50 E2 G16 F16 D4 F8 G8 E4 D#2 C#16".

3.2 Signal Information Retrieval

After picking the wav file, we invoke the wavread() function of MATLAB to get some handy information from the wave form. We get several outputs as a result: the discrete-time representation of musical signal (sampled data) ($s[n]$), the sampling frequency (in Hertz) (f_S), and the number of bits per sample which was used to encode the data in the file ($nbits$).

3.2.1 Getting 'Times'

Clearly, we need the signal information to be expressed as a function of time but it is absent. We can build the accompagnying time information manually. First, we get the number of samples contained in the signal. Second, the sampling frequency signifies the number of samples taken per second. Finally, from the division of number of samples into the sampling frequency, we can maintain the signal's duration.

$$Duration\ of\ Signal = \frac{all\ samples}{sampling\ frequency}\ , in\ seconds \qquad (3.1)$$

For the given duration, we can now generate the linearly spaced (via the sampling frequency) time vector, $t[n]$, which starts from second zero and ends at the total of duration (in seconds).

3.2.2 Getting 'Notes' and 'Octaves'

We work on 108 notes' pitches which all represent fundamental frequencies on the piano, scattered as the organization of octaves. For the rapid data manipulation, we have stored them in a dynamically loaded matrix structure that MATLAB environment provides as .mat extended file. This is called $Octaves$ matrix that is constituted of twelve rows and nine columns and designate the twelve half tones of nine octaves starting from C to B each time (a complete octave) . The frequency range used here is between 16,35 Hz and 7902,13 Hz. Here we see in Figure 3.1 some part of piano keys and their corresponding fundamental frequencies [22].

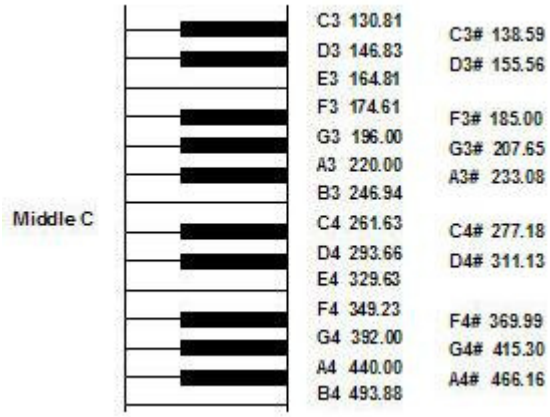

Figure 3.1: Pitches on the piano shown in the scientific notation.

3.3 Making Single Channel Signal

As an intermediate step, we have reduced the matrix $s[n]$ to a column vector by eliminating the second column. If we call the matrix with two columns as a stereo channel music wave, therefore we obtained a mono channel one since the representation of music signal is by default stereo in MATLAB environment. We assumed that the information in the second column of $s[n]$ is redundant to carry on for the further steps and the first column's information is sufficient to make appropriate processing. We can see signal versus times in Figure 3.2.

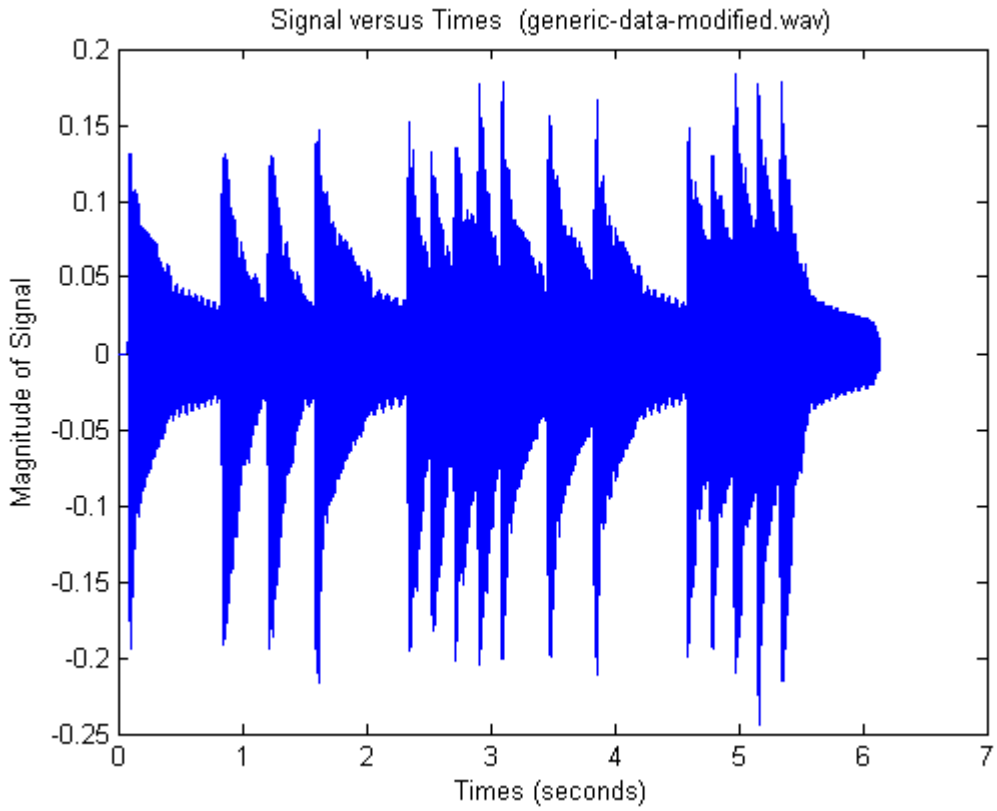

Figure 3.2: The sound waveform carrying the information of 16 notes.

3.4 Construction of Amplitude Envelope of Signal

At the second step, we construct a magnitude envelope of the signal. All along it, we traverse through the signal data a sliding window in which we search positive maximum [23]. The peaks below this amount are just ignored while calculating the envelope. Actually, the window represents just a proportion of the full data. Window size can be adjusted before the runtime. Beside this, the behaviour of the window traversal is quite worth of mentioning because it has an important role on the detection of right peaks. Each successive window is next to the last one. For this sake, we defined a jump amount which signifies the overlapping data number among these windows. In other words, the commencement of the further window is assigned by adding the jump amount to the previous windows's beginning. And current window's end is calculated according to the size of window. Moreover, size of window and jump amount can be represented (explained from this point further as ws and ja, respectively) by an unit of time or a number of indexes that a single window covers along the signal data. As intended, we save the locations where we

find the maximum valued peak in any window. After traversing the whole signal, we get a magnitude envelope Env and an index array $IEnv$.

In our implementation, we deal with the percussive sounds which are not generally periodic that's why the selection of the window size is independent from the cyclic structure of the periodic signals which repeats themselves in a definite period [24]. While working with periodic sounds, it is a wise choice to select a window which is one period long. However, in our case, a sufficiently large window size must be determined in order to satisfy both frequency and time resolution. If the window size is too large, we may not catch the peaks having meaningful magnitude values which can vary in a much smaller interval of time than the window can sense. Therefore, onset detection will skip and fail some notes in the further steps of implementation (i.e note segmentation) because time resolution is lost.

On the other hand, to cope with frequency resolution, the window size must be at least equal or greater than the duration of the lowest frequency- f_{note} -pitched note since the period of this note, T_{note}, is equal to

$$T_{note} = \frac{1}{f_{note}} \text{ seconds.} \tag{3.2}$$

In our case, the lowest possible f_{note} which is 16 Hz, scalarly, is the lowest audible pitch (for a healthy ear) in the human audible threshold [25]. It will usually a good choice to have $ws \geq T_{note}$ for tracing only the peaks [23]. This aids to get good results in the case of the test song having notes generated from one or two octaves with restricted note value types only allowed while the waveform has such a well-drawn shape seen like on Figure 3.3.

Contradictorily, this approach did not work fine for most of our test cases which were constituted randomly generated notes coming over nine octaves while all seven main note durations allowed. Instead, we approached setting the window size a little bit different that we made parameterized the window size as a scalar assigned to the construction of amplitude envelope process. A good one which we always used, was over at least four times greater than f_{note} with which we have gotten good results. At

the same time, we used an overlapping jump amount equal to the quarter of window size, by default. In Section 5.14, we provide lot of information about the reason of this selection regarding to the support of the experimental results that we obtained espescially in the third experiment we conducted (see Table 5.4, 5.5, and 5.6 with Figure 5.18 - 5.23). We have kept so rigid the overlapping amount in the sense of repeating nearly most part of the previous window content. We iterate the window one quarter at a time in our runs. We do not want to get only major peaks, beside, also to maintain the ones which are relatively minor. They could be not easy to notice at first sight. The relatively minor peaks must be caught in order to segment notes with too small durations such 32nd's or 64th's in the further steps. Figure 3.3 shows the signal's amplitude envelope-colored as red- which was constructed according to the mentioned method. The signal is colored as blue.

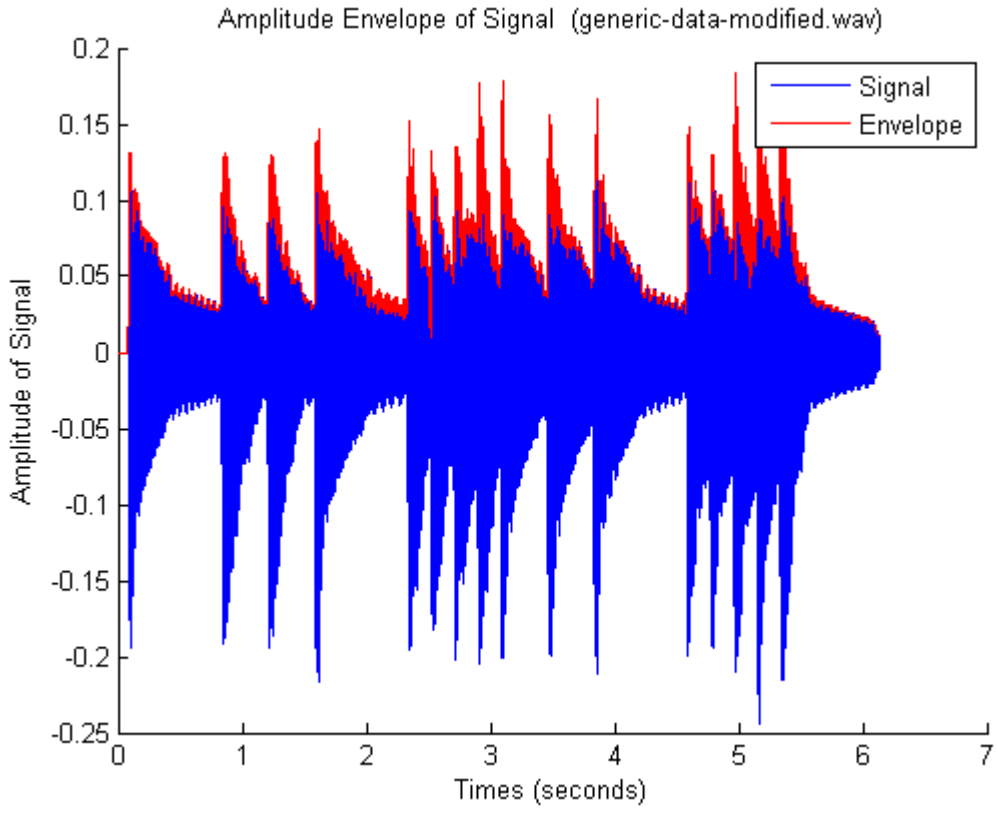

Figure 3.3: The amplitude envelope drawn onto the original waveform.

We provide the pseudocode for amplitude envelope construction in Figure 3.4.

```
COSTRUCT-AMPLITUDE-ENVELOPE-ALGORITHM( s[n], ws, ja)
00
01      # If not assigned, initialize to default values (64 and 16)
02              ws ← 64
03              ja ← 16
04      # Otherwise
05              continue
06
07      # Initialization
08      starting ← 1  and  ending ← ws
09      counter ← 1
10      Env  ← array[1  × counter]
11      IEnv  ← array[1  × counter]
12
13      # Traversing a window through signal
14      while  ending  ≤  length( s[n] )
15
16              # We try to find a maximum amplitude value in each window
17              window_counter ←  s( starting  to  ending )
18              maximum ← max( window_counter)
19
20              # Find the index of maximum valued window element
21              index ← find( maximum==window_counter )
22
23              # Save the value of maximum and its location in signal
24              Env(counter) ← maximum
25              IEnv(counter) ← starting  +  index  –  1
26
27              # Iterating the window boundaries
28              starting=starting  +  ja
29              ending=starting  +  ws
30
31              # Check out the array out of boundaries
32              if ending  >  length(s[n])  then
33                      break
34              end if
35
36              # Increment counter by 1
37              counter ← counter + 1
38      end while
39
40      # Return envelope and its indices
41      return Env and IEnv
```

Figure 3.4: Pseudocode of the amplitude envelope construction.

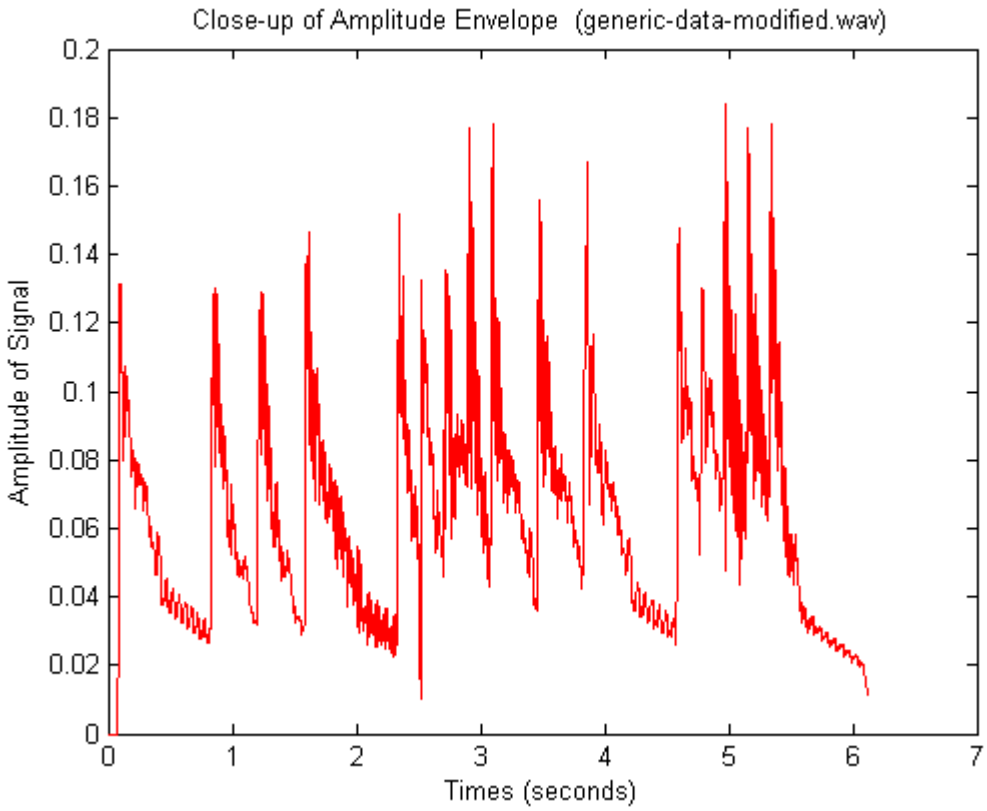

Figure 3.5: The amplitude envelope of $\boldsymbol{s}[\boldsymbol{n}]$ viewed alone.

3.5 Smoothing the Envelope Shape

In Figure 3.5, we see a massive amount of data that is quite far from being useful in detecting the notes. We have to reduce or basically simplify it by using some pre-processing methods like 'smoothing'. It is possible to notice easily that both the increase of the amplitude envelope and sudden change in the envelope probably means the occurence of an onset, especially for the percussive music containing strong percussive transients [26]. Here the onset detection function is an envelope follower in terms of time in order to be able to detect where and when the abrupt change occurs. This type of transient exhibits typical behaviour of percussive onset which will rise and reach its locally maximal peak value in a very small interval of time. Actually, we are dealing with one of the signal's temporal features that concern abrupt events happening in the signal.

The content of the signal can be differentiated as onset events and other abrupt events. This distinction is fully dependent to the ability of classification and internal

organization of the detection function. We have to focus more intensively on how the detection function chooses identifiable features to detect right onset events. One of these features is local maximum. However not all local maxima must to yield a musical note. Local maxima can vary in size and shape but not tremendously. Some local maximum is affected from noise, or the other kind of musical events like vibrato, tremolo, etc...[26] This type of amplitude modulation causes non-related peaks with the concurrent presence of event-related peaks. For all the reasons we have told above, the implementation requires a filtering of non-related peaks by smoothing the amplitude envelope. General convention is based on low-pass filtering. By applying this filter, we tend to make possible a simple and computationally efficient temporal method which lets clearly to pick the right peaks from the smoothed envelope for highly percussive onsets-transients or events.

What is a filter? Generally, a filter is a sytem that is designed to remove some component or modify some characteristics of a signal. Among these characteristics, we are interested especially in frequency, hence the filter that we will design has to be specialized on frequency. Among several options, there are three 'frequency selective' filters: *Lowpass, Highpass, and Bandpass Filters*.

All the listed filter types have a common feature that all of them may well remove certain frequencies while letting others pass through the system relatively unmodified [27]. For example, a highpass filter allows to be passed for high frequency components of a signal without changing them at the same time by killing all low frequencies [27]. Oppositely, low frequency components of a signal will pass through the filter unmodified while high frequencies will be completely eliminated if a lowpass filter is used [27]. Beside these, bandpass filter seems to be more selective than other two because a band of frequencies which are now allowed to be passed may possibly be pertained to some part of low and high frequency regions at the same time. All frequency components of a signal that fall into the range of the given band are passed unchanged through the filter while all other frequencies stayed out the range will be completely removed [27].

Let's have a look at the organization of lowpass filter. We can separate the frequency response of LPF into two regions or strictly saying two bands. These bands are passband and stopband, respectively. The response of the filter inside the boundaries of passband is one. Outside the boundaries, so it means in stopband region, its response is simply zero. The passband boundaries are determinated by a 'cutoff frequency'. All the stuff we mentioned about LPF until here is legitimate in the sense of a theoretical and ideal LPF. To ease the understanding, Figure 3.6 shows the behaviour of the ideal LPF.

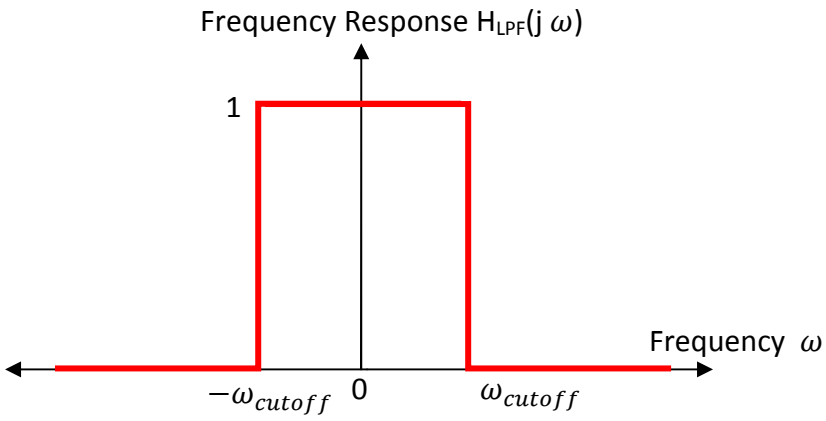

Figure 3.6: Frequency response of an ideal LPF with its cutoff at ω_{cutoff} .

An ideal lowpass filter is defined as

$$H_{LPF}(j\omega) = \begin{cases} 1 & |w| \leq \omega_{cutoff} \\ 0 & |w| > \omega_{cutoff} \end{cases} \tag{3.3}$$

where ω_{cutoff} is the cutoff frequency and $H_{LPF}(j\omega)$ is the response of the filter [28].

We have used a Butterworth IIR filter as a low-pass filter [27]. In MATLAB environment, we create a low-pass filter design with five arguments which are passband frequency,stopband frequency, passband ripple, stopband attenuation and sampling frequency. Then, the filter returned an output for the given input which was the amplitude envelope. The output is the smoothed version of the envelope. See Figure 3.5 and then Figure 3.6. The effect is apparent.

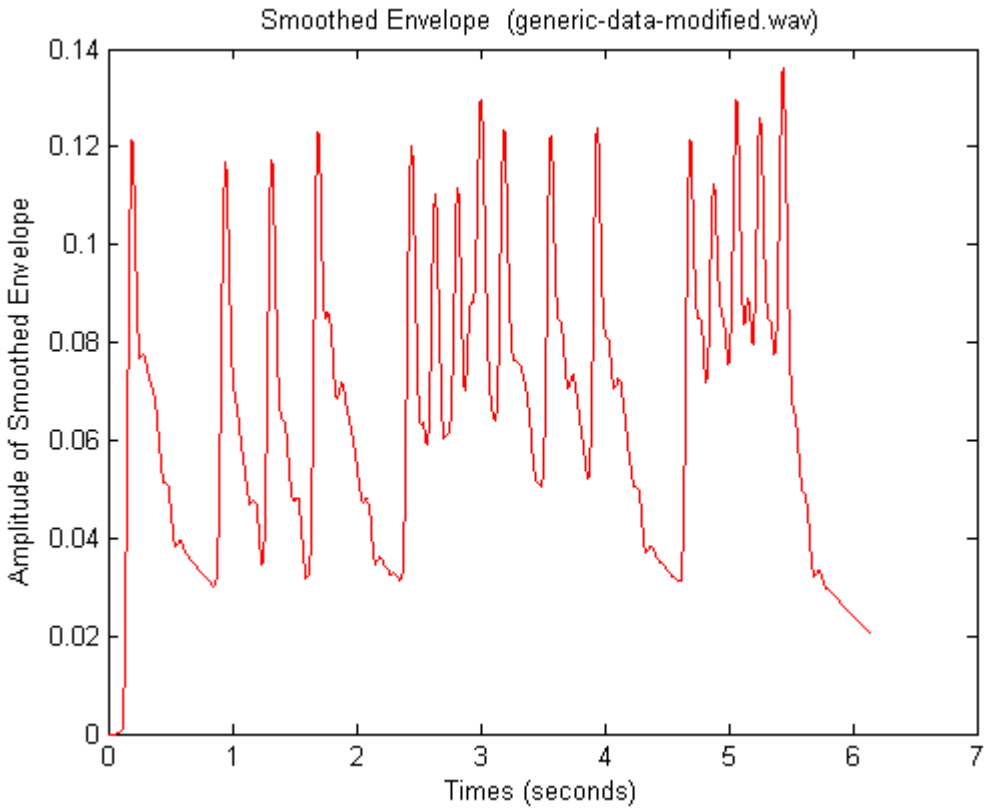

Figure 3.7: Smoothed envelope suitable for note segmentation.

In Figure 3.7, attacks are seen very clearly. This data is more convenient for segmentation than the previous one in Figure 3.5.

3.6 Slope Detector

At the previous step, we have smoothed the amplitude envelope and we obtained its smoothed version, $SEnv$. In this step, we use a slope detector. It is applied to the data of $SEnv$ by computing a linear regression over its several data points. The slope detector traverses through every data point of $SEnv$ one by one. It is proposed to use a number of points-say k points- at a time while approximating. These k points get the name "approximation points". In the literature, the proposed number of approximating must be at least four or eight at most [23]. The slope detector creates a sequence of overlapping line segments that float over the data. The slope of each line segment is saved up and forms a piece of the slope array $S\{n\}$. This array is very crucial and will be used in the few next phases of our implementation.

Basically, the idea of regression is to fit a curve to the data. Actually, we would like to approximate the output by using a model. In our case, linear regression is our model and we try to fit a line to our data which is $SEnv$. It must be considered the data as a separate data points distributed over the cartesian plane. Moreover, our data has a smoothed wave form of amplitude envelope containing percussive peaks. Actually, every smoothed envelope value can be identified as a function of time. So, the plane's horizontal axis is the array $t[n]$ and the vertical axis is the array $SEnv$. The demonstration of this data set in machine learning convention is $\mathrm{X} = \{x^t, r^t\}_1^N$ where x^t is $t[n]$ and r^t is $SEnv$.

In the machine learning literature, the model is structured as a discriminant function $g(\boldsymbol{x})$ which is considered as the best representative of the general trend or data set [29]. If the model in use fits the data very well, its approximation will be quite good and the model error will be very low. This is so called empirical error. We can define it for a given training set X as

$$E(g|\mathrm{X}) = \frac{1}{N}\sum_{t=1}^{N}[r^t - g(\boldsymbol{x}^t)]^2 \tag{3.4}$$

Here, the least square error function is used because we try to minize the empirical error [23]. The discriminant function must satisfy this constraint. The r and $g(\boldsymbol{x})$ have numeric values. For a good slope detector, best choice of discriminant function is a line equation which can be demostrated with the generalized equation

$$g(\boldsymbol{x}) = w_1 x_1 + \cdots + w_d x_d + w_0 = \sum_{j=1}^{d} w_j x_j + w_0 \tag{3.5}$$

where d is the number of features. In our case, d is one because we try to estimate only the value of $SEnv$. So, we can rewrite it as a linear equation as

$$g(\boldsymbol{x}) = w_1 \boldsymbol{x} + w_0 \tag{3.6}$$

where w_1 and w_0 are the parameters that we learn form the data.

Thus, the linear model parameters $w_1 \; and \; w_0$ should minimize the empirical error which can be shown as

$$E(w_1, w_0|\mathrm{X}) = \sum_{t=1}^{N} [r^t - (w_1 x + w_0)]^2 \tag{3.7}$$

The minimum value of the given empirical error above can be calculated by taking the partial derivatives of E with respect to $w_1 \; and \; w_0$then setting them equal to zero and solving for the two unknowns

$$w_1 = \frac{\sum_{t=1}^{N} x^t r^t - \overline{xr} N}{\sum_{t}^{N} (x^t)^2 - N\bar{x}^2} \tag{3.8}$$

$$w_0 = \bar{r} - w_1 \bar{x} \tag{3.9}$$

where

$$\bar{x} = \frac{\sum_{t=1}^{N} x^t}{N} \quad \text{and} \quad \bar{r} = \frac{\sum_{t=1}^{N} r^t}{N} \tag{3.10}$$

While running slope detector, we minimized the amount of data that linear regression used to get once when it is executed. Normally, linear regression or any kind of regression gets a complete data set like X. However this time, we separate our data set into pieces where each piece can contain only k approximation points. For these set of points, we try to find a line segment which best fits to the small data of k approximating points. The slope we want to find is w_1, actually.

In the machine learning literature, $\boldsymbol{x}$ is written in bold for the purpose of significating that $\boldsymbol{x}$ is a set of data. During the traversal of slope detector, we calculate a slope value for a set of data points $\boldsymbol{x}$ starting from the actual data point and ending with the data point which is k index further from the actual data point. For example, if k is eight and the current data index is i, we will calculate a slope of line approximated for these eight points denoted with the pair of indices $(i, i + k - 1)$. After computing the intercept value, w_0, we can even draw the line for the given set of k points over

the main data in order to see if $g(\boldsymbol{x})$ does really follow the trend and fits to the data, too.

The more the index i incremented, the more the new lines will appear which are overlapped since the new approximation will be proceeeded between the data points indexes expressed as $(i+1,\ i+1+k-1)$. This shows that nearly most of the two successive set of k points ultimately overlaps each other. Therefore, the lines derived from the two or more successive set of k are overlapped each other, as well. Now, it is possible to see that those lines are very good follower of the waveform of $SEnv$ (see Figure 3.8).

In Figure 3.8, The result of slope detector is shown. Firstly, the top part of the plot shows an excerpt of two peaks from the data of $SENV$ in a time interval between 4,325th second and 4,37th second. At the bottom part, it is seen the zoomed version inside the squared area. The lines computed during slope detection are drawn in black and fit very well to the data. Data points pertaining to $SENV$ are shown in red asteriks.

To conclude this part, we can add the point that during the traversal, linear regression calculation is repeated each time for a new set of approximation points. Even though each separate calculation made in each linear regression iteration for a set of k points can be assumed as they are executed in a constant time, the number of repetition can affect the performance especially for a long music piece. This case happens because we linearly traverse the data and at each index we repeat the same number of operations. For an amplitude envelope length L in terms of indexes that it consists of, the runtime of this algorithm is $O(L)$, asymptotically. The pseudocode of the algorithm is given in Figure 3.9.

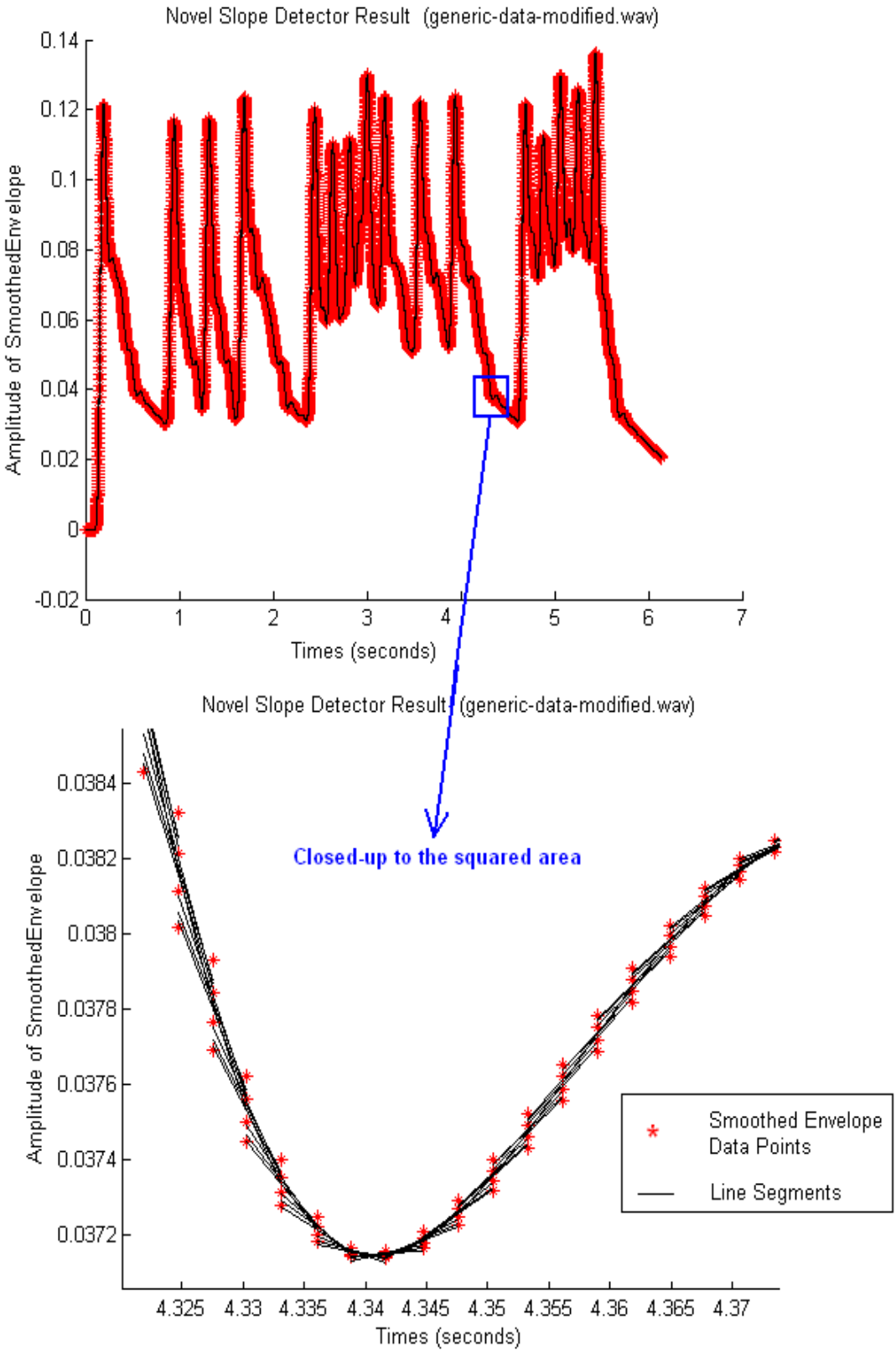

Figure 3.8: Slope detection's result seen. The top part of the plot shows the squared area of $SENV$. At the bottom part, the squared area is zoomed.

```
RUN-NOVEL-SLOPE-DETECTOR-ALGORITHM( SEnv, t[IEnv], k )
01
02      # Check 4 ≤ k ≤ 8. Otherwise k is assigned to default as 4
03      if condition fails then
04              k ← 4
05
06      # Initialization slopes array
07      counter ← 1  and  index ← 1
08      S{} ← array[1 × counter]
09      Intercepts ← array[1 × counter]
10
11      # Calculating linear regression for a data set with k points of SEnv
12      while index ≤ length(t)  and  index + k ≤ length(t)
13
14              # y is a k-points data set piece and x is time correspondent
15              x ← array t( index to index + k - 1  )
16              y ← array SEnv ( index to index + k - 1  )
17
18              # xy is a vector whose elements are the result of the
19              # the elementwise multiplication of x and y
20              xy ← x.*y
21
22              # x2 and y2 are arrays created in the same manner
23              x2 ← x.*x
24              y2 ← y.*y
25
26              # The series of some scalar values which is found by summation
27              sumx  ← sum(x)
28              sumy  ← sum(y)
29              sumxy  ← sum(xy)
30              sumx2  ← sum(x2)
31              sumy2  ← sum(y2)
32
33              # squares of the scalar values sumx and sumy are computed
34              squaresumx  ← sumx * sumx
35              squaresumy  ← sumy * sumy
36
37              # a slope is calculated
38              m  ← [ (k * sumxy) - (sumx * sumy) ] / [ (k * sumx2) - squaresumx ]
39              S{ counter }  ← m
40
41              # an intercept is calculated
42              intercept  ← [ sumy - ( m * sumx ) ] / k
43              Intercepts( counter )  ← intercept
44
45              # increment the values of counter and index
46              counter ← counter + 1
47              index ← index + 1
48      end while
49
50      # Slope array is returned
51      return  S{n}
```

Figure 3.9: Pseudocode of the slope detection.

3.7 The Model 'Rise-Time'

With a smoothed envelope as in Figure 3.7, we can easily concentrate on the attacks events. When an attack happens, energy change per unit time change [24, 26]. Also, the slope of the rising peaks change, either [24]. This case generally occurs at the onset for the percussive music. Remember that a note onset is a place that intensity of amplitude is very low. Suddenly, the intensity tremendously starts to increase. We call this time as 'rise-time' [24]. Piano is an instrument that has very quick attack near its onset [24]. We can model this behaviour. The model will have two parameters: the 'rise-time' and the 'rise-slope'. When a 'rise-time' occurs, we could find some attacks that their intensities are always higher than a threshold value [24]. Usually, this threshold can be tuned according to the need. In such a case, the attack or the peak gets its maximum value. This is called 'time-of-max'. We pick these peaks when they reach their time-of-max. In addition, we can use the 'rise-slope' parameter. We have seen how the slopes are obtained with a slope detector in Section 3.6. At the time a percussive attack occurs, the amount of change in the slope mean ultimately increased, too. If we can find such a point that a sudden change occurs in the slope mean we can identify the note onset. This point is called 'rise-slope'. When a rise time has just started, the slope value gets a very big value. However, as well as the intensity of the attack increases, slopes begin to loose its intensity. Just after the time-of-max, the slope changes its sign. The attack intensity begins to decrease after this point.

Because we have all the data, we can search and find for the slopes values which are greater than a given threshold i.e. 0,016. In fact, these slopes values correspond to the attacks (see Figure 3.10).

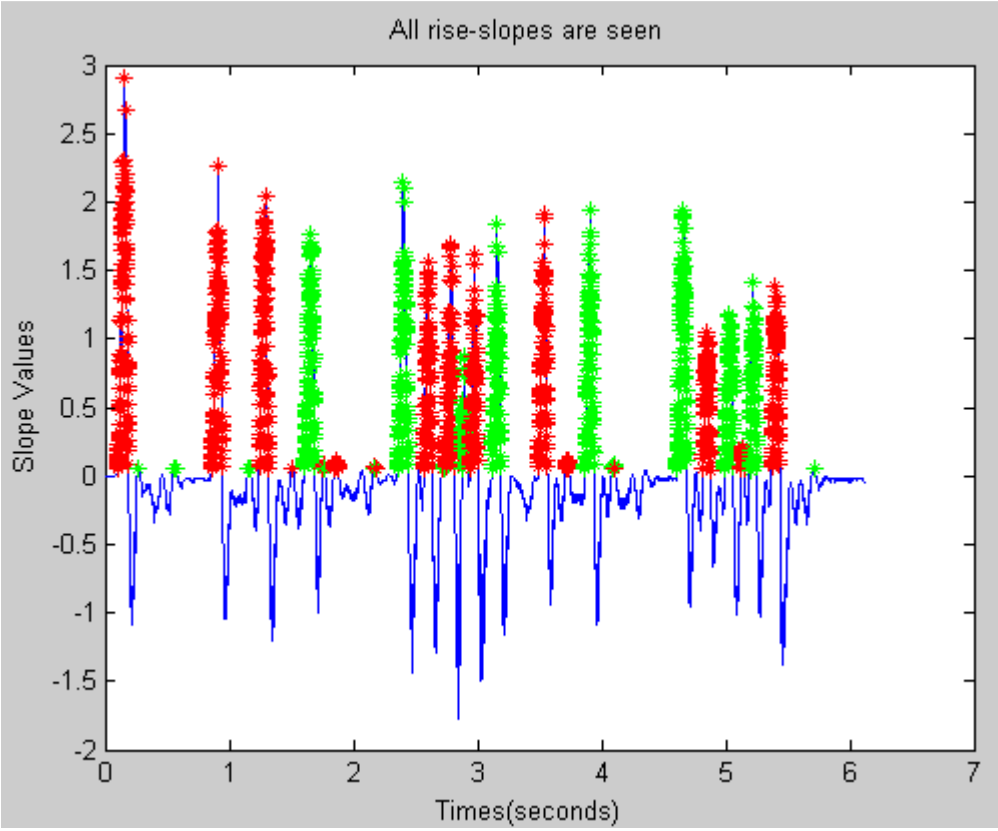

Figure 3.10: All rise-slope values above 0,016 seen and colored as red and green, alternatively.

In Figure 3.10, all rise-slopes are shown. To differentiate them from each other, they are colored in an alternative modulation of red and green. It is clearly seen that some slopes values above the threshold can not be related to attacks.. They have very little values when comparing other major slope peaks. However, we collect all the attacks. When we draw all these attacks over the waveform of the smoothed envelope we got in Section 3.5, we notice that they are perfectly placed between the onset and time-of-max (see Figure 3.11).

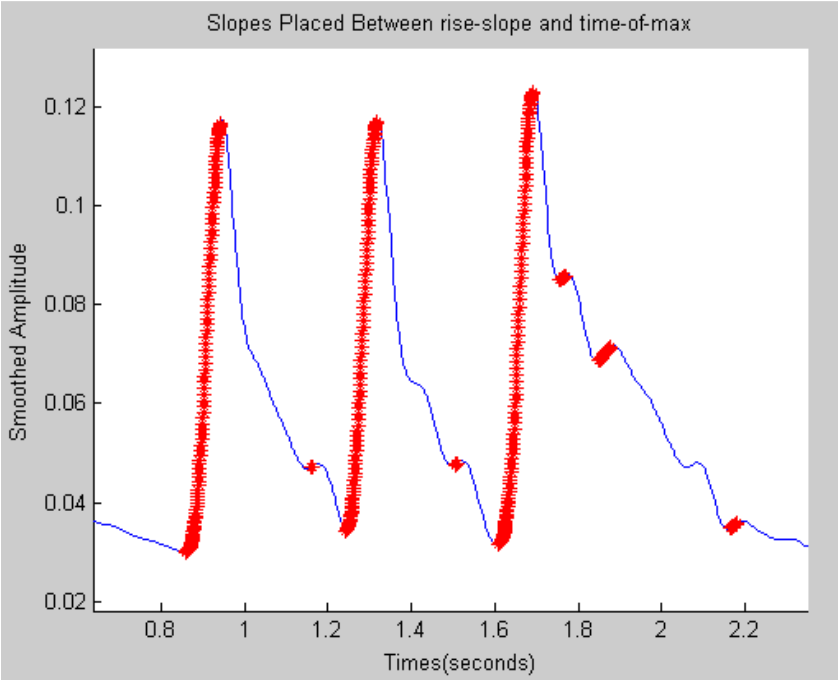

Figure 3.11: Slopes placed between rise-slope and time-of-max.

There must be much more attacks than the number of notes that we would like to detect. In Figure 3.11, we see three long attack covered by slopes values. These are suitable for detection. However, the very short one must be cleaned. They can cause in false detection. In the figure, red asteriks symbolize the slopes data points.

3.7.1 Progressing the Model

To avoid the false detection, most of these short attacks will be eliminated by a simple approach. We will compute arithmetic means for all slope blocks in Figure 3.10. Then, we find the maximum valued mean among them. We normalize all of them with this maximum value. By using a mean threshold this time, we get rid of all slopes blocks whose values are less than the threshold. Probably, a threshold of 0,1 may be sufficient for this kind of elimination. Afterward, the short slope blocks are eliminated. This means that only related attacks remain (see Figure 3.12). This improvement is our contribution to the model. We denote these remaining attacks by a n-by-2 matrix, $Blocks$. First column of this matrix carries the starting index information of an attack and second column carries the ending index information of that same attack. All indices information correspond to the position of attacks in the slope array $S\{n\}$.

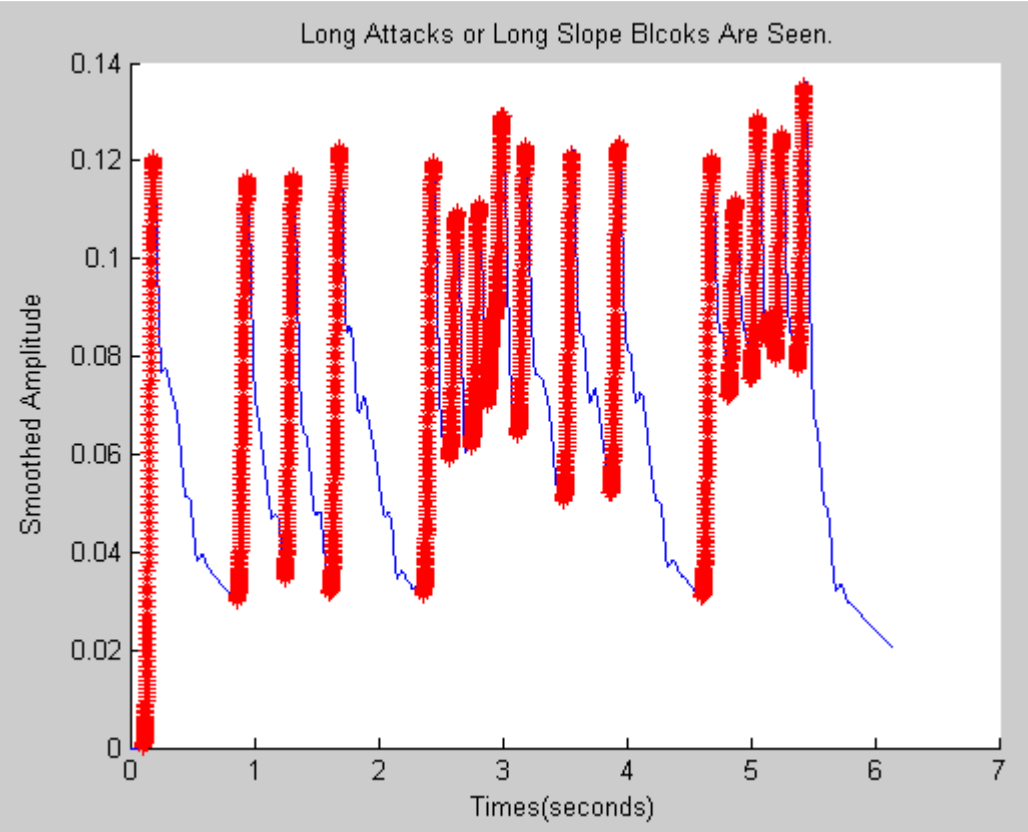

Figure 3.12: The real attacks after progressing the model.

3.8 CUSUM Algorithm

To segment the notes from the smoothed envelope, we will need cumulative sums of slopes. We search a point whereas a sudden change in the slope occurs. In statistics, this problem is described as the problem of a change in mean or described as the detection of jumps in mean [30, 31]. One of the important algorithms offered to solve this problem is Hinkley's cumulative sum [30].

The algorithm searches for jumps in the mean. It tries to find deviations with respect of the cumulative sum [30]. Hinkley offered two detectors: One is responsible for downward jumps and other one is the responsible for upward jumps. We are interested in the second one because we are in quest of a sudden increase of slopes. The detector formula can be given by

$$U_M = \sum_{i=1}^{N} \left(y_i - \mu_0 - \frac{v_M}{2} \right) \tag{3.11}$$

where U_M is cumulative sums and y_i is current data and v_M is the jump magnitude and μ_0 is the mean of the current data [30]. For i=0, μ_0 is equal to zero. The jump magnitude can be shown as $v_M = \mu_{i+1} - \mu_i$.

In order to find the point where an abrupt change (increase or upward jump) occurs, we have to apply Page-Hinkley stoping rule [31]. This rule checks for every U_M that the susbstraction of this value from the minimum cumulative sum is whether greater than a threshold value or not. Usually, such a threshold is given as H and gets a value like 0.00001. If this case holds, the index which denotes the current value of cumulative sum is saved to locate where the abrupt change once occurred. We can recap the test we apply by a formula as follows

$$\text{test the case for } U_n - N_n \geq H \tag{3.12}$$

where $N_n = \min_{0 \leq k \leq n} U_k$.

3.9 Note Segmentation

So far, we obtained a smoothed envelope, a slope array for this envelope, cumulative sums, and attacks gotten from the 'Rise-Times' Model. Segmentation uses all the information mentioned above. We apply a procedure which has three partitions. They are proceeded in order for each note. The first partition is about to find abrupt change in the slope array with the aid of cumulative sums. The second partition aims to find local maxima values for every attack. And the third step, we would like to find local minima values. Overview of three note segmentation phases is seen in Figure 3.13.

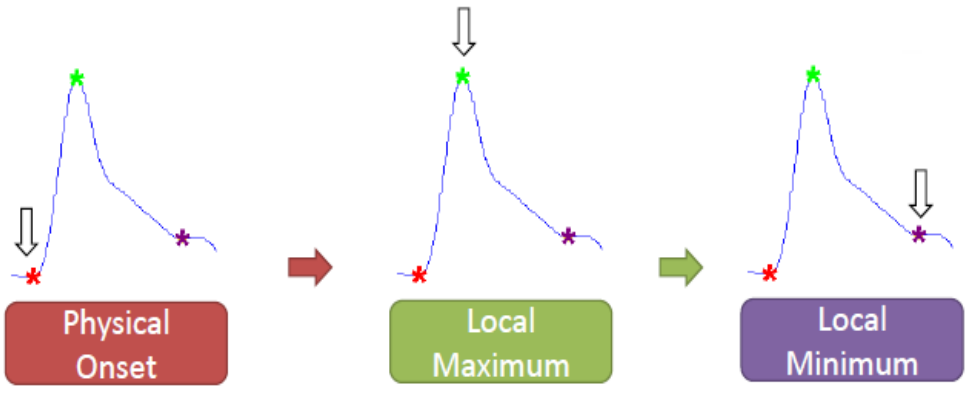

Figure 3.13: Three phases of single note segmentation process.

To find abrupt change in the slope array, we test the Hinkley's stopping rule for a given attack and a treshold value [26, 23]. For the attack detected from the step described in Section 3.7, we try to find a jump point. If the test holds for an index u located in the cumulative sums, we save this location in the array $Onset$ as the onset index of the note segmenting. In fact, this $Onset$ value is obtained by the expression of the amplitude envelope indices array as a function of the index u. We can formulate the case as follows

$$Onset_i = IENV(u) \tag{3.13}$$

where i symbolizes the currently segmenting note.

Then, we try to find time-of-max for the segmenting attack. From the index u, we begin to search in the slope array a location where the slope change its sign. We test the current slope data for a given epsilon value e.g. 0.00001. We can show the test as follows

$$S_{n-1} > \epsilon \; and \; S_n < -\epsilon \tag{3.14}$$

where S_{n-1} and S_n are the slope value [23]. In fact, the local maximum value is the most mature part of the attack transient. From this point, it begins to loose its intensity [26]. We save the index location v for the slope array whereas the test holds. By putting the index v inside the array $IENV$, we express the index position in terms of amplitude envelope indices. We store all these expressions for all segmenting notes inside the array Max. We can formulate the case as follows

$$Max_i = IENV(v) \tag{3.15}$$

Finally, we try to find a local minimum that the note segmenting starts to fade out. Before detecting the next onset, we absolutely need this information for the appropriate onset estimation. From the index v, we begin to search in the slope array a location where the slope change its sign. This time, we make the same test for the current slope data but the sign of slope will be positive [23]. We can show the test as follows

$$S_{n-1} < -\epsilon \; and \; S_n > \epsilon \tag{3.16}$$

We save the index location w for the slope array whereas the test holds. The expression of w in terms of amplitude envelope indices is stored in an array Min for all segmenting notes. We can formulate the case as follows

$$Min_i = IENV(w) \tag{3.17}$$

Therefore, an iteration of the note segmentation phase was completed. We repeat these three steps until all attacks detected by using the model Rise-Times are segmented. See Figure 3.14 for the segmented notes.

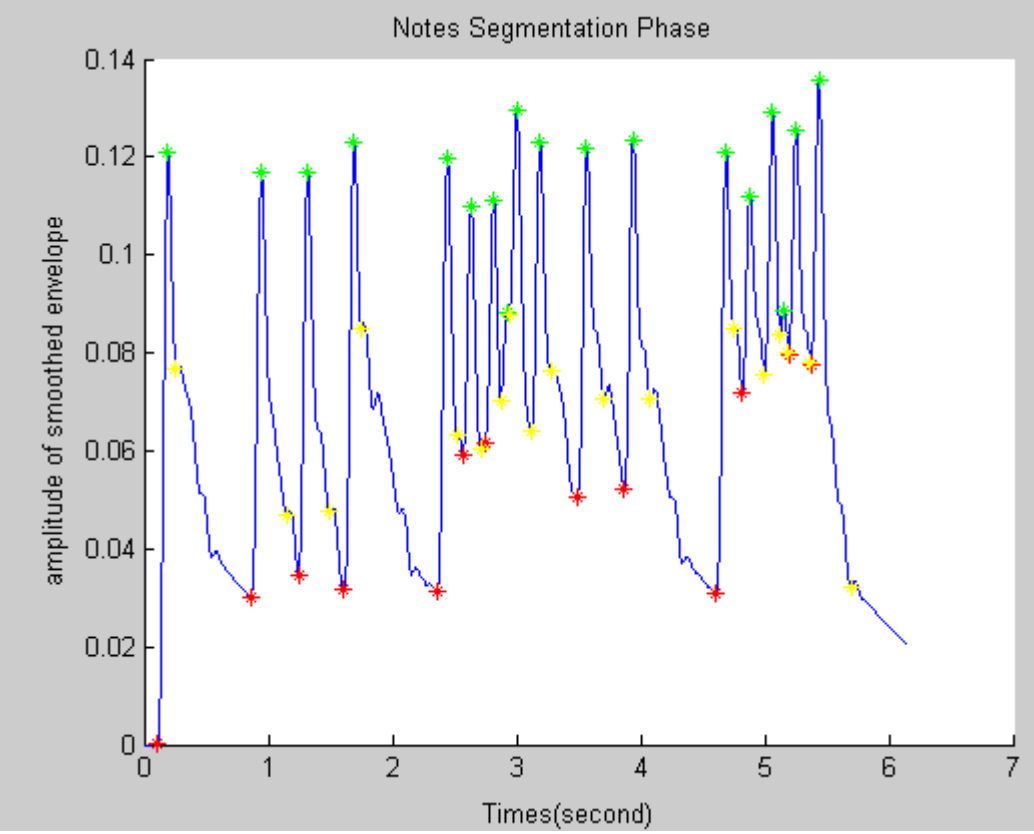

Figure 3.14: Notes segmented for the file *generic-data-modified.wav*.

In Figure 3.14, we see the segmented notes. Their onsets are colored in red. The local maxima are colored as green and the local minima are colored as yellow. Each set of three asteriks forms a note segmented. We can consider each three components as the note's starting, attack value, and finishing.

3.10 Spurious Attack Elimination

After note segmentation process, we have a number of notes detected. Among them, some are the real ones that we would like to keep but some are not. The nature of percussive music causes these types of undesirable attacks just before or after the major attacks that we want to detect [24, 26]. These attacks are significantly tinier both in durations and amplitudes when compared with the genuine and major attacks (see Figure 3.15).

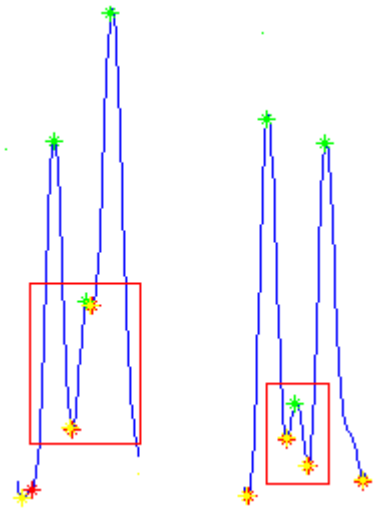

Figure 3.15: Two spurious attacks enclosed in red squares.

In Figure 3.15, two spurious attack fragments which were taken from Figure 3.14 are viewed and enclosed in red squares. The left one is an example for a spurious attack before a major attack. The right spurious attack is an example for an spurious attack just after the major attack.

By using this property, we can distinguish these spurious attacks from the set of detected notes in Section 3.9. These redundant notes must be eliminated in order to make a proper transcription [32]. Otherwise, accuracy of the method will be tremendously decreased [26]. Elimination method consists of three steps: Firstly, we compute the length of each attack. Then, we find a median of these lengths and calculate a threshold value by multiplying the median with a specific constant. Lastly, we get rid of the lengths which is smaller than the threshold value.

First of all, all attacks are shown in Figure 3.14 as data points whose compositions are a 3-tuple set. In the cartesian plane, each note has three values in both x and y axes. When we put the onset, local maximum, and local minimum indices' values inside the time array t, we get the corresponding time components for each note as a result. To obtain the y axis values, we use three index vectors $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ which are formed during the note segmentation process. When we put them as index inside the smoothed envelope array $SENV$, we get all y axis values for each tree components of all segmented notes. We can formulate the case as follows

$$OnsetVal = SENV(\boldsymbol{u}), MaxVal = SENV(\boldsymbol{v}) \text{ and } MinVal = SENV(\boldsymbol{w}) \quad (3.18)$$

respectively. Two of these y axis values are used in the calculation of an attack length. For all the segmented notes, we compute an attack length by substracting the onset value from local maximum value. As a result, the substraction yields a scalar value l_i. We can formulate the substraction as follows

$$l_i = MaxVal_i - OnsetVal_i \quad (3.19)$$

where i denotes the corresponding note. Figure 3.16 summarizes a generic step of attack length computation.

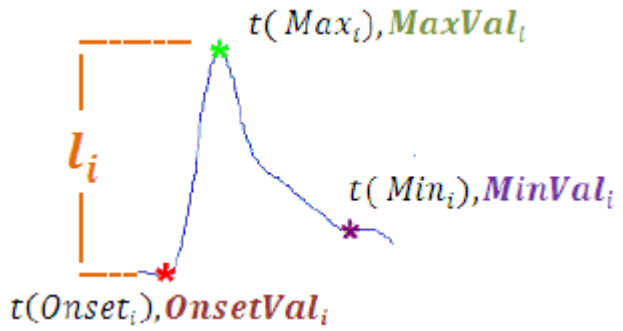

Figure 3.16: Note components and the length of attack $\boldsymbol{i}$.

In Figure 3.16, onset, local maximum, local minimum values and their corresponding time components for a note are seen. The length of a generic attack is also provided.

After obtaining all attack lengths, we would like to calculate the threshold h. Firstly, we find the median of attack lengths. We denote it as m. Then, we multiply this value with a specific constant c in order to produce the elimination threshold. We can formulate the case as follows

$$h = mc \tag{3.20}$$

The value of the coefficient is so vital for both the achievement of elimination and the accuracy of entire transcription process. We determine its value according to the results of the first experiment explained in Section 5.14 (see Table 5.2 and Figure 5.15).

Therefore, we finish the preparation steps for the core elimination phase. All we only do next is to check whether an attack length is greater than the threshold h or not. If that condition is satisfied, we add the currently testing attack length's index to a new set of non-spurious attacks' indices. We denote this newly formed set as $NewInd$ which is used to update the content of the arrays $Onset, OnsetVal, Max, MaxVal, Min$ and $MinVal$. Hence, this update operation purges the spurious attacks' indices from these arrays by only inserting the clean ones this time. Consequently, we prevent detecting the spurious notes. After elimination, the clean case of all segmented notes is shown in Figure 3.17.

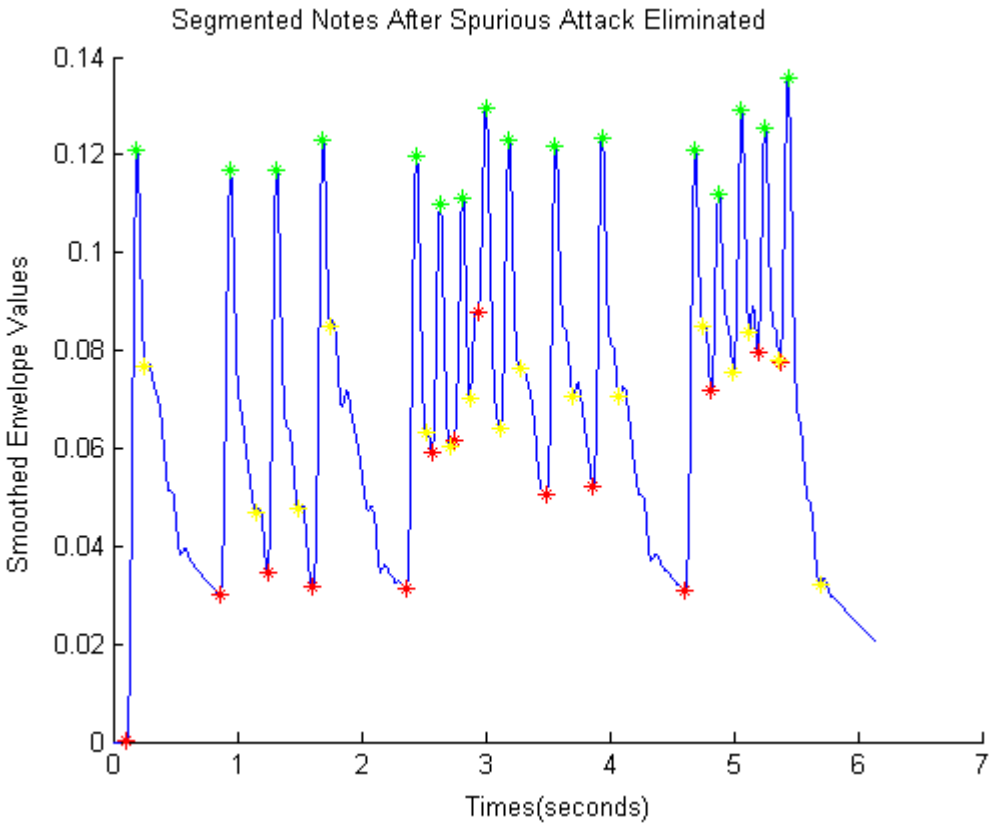

Figure 3.17: All remaining notes seen after spurious attack elimination.

In Figure 3.17, all remaining notes are shown after spurious attack elimination. The red asterisks denote the note onsets. The green asteriks denote the time-of-max of every attack. Lastly, the yellow asteriks represent the end of the note.

After the elimination process is completed, the difference between Figure 3.14 and Figure 3.17 is apparent. Some spurious attacks which are relatively shorter in lengths and smaller in amplitudes are eliminated. When two figures are compared, it can be easily noticed that some local maxima values are absent in the second one, especially around the envelope value 0.08. The segmented notes are ready for the pitch detection phase.

3.11 Pitch Detection between Successive Note Onsets

In Section 1.2.4.1, we explain that pitch describes the fundamental frequency of a music note. This fundamental frequency serves to place the musical note in the musical staff as mentioned in 1.2.4.2**.** Staff representation also indicates the note's place as a function of time [10]. Moreover, note pitches of piano instrument are well-

known for a number of octaves [22, 33]. By estimating the pitch of the note, we aim to detect the note name from where it falls to an appropriate octave location. Pitch detection proceeds in three steps: Firstly, we prepare the frames of the signal s between all successive note onsets. Then, we obtain the frequency components for each frame by taking $n-$point FFT. We pick the most powerful frequency component as a candidate pitch. Lastly, we look-up the most nearest piano pitch for this candidate. From its place in the look-up table, we can reach the name of the note we try to detect.

To be able to detect note pitches, we assume that there should probably exist a leading frequency component between the interval of two successive note onsets. In fact, this interval constitutes a frame of the signal which is denoted as new_s_i. Every such frame new_s_i has a different size because the starting and ending positions vary according to the currently employed successive onsets. In addition, we make another assumption is that the instant that the note occurs can be estimated by calculating the median of the simultaneously evaluating frame in terms of time. For that reason, a time frame new_t_i which corresponds to new_s_i is required to approximate the pitch time. In birief, the estimated pitch time is the median of new_t_i. Figure 3.18 shows the successive frames for the segmented second, third and fourth notes of Figure 3.17.

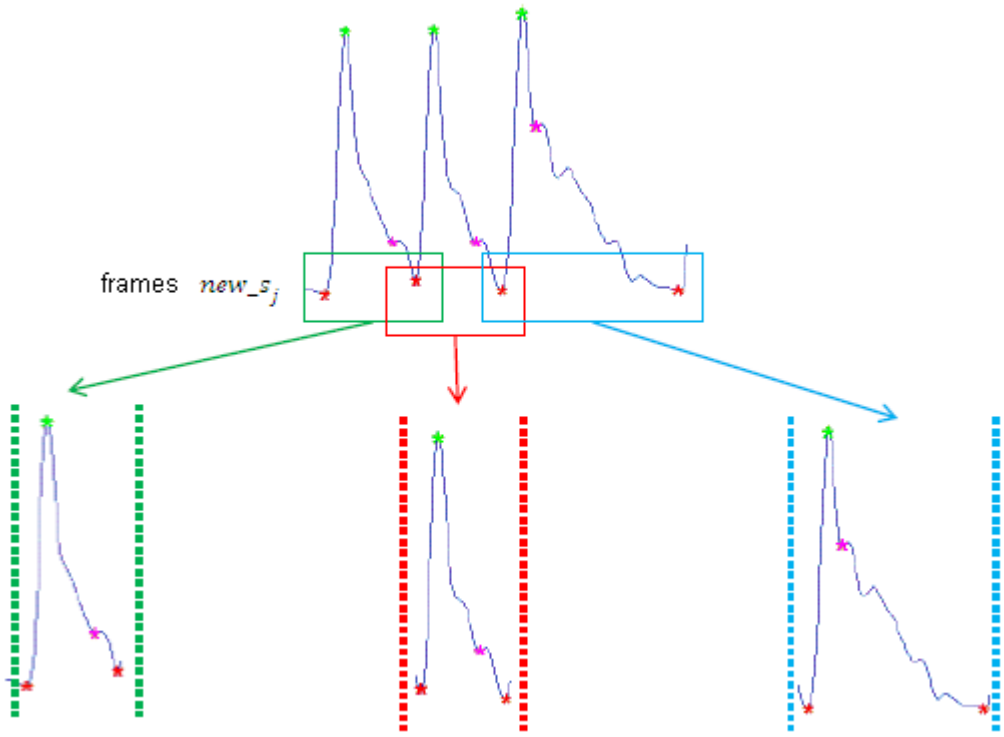

Figure 3.18: Three successive frames. We try to find a pitch value for each.

Frequency components of each frame are calculated via the Fast Fourier Transform. This operation requires two inputs: a signal frame new_s_i and a number $NFFT$ which indicates the length of the transform. Because every frame may have different frame length, $NFFT$ is computed in every iteration according to the frame size. In order to run FFT algorithm efficiently, it is a wise choice to select a $NFFT$ value which is power of two [33]. Hence, we can formulate the computation of $NFFT$ as follows

$$NFFT = 2^{nextpow2(|new_{s_i}|)} \tag{3.21}$$

where $|new\_s_i|$ is the length of the ith frame and $nextpow2$ is the operator which returns the smallest exponent that makes the value of $NFFT$ greater than or equal to the length of the frame. As a result of the transform, we obtain the frequency components of the signal frame new_s_i. We denote the transform as Y which is expressed as a function of frequencies.

$$Y = fft(new_{s_i}, NFFT) \tag{3.22}$$

The power spectrum of the signal frame new_s_i is the elements of array $|Y|^2/NFFT$ [34]. It is well known that the square of the signal represents its power. We measure the energy distribution at various frequencies. We mention that a frequency leads for each frame. Its power should be ultimately greater than other frequencies. Therefore, the pitch candidate that we seek is the most powerful frequency of the frame (see Figure 3.19). The range of frequencies varies from 0 to the half of the human audible threshold [25]. The values greater than this range may well be spare because we utilize nine octaves with frequencies of 8 kHz at maximum [35].

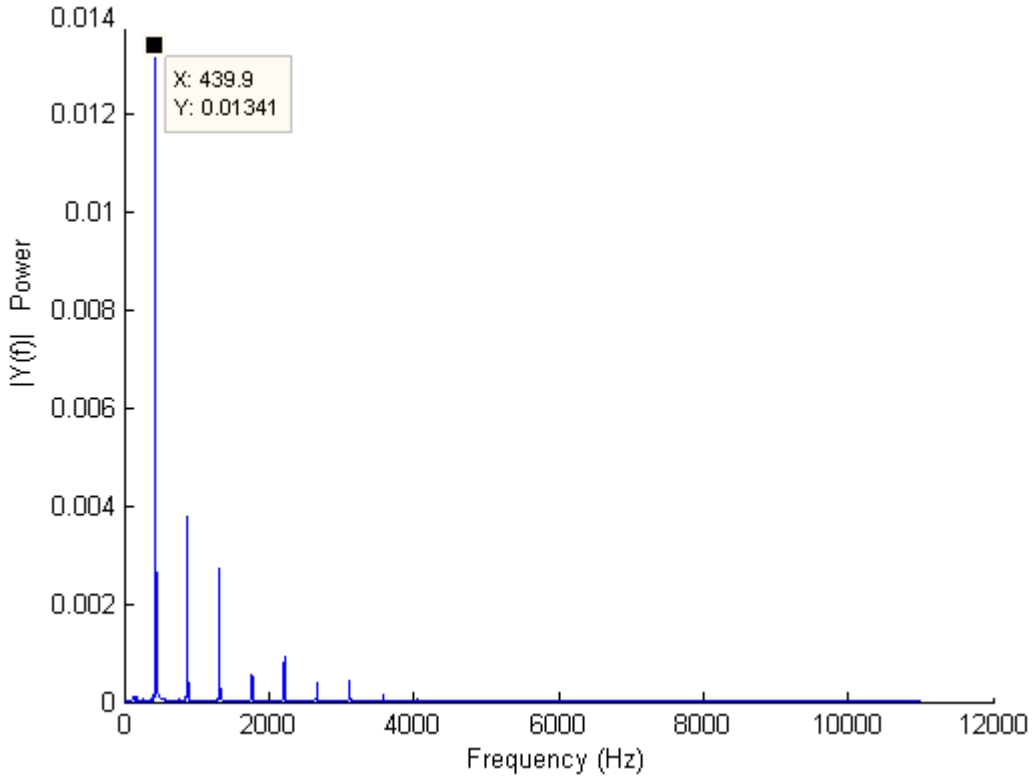

Figure 3.19: The most powerful frequency and its strength, holding for the musical note A from the middle octave.

In Figure 3.19, the value of the strongest frequency f' is 439,9 Hz. Mathematically, we can express the case as follows

$$\max(Y) = Y(f') \tag{3.23}$$

where $Y(f')$ is equal to 0,01341. Other frequencies are relatively weak in comparison with f'. Thus, we find the measured frequency of the second segmented

note in Figure 3.17. In addition, it exactly matches with the first frame's pitch of Figure 3.18.

After obtaining the most powerful frequency f', we seek the nearest value inside the look-up table of $Octaves$. The look-up table frequency ltf will be the pitch value ($Pitch_j$) that we would like to detect. The row and column information of ltf stands for two different concepts: Its column index points to the octave number i.e. the note middle A resides in the fourth octave. In addition, the row index corresponds to the note number inside the look-up table $Notes$. With this number, we obtain the name of the note ($NName_j$) from $Notes$ i.e. the note middle A is located inside the tenth row of $Octaves$. Because there are twelve semitones in the western music, the tenth one refers to an A note when sorted from the note C to the note B. To conclude, we obtain the pitch values and note names of segmented notes. In addition, we store them in $Pitch$ and $NName$ arrays, respectively. For instance, the three successive frames in Figure 3.18 get the pitch values as 440Hz, 493.88Hz, and 293.66Hz and the note names as A, B and D, respectively. By using the pitch time information, we can demonstrate all detected pitches as a function of time for all segmented notes (see Figure 3.20).

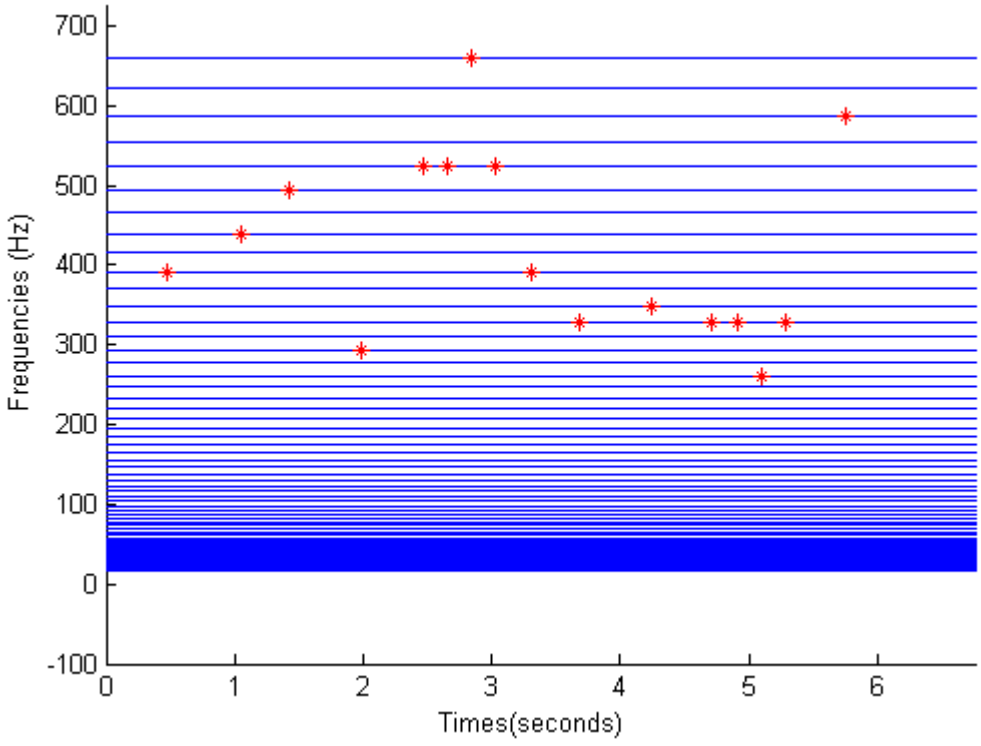

Figure 3.20: Pitch values drawn according to their pitch times.

There may be some time shift while representing the note pitches because we take the median values of each time frame as our assumption illustrates. This approach does not cause any severe problem because all notes' pitches are shifted a bit relatively to their genuine locations in the time axis.

3.12 Note Durations Calculation

In order to find note value types, we need note durations. Because we already know the note onsets, we can estimate all note durations from their corresponding time values. From this point of view, a note duration can be described as the amount of time passed from one onset to its right sibling. We assume that the successive onsets can be used as note offsets interchangeably because we only deal with the musical notes as music events. For example, there is no rest between the notes in our data.

For all onsets, we first obtain the matching time values. Then, we substract the current onset time from the right sibling's one except the last onset time which is substracted from the total playing time of the data. This is because the last onset does not have any right sibling onset. We can generalize the case with the formula as follows

$$NDur(j) = t\left[Onset_{j+1}\right] - t[Onset_j] \tag{3.24}$$

where $NDur(j)$ is a note duration computed for the jth note and $t[Onset_j]$ is the instant when the jth note onset occurs. Figure 3.21 summarizes the whole process.

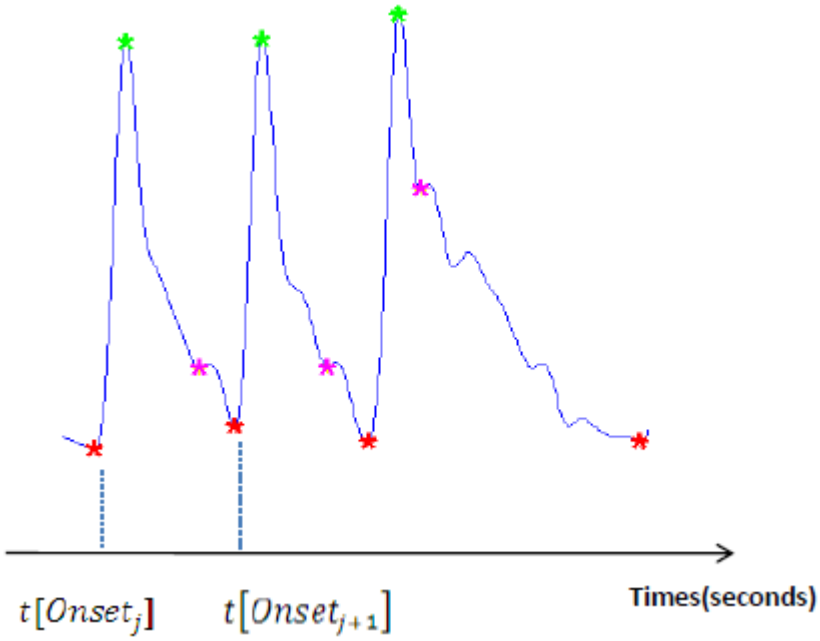

Figure 3.21: Note duration calculation made for all successive note onsets.

3.13 Note Value Types Detection

In Section 1.2.4.3, we introduced five different note value types that are unknown in our test data and we would like to detect and offer a value type for each note we segment. We assume an algorithm to solve this problem. The task is constituted from two rounds. In the first round, we try to classify the note durations estimated in Section 3.12 according to the five known classes. As intended, they are all five rhythmic note value types mentioned in Section 1.2.4.3. We denote the number of distinct note value types as $disType$. We try to find groups of note durations which are gathered very close to each other. For each distinct group, we calculate an arithmetic mean that symbolizes the average note duration. In the second round, we assign each group's average to all note durations falling into that group.

At the beginning of the first round, we have note durations and distinct types of note values known. First of all, we find the longest note duration lnd. We collect all note durations standing in the range of peripheral boundaries. They are scalar values to limit a group's circumference. There exist one left ($leftB$) and one right ($rightB$) boundaries at total. In this case, the right boundary is lnd itself and the left boundary is calculated by the multiplication of lnd and a constant denoted as $periPer$. This

constant describes a percentage. Then, we add all note durations that are greater than or equal to $leftB$ and smaller than or equal to $rightB$ to the current group whose name is 'notes-around' ($NAround$). We call as the centroid (C_q) for the arithmetic mean of this group. Next, we compute a pseudo centroid ($tempC$) of the next group in order to calculate this group's centroid at the next iteration. For this purpose, we only divide C_q by two. This is the special step of this algorithm. In a generic step, we compute the peripheral boundaries according to $tempC$. In this case, the left boundary is calculated by the multiplication of $tempC$ and a constant denoted as $periPer$ and the right boundary is calculated by the multiplication of $tempC$ and the result of $(1 - periPer)$. To cope with the risk of the failure of the boundary condition checking, we assign $tempC$ as the centroid value because we do not have any suitable note durations to calculate a mean. This round finishes by obtaining all centroid values (see Figure 3.22).

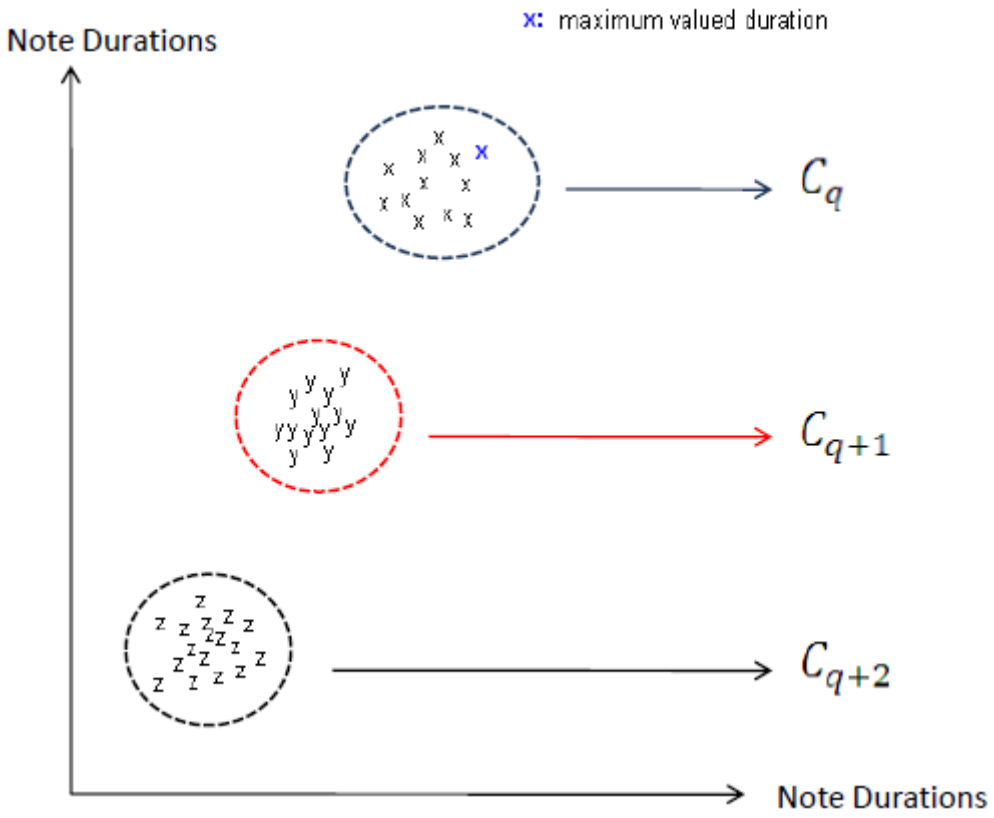

Figure 3.22: Mechanism of centroid calculation and its representaion in the plane.

In the second round, we map the centroid values to all given note durations. For this sake, we find the most nearest centroid value C_k for any note duration $NDur(j)$ and

then we assign C_k to $NDur(j)$. After this round completed, we have the same note duration for all data points of the same group. The mapped note durations are denoted with $MapDur$. Obligatorily, all members of a rhythmic note value type must have the same note durations as the music rules imply. In the next phases, we use these generalized note durations. Figure 3.23 summarizes the result of the second round. See Appendix B.1 and B.2 for the pseudocodes of both two rounds.

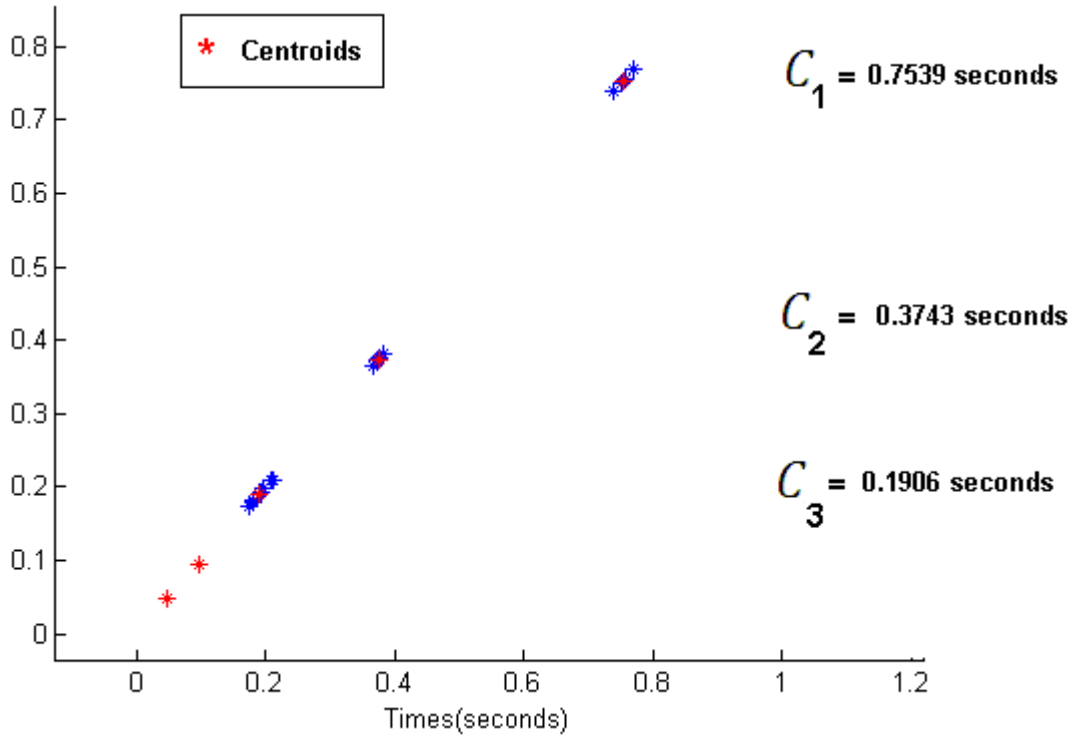

Figure 3.23: The centroids and their surrounding data points seen.

In Figure 3.23, we encounter three calculated centroids and two pseudo centroids which does not have any data points at their surroundings. When zoomed, the note durations residing in an environment of a centroid, usually differ from each other. The underlaying reason for this deviation is that we calculate note durations from the temporal distances of two successive onsets. This distance can be varied (very close or just close) for a group of durations because the onset position is dependent on the percussive music waveform. We do not expect that abrupt changes in the waveform have regularities for a data file completely created in a random fashion in terms of note value types.

3.14 Note Labels Assignment

To complete the transcription, we need to give labels to the segmented notes. In the music theory, rhythmic note value types are described as even numbers from two to sixty four except the whole note [10]. In Section 1.2.4.3, we explain the relationships between rhythmic notes. Whole note gets the number one, half note gets the number two, quarter note gets the number four, eighth note gets the number 8 and so on. We provide another algorithm which combines the note names with the labeling numbers mentioned above in order to provide fully qualified note names such as $G1, A\#2, B8, D4, C16$, etc... This matching operation is made via the mapped centroid durations. We assume that largest centroid should get the label of whole note. In the same manner, second largest centroid should be matched with the label of half note. In fact, we match the labels of five different note value types to the provided centroids durations in ascending order. In addition, the centroids are automatically sorted due to the nature of the detection method (see Section 3.13). We assume that this algorithm works appropriately when the randomly created data content is constituted by all five value types. A counter example can explain easily how procedure works. If we have a set of notes whose value types are quarter, eighth and sixteenth, the procedure combines the labeling numbers one, two and four with the corresponding note names of this set. In this case, the algorithm assigns these labeling numbers to this set because the decision is made according to the provided centroid values. Figure 3.23 is an example of this situation. See Appendix C for the pseudocode of the note label assignment algorithm.

Chapter 4

Proposed Transcription Algorithm

In this section, we propose an algorithm which is used for detecting notes of a musical signal. We have explained the components in the third chapter. The algorithm's intermediate values are fully dependent to the previous outcomes since they are required to be inputs in the next components executions. So, we can define this situation as a linear dependency of algorithm's contents. In software architecture literature, this structure is named as Pipe Line or just Pipes [36]. Here we have the pseudocode for the algorithm:

PROPOSED-TRANSCRIPTION-ALGORITHM($s[n]$, $t[n]$, f_S, $Notes$, $Octaves$)

```
    # 1. Construction of amplitude envelope of signal
    [Env, IEnv] ← constructMagnitudeEnvelope (s[n], ws, ja)

    # 2. Smoothing the envelope waveform
    SEnv ← smoothEnvelope(Env, f_S)

    # 3. Run slope detector
    S{n} ← runNovelSlopeDetector(SEnv, t[IEnv], k)

    # 4. Launch 'Rise-Times' Model
    Blocks ← runModel(S{n}, modelThreshold)

    # 5. Progress the model
    Blocks ← progressModel(S{n}, Blocks, meanThreshold)

    # 6. Run Hinkley's CUSUM algorithm
    U_M ← runCusum(S{n}, detectorType)

    # 7. Segment the notes
    [Onset, OnsetVal, Max, MaxVal, Min, MinVal] ←
                segmentAllNotes(SEnv, S{n}, IEnv, U_M, Blocks, H, ϵ)

    # 8. Spurious attacks elimination
    NewInd ← clearSpuriousAttacks(OnsetVal, MaxVal, c)

    # 9. Reassignment of new indices
    Onset ← Onset(NewInd)   and   OnsetVal ← OnsetVal(NewInd)
    Max ← Max(NewInd)       and   MaxVal ← MaxVal(NewInd)
    Min ← Min(NewInd)       and   MinVal ← MinVal(NewInd)
```

Figure 4.1: Pseudocode of proposed transcription algorithm (continued).

```
30      # 10. Pitch detection between successive note onsets
31      [Pitch, NName] ← detectPitch(Notes, Octaves, Onset, t[n], s[n], f_S)
32
33      # 11. Note durations calculation
34      NDur ← calculateNoteDurations (t[n], Onset)
35
36      # 12. Note Value Types Detection
37      [C, MapDur] ← findNoteValueTypes(NDur, periPer, disType)
38
39      # 13. Notes Labels Assignment
40      NoteLabels ← assignNoteLabels(MapDur, NName)
```

Figure 4.1: Pseudocode of proposed transcription algorithm (continued).

In Figure 4.1, we present our proposed transcription algorithm. It can be considered as a final summary of all we talked about in Chapter 3. The input arguments of the algorithm are the audio signal ($s[n]$), its corresponding time vector ($t[n]$), the sampling frequency (f_S), the look-up tables $Notes$ and $Octaves$, respectively. By using these arguments, our aim is to transcribe fully qualified note names at the end as it is stated as $NoteLabels$ in line 40 of the figure. There are 13 phases to achieve the transcription task.

In the first phase, we construct an amplitude envelope of the audio signal ($s[n]$) because the percussive music pieces that we work on allow to segment note onset easily (line 02). This situation facilitates the entire segmentation process (see Section 3.4). In order to obtain the envelope, we used a windowing system that results in picking the maximum valued signal element from the window. Every window has a size denoted by ws. Windows slide through the signal according to a jump amount (ja). At the end of this phase, we obtain an envelope array Env and its corresponding index array $IEnv$.

As it is seen in Figure 3.5, the envelope is not suitable for note segmentation. We apply a Butterworth IIR lowpass filter to the envelope (Env). As a result, we obtain the smoothed envelope ($SEnv$) (see Figure 3.7). Beside the usual parameters of lowpass filter (see Section 3.5), we also used sampling frequency (f_S) as a parameter (line 05).

In the third phase, we built overlapping lines over the waveform of smoothed envelope ($SEnv$) by employing the linear regression (line 08). We utilize k approximation points of $SEnv$ each time in order to get a line from the regression. Each line's slope is stored as an element of the slope array $S\{n\}$. Abrupt changes in the slope can be considered as an onset of a musical note.

To select significant slope values, we employ the 'Rise-Times' Model. This model eliminates the slope values below a certain threshold value ($modelThreshold$) (line 11). We expect to detect a note event from slope changes (see Figure 3.10). Such a slope change first makes a sudden rise in magnitude and then reaches a peak point. From that point, it begins to decay below the threshold value. Every such slope motion is called "block". We collect the starting and ending indices of all blocks into the first and second columns of the matrix of $Blocks$, as an output.

With respect to the nature of percussive onsets, the slope blocks correspond to the attacks of the smoothed envelope (see Figure 3.11). They are placed between rise-slope and time-of-max. However, not all attacks can be note events. Some of the attacks have significantly smaller amplitudes. We can process the model by eliminating the blocks whose arithmetic means are less than the mean threshold value ($meanThreshold$). Thus, some blocks are eliminated from the matrix $Blocks$ which is returned at the end (line 14).

In the sixth phase, Hinkley's CUSUM calculates the cumulative sums (U_M) of the slopes (line 17). Because we look for sudden changes or increases in slope, the detector type ($detectorType$) must be assigned to 'increase'.

In note segmentation phase, our target is to retrive the note event as a union of three components: physical note onsets ($Onset$), time-of-max or in other words local maxima (Max), and local minima (Min). These are all index arrays. Beside these, there are also corresponding smoothed envelope ($SEnv$) values for all of them: $OnsetVal$, $MaxVal$, and $MinVal$. The last three are representative of y-ordinate in the plane while the former ones represent x-axis values (see Figure 3.16). For each block of $Blocks$, we segment a single note (line 20 and 21). First, we obtain note

onset by applying the Page-Hinkley stopping rule over the data of cumulative sums (U_M). A threshold (H) is employed to determine the occurence of an abrupt change. Moreover, to get the local maxima and minima values, we try to detect where the absolute value of slope ($S\{n\}$) falls below (ϵ) or ($-\epsilon$) (3.14 and 3.16).

During the execution of the previous phase, some unrelated attacks are also detected as music events. We have to eliminate these spurious attacks (line 24). We compute the length of every attack by substracting onset value from time-of-max value (3.19). We discard the attacks whose lengths are lower than the elimination threshold (h). This threshold is calculated from the multiplication (3.20) of the median of all attacks with the given coefficient (c). Exclusion includes all three components of a segmented note. Finally, we return a set of remaining attacks as $NewInd$.

In the ninth phase, an obligatory update operation is made in order to reflect the absence of eliminated attacks. From line 27 to 29 of the figure, six arrays are affected since they are redefined with the new set of attacks ($NewInd$). Therefore, elimination is accomplished in system wide. Accuracy of the subsequent phases strongly depends on this phase. Otherwise, it can decrease drastically.

For every segmented note, we aim to get a pair of pitch and note name ($Pitch, NName$) in the pitch detection phase (line 31). This operation is proceeded between the successive physical onsets ($Onset$). For given time vector ($t[n]$), signal ($s[n]$), and sampling frequency (f_S), we split two frames of first two parameters each time we investigate successive note onsets. These two frames are used in a FFT to find the main frequency component. By comparing to a look-up table of frequencies ($Octaves$), we learn the closest equivalent piano pitch of our estimated frequency. From the column number where we get the pitch value, we retrieve the note name by inserting it as an index inside the look-up table of note names ($Notes$).

In order to calculate note durations, we first obtain the temporal equivalent of the physical onsets ($Onset$) by using the time vector $t[n]$. Next, we substract every successive onsets' temporal equivalents from each other. Every temporal difference

we get signifies an estimated note duration. We store them in a note duration array $NDur$ (line 34).

In the twelfth phase, we consider that estimated note durations ($NDur$) are distributed according to a clustered and hierarchical order (line 37). We illustrate this distribution as in Figure 3.23. Every cluster corresponds to a generalized note duration. Our aim is to find these generalized note durations. In addition, we have a constraint that the number of note duration clusters ($disType$) that an audio file can possibly yield is fixed and known i.e: 5. We use a two-round algorithm to estimate clustered note durations (Section 3.13). At the first round, we match data points to their related clusters which are formed according to the calculation of scalar boundaries for a given peripheral percentage amount ($periPer$). Every cluster's mean value gives the centroid of this cluster. Distinct note type values refer to the centroid values (C). At the second round, every note duration is associated with its distinct note value type. As an output, a mapped note duration array ($MapDur$) is obtained.

In the final phase, we try to estimate the fully qualified note labels in conjunction with the rhythmical representation of mapped note durations ($MapDur$) and note names ($NName$). For the mapped duration of every note, we infer a label number from the ordered list of labels ("1, 2, 4, 8, 16"). The highest and lowest marks are "1" and "16", respectively. Lastly, this mark is appended to a note name. The result is a fully qualified note label like $C\#8$ (line 40). All note labels are stored in the array $NoteLabels$ which is the only output of the entire algorithm.

Chapter 5
Results

5.1 Data

In our experiments, we have used a data of wave file each time. I used a software which was completely written by me and serves to generate these wave files as it was mentioned in Section 3.1 . Each file consists of 50 randomly generated notes. These randomly generated notes may be one of the notes coming from the octave where middle C(C4) is located as the beginning note and C5 is the finishing note. We can say that there are possibly 12 notes when generating a note. The midi note numbers' equivalents are the interval of integers from 60 to 72 for this octave. Moreover, we have another constraint that these randomly generated notes may have five different note values types which were pre-set as whole, half, quarter, eighth, and sixteenth. This setting is configurable in our data creator program.

A wave file created under these circumstances i.e. test-5-50-notes.wav plays for 33 seconds. It is prepared by using Pulse Code Modulation and was sampled with 44.100 Hz where each sample is encoded with 16 bits.

Every data file like *test-5-50-notes.wav* comes with an another text file which gives the music scores as mentioned in Section 3.1. Here we have *test-5-50-notes.txt* for the test file provided above (see Figure 5.1).

```
scores: 50 E2 G16 F16 D4 F8 G8 E4 D#2 C#16 E16 F4 E4 G4 F8
D#8 C2 F2 D2 E16 G#8 C16 A4 D16 G4 F#4 A16 G4 C8 F#1 C#8
D16 G16 D16 D4 F4 F8 E16 D8 D2 F16 A16 D16 E16 C8 A2 G4
D#16 G#8 F#16 F1
```

Figure 5.1: The content of the file *test-5-50-notes.txt*.

In Figure 5.1, there are 50 music scores preceded by a starting label "*scores:*". This label is followed by an integer number to indicate the number of notes inserted in this file. Then, fifty fully qualified note names are appeared.

A fragment of randomly generated notes inside this mentioned file above is exhibited in Figure 5.2 as following:

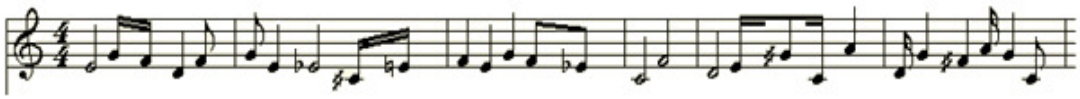

Figure 5.2: 28 notes out of 50 placed in the music staff.

5.2 Envelope

As stated in Section 3.2 and its sub sections, we retrieved the signal and its related accompanying set of information e.g t[n]. Now, the signal retrieved is just a column vector whose size is [730384x1]. This size holds for the column vector t[n] which was prepared with the rationale explained in Section 3.2.1 and its size is equal to the size of the signal. We constructed an amplitude envelope for this signal with a window size of 64 and a jump amount of 16. These are specific scalar values and can be tuned according to the subjected data. In addition, we determine these parameters' values in the light of the resuls of the second and third experiments that we conducted (see Table from 5.3 to 5.6 and Figure from 5.17 to 5.23). Detailed information about the reason of the selection of the assigned values i.e: ws=64 is provided in Section 5.14. As a result, we obtained an array Env whose size is [1x45645] (see the Figure 5.3).

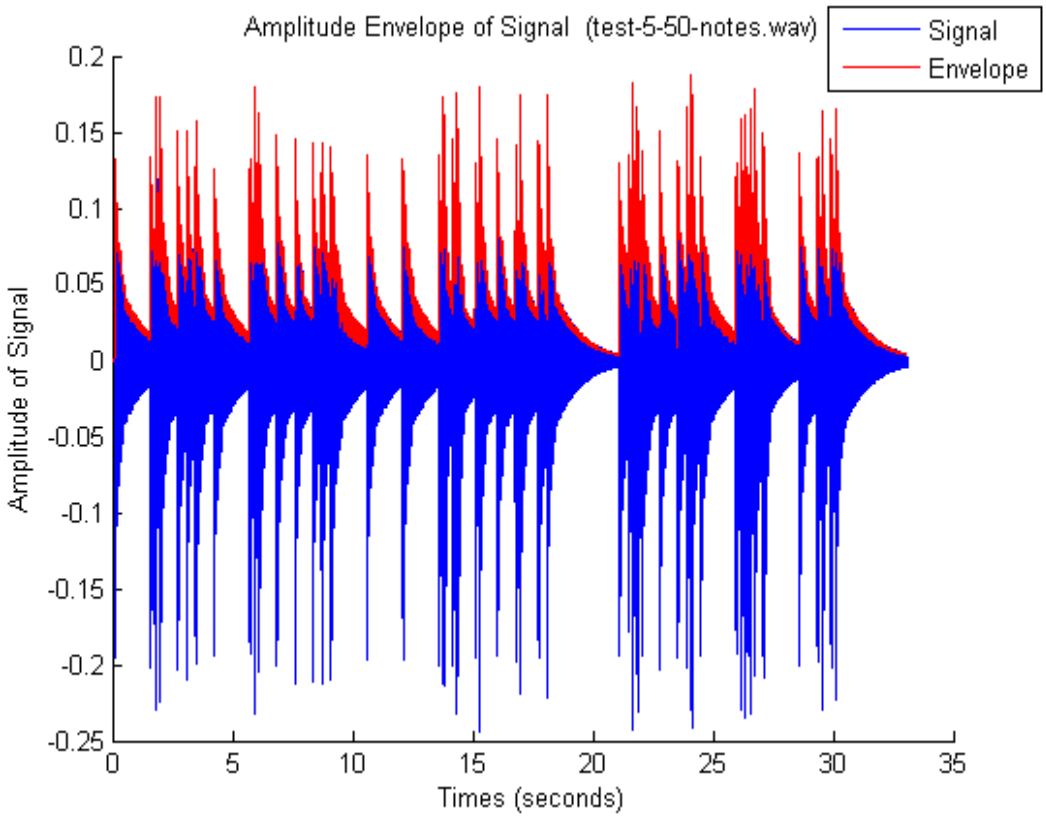

Figure 5.3: The amplitude envelope drawn onto the original waveform.

5.3 Smoothing

For the sake of good note segmentation, we have smoothed the envelope data by using a low-pass filter which was designed as a Butterworth filter whose passband frequency is 150Hz, stopband frequency is 450Hz, stopband attenuation is 60, and passband ripple is 1. As a result, we obtained an array $SEnv$ whose size is equal to the one of the array Env as intended. As a result, we have a better waveform without such a very quick and instantaneous vibrations. We benefit from its reduced waveform in the next step. To ease the understanding, see Figure 5.4 and Figure 5.5. When we compare them, Figure 5.4 carries lot of information which may be redundant, affects the algorithm performance and must be further smoothed before the filter was applied. However, the filter smooths the envelope and then, we deal with less amount of information which is more useful for the sake of implementation.

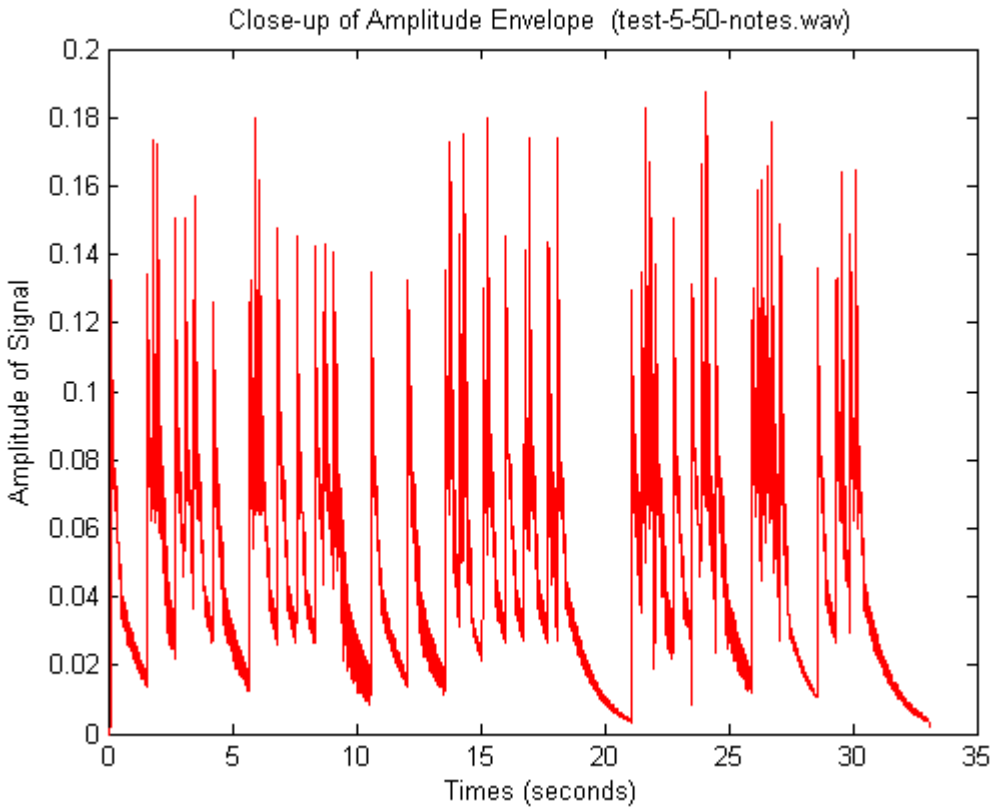

Figure 5.4: Amplitude envelope before smoothing.

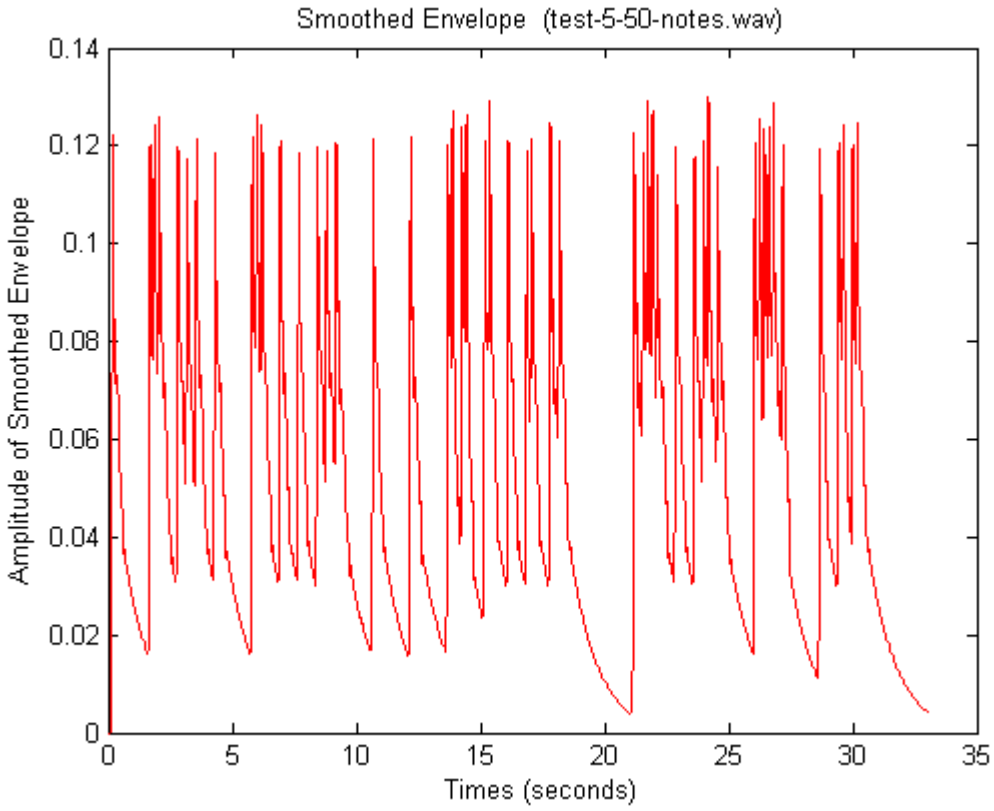

Figure 5.5: Amplitude envelope after smoothing.

5.4 Slope Detection

The smoothed version of amplitude envelope is used in slope detection by calculating linear regression for a given small data set of k approximation points. We have discussed the internal organization of this method in Section 3.6 . By using eight approximation points each time for one step of the detection, we get an array Slopes filled with slopes of each line created during the process. The size of this array is equal to the size of the smoothed envelope minus eight because. Actually, when the number of approximation points is eight, from the last eighth data point of the smoothed envelope, there did note remain any more eight points to approximate and never will be present because the data is finished. So, the last slope value belongs to the line which was derived from the eight last points. See Figure 5.6 and its zoomed close-up version of Figure 5.7.

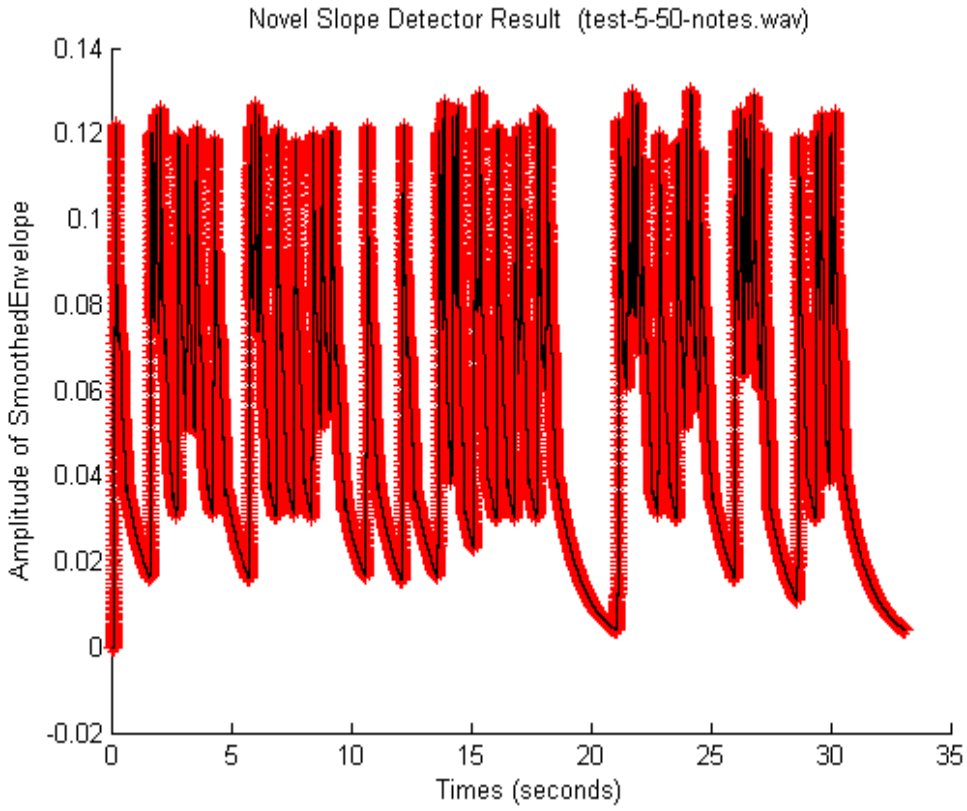

Figure 5.6: Slope detection result for the file *test-5-50-notes.wav*.

In Figure 5.6, the black lines created by the slope detector's execution are seen. They perfectly fit the waveform of $SENV$. The red asteriks are data points of $SENV$.

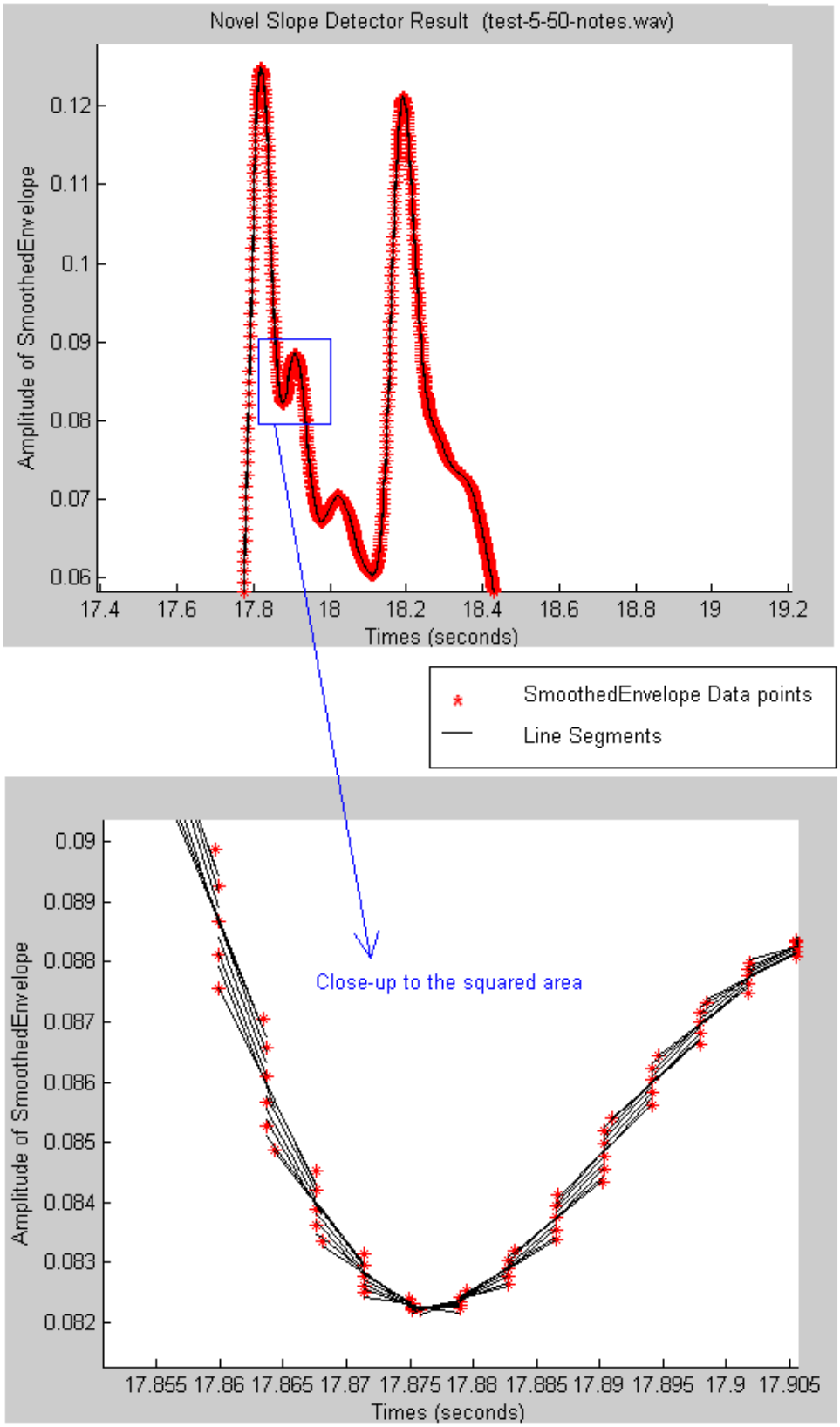

Figure 5.7: Close-up of the slope detection result.

In Figure 5.7, a close-up of Figure 5.6 is viewed. The top part of the figure shows an excerpt of two peaks from the data of $SENV$ in a time interval between 17th second

and 18th second. At the bottom part, it is seen the zoomed version inside the squared area. The lines computed during slope detection are drawn and fit very well to the data. Data point are shown in red asteriks.

5.5 'Rise-Time' Model

With the slopes calculated, we can now concentrate on the attack rise times in the data. One way is discussed in Section 3.7 that the slopes values being over a selected threshold can be considered as valuable material and we would like to focus such areas where we can pick some candidates to detect a music note. We configured the model threshold to the real value of 0,016. The slope values over this threshold are related to possible attack time of a note. When we launch this model, we get all attacks having slope values over the threshold. Some of them are related attaks and some of them, unfortunately not. We save every attacks' starting and ending locations as a function of time and indices. For test-5-50-notes.wav, by using this model, we get 145 attacks. We have a prior knowledge that data consists of 50 notes. There are lot of attacks to be eliminated. We can even make this elimination by making just a little trick. In Section 3.7.1, we discussed how we can improve our model. This makes a significant improvement that there remain only 55 attack blocks. This amount is very close to the real number of notes. We can show this elimination effect with the close-up views in a before-and-after manner (see Figure 5.8). Onto the waveform of smoothed envelope(drawn in blue), we draw the attack derived from the model in red. Actually, the attack represent the slope values corresponding to a musical event. This is true mostly for the related attacks. For instance, the slopes values perfectly reside on the percussive onset startings and get through along the transient and finally it reaches to the top of the peak which is a local maximum. This case is exhibited apparently on Figure 5.8. The slope values fit to the smoothed envelope perfectly while floating over its surface. After the elimination, we understand that the redundant attack components were disappeared from where they were black-squared, previously.

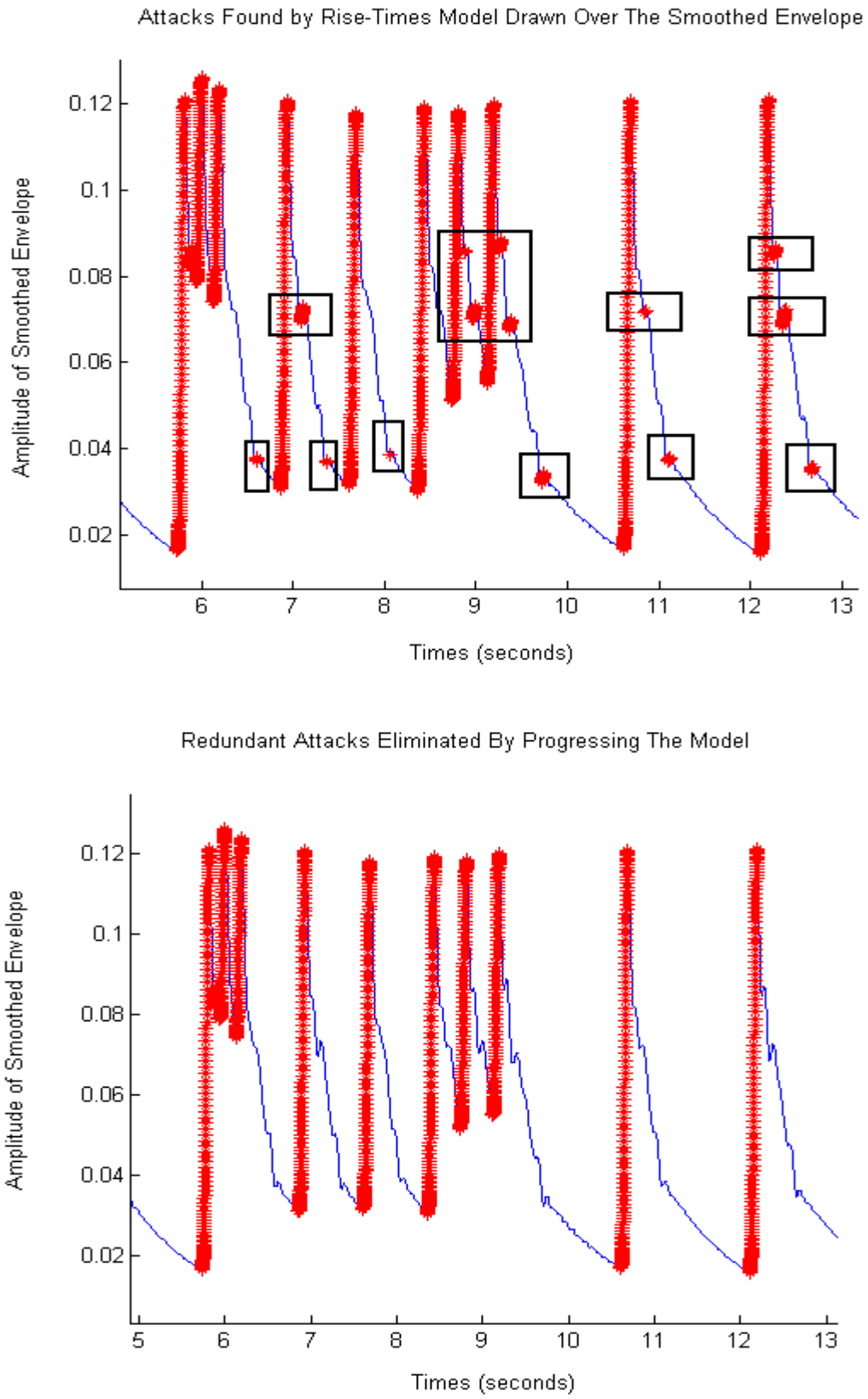

Figure 5.8: The 'Rise-Times' Model and its progression viewed.

In Figure 5.8, we see the application of the Rise-Times Model and its progression's result. The top sketch shows the redundant attacks found by the model enclosed in the black squares. By progressing the model, we leave these attacks and we get more cleanier and redundant attacks isolated model state. The top graph is before case and the bottom graph is after case. In both cases, a little excerpt of the complete data was zoomed for the convenience reason.

5.6 CUSUM Algorithm

To find out where the abrupt increase happens when a percussive music event occurs, we are in quest of getting cumulative sums of the slope array. The CUSUM algorithm was configured to detect sudden increase because the detector type is set to 'increase'. The reason of the selection of the detector type was explained in Section 3.8. As a result of the execution of this algorithm, we obtained cumulative sums of the slope array. The cumulative sums are stored in an array data structure with the size equal to the one of slope array. We will use this information while we segment the notes.

5.7 Note Segmentation

We call the segmentation procedure with all information obtained from the previous steps of our implementation as it was denoted in Section 3.9 earlier. In addition, we add two segmentation threshold variables which are used in calculating the 'end-of-attack' part of the 'Rise-Times' model and at the local minimum computation. These two variables which are named as threshold and epsilon may well be tuned according to the data tested.Their default values are 0,00001 for both. As a result, we obtained the values of physical onsets, the values of local maxima, and the values of local minima with their corresponding locations as a function of index. To ease of understanding, we show them in Figure 5.9.

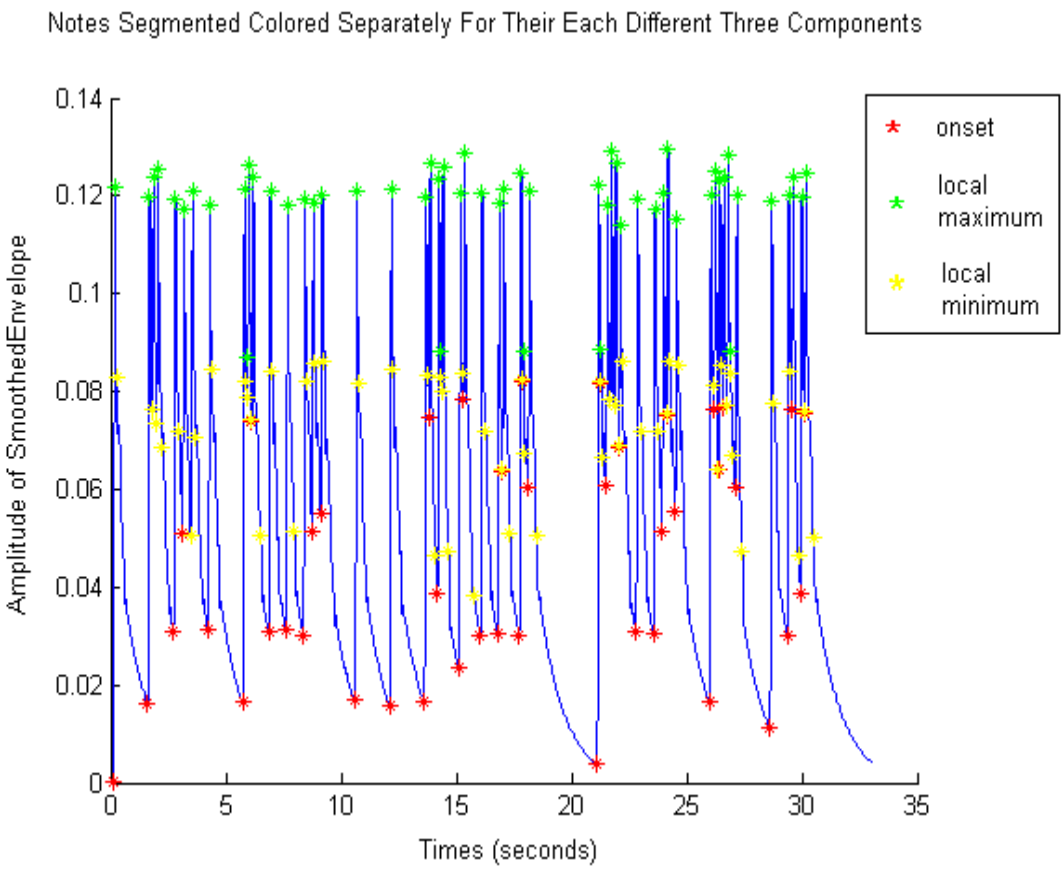

Figure 5.9: Segmented notes drawn with three temporal components.

In Figure 5.9, physical note onsets are drawn in red asteriks. Having being percussive onset, the note starting location in the time may well be designated with the pointing of onset. This is a legitimate statement also for the yellow asteriks which remark the finish of the notes. The green ones are the attack most mature part signifying the peak value. As a result, we got all these values for the previously detected 55 attacks and stored them in the three different representative arrays whose names are $OnsetVal$, $MaxVal$ and $MinVal$, respectively. In addition, we saved the locations of all these values in a function of time and indices, in three different arrays again as natural. All these six arrays size are [1x55]. This size alarms us about the excess attack value determined in a redundant fashion.

5.8 Spurious Attack Elimination

As stated in Section 5.10, the elimination of the spurious attacks maintains the accuracy of the implementation. We obligatorily apply this process because the nature of percussive onset is wide open to introduce kind of spurious attack with the genuine attack. When we run this procedure, we found out the last case of attacks totally purged from the spurious ones. Therefore, number of the candidate attacks to be assessed as a music note in a while was reduced to fifty. This is the number that we would like to get however it is too early to smile. Did we appropriately get rid of the right attacks from our attack set? We will learn this in a few steps further. Figure 5.10 shows the situation of segmented notes after the elimination spurious attacks for the test file test-5-50-notes.wav.

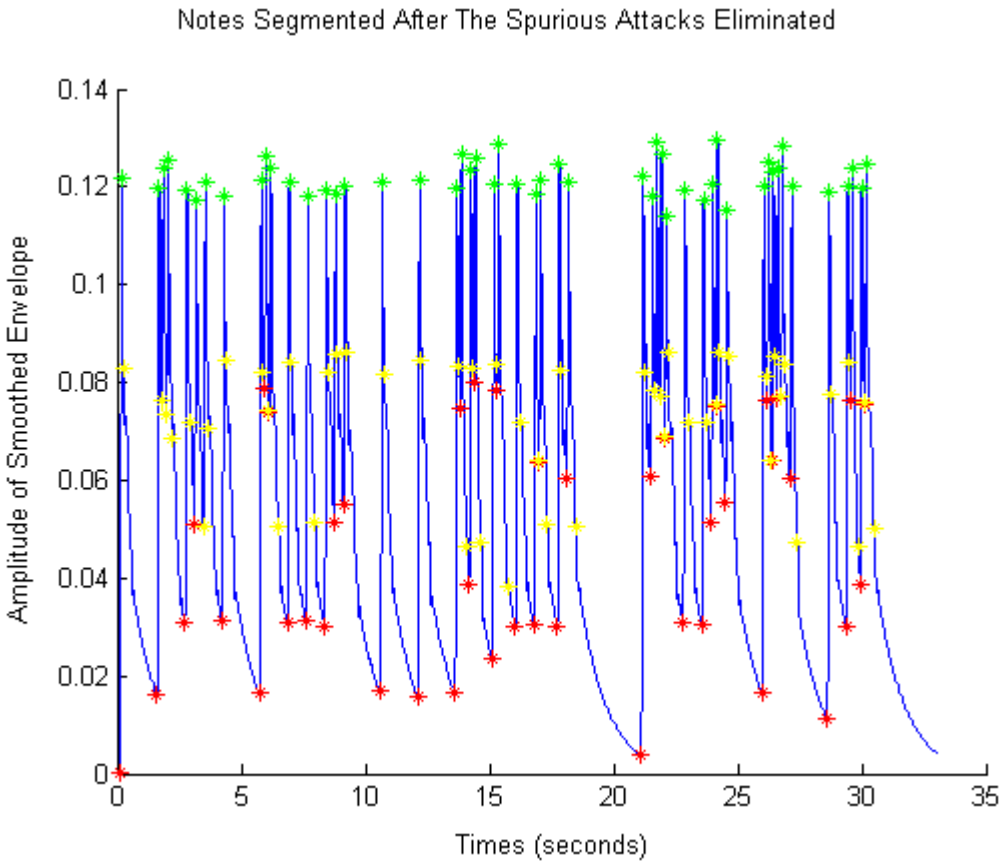

Figure 5.10: Segmented notes seen after the elimination.

Via the figure, we can identify easily which five attacks are eliminated. Pay attention to notice the difference between this figure and the previous one. If we focus on the

center of graph, we will notice that five green asteriks present in Figure 5.9, are now absent near the amplitude value of 0,08. This helps seeing that some of the real attacks were eliminated. Of course, other two component of these eliminated five attacks are also eliminated from the set of detected onsets and local minima.

5.9 Pitch Detection

At the moment, we have segmented fifty notes whose pitches are unknown. We will learn the note pitches by making a number of Fast Fourier Transform operations between the successive physical onsets remained after the spurious attack elimination phase by applying the criteria stated in Section 3.11. These pitches are the frequency values of the notes which are originated from the middle octave of grand piano instrument. As a result, we got pitches and corresponding names of notes for these pitches. We have found fifty pitches and fifty note names. We provide a figure for the display of the pitches found as a function of time in Figure 5.11. In the figure, every red asterisk actually shows a detected note and is placed into the cartesian plane according to its pitch value and its corresponding pitch time derived as mentioned in the section 3.11. We summarize the pitches and the note names that we found in Table 5.1's second and third columns.

Table 5.1 summarizes all the results obtained in various steps of implementations for fifty notes. The column names are very obvious. At the leftmost column, the abbreviation of number is used and signifies the note index in the order of detection. The fourth colum represents the note duration calculated in Section 3.12. Then, all notes are mapped to their corresponding centroid durations in Section 3.13. Finally, the last colum is the output of the label assignment process of Section 3.14. The numbers used in the note label signify the note value type in a rhytmic metric.

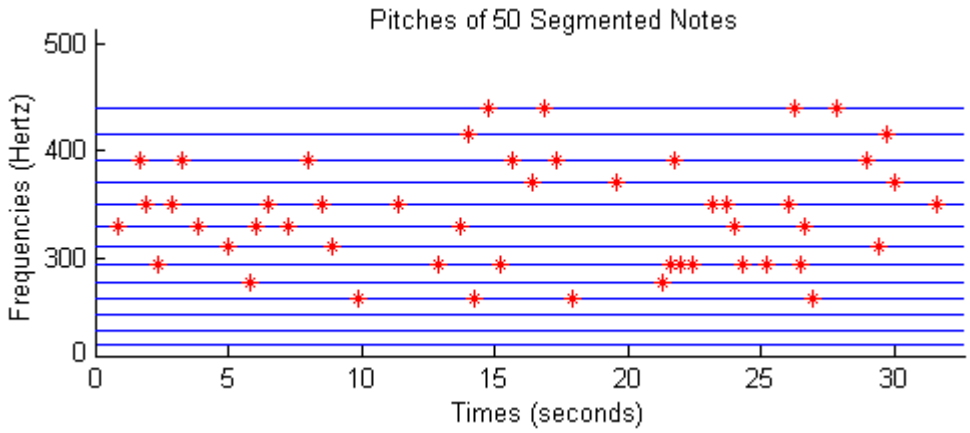

Figure 5.11: Pitches found for the fifty segmented notes.

Table 5.1**:** Results obtained in various steps of implementation.

#	Pitch (Hertz)	Note	Duration (seconds)	Mapped Duration (seconds)	Label
1	329.6300	E	1.5012	1.4935	E2
2	392.0000	G	0.1968	0.1964	G16
3	349.2300	F	0.1903	0.1964	F16
4	293.6600	D	0.7382	0.7405	D4
5	349.2300	F	0.3800	0.3773	F8
6	392.0000	G	0.3805	0.3773	G8
7	329.6300	E	0.7395	0.7405	E4
8	311.1300	D#	1.4996	1.4935	D#2
9	277.1800	C#	0.2108	0.1964	C#16
10	329.6300	E	0.1847	0.1964	E16
11	349.2300	F	0.7312	0.7405	F4
12	329.6300	E	0.7507	0.7405	E4
13	392.0000	G	0.7480	0.7405	G4
14	349.2300	F	0.3783	0.3773	F8
15	311.1300	D#	0.3844	0.3773	D#8
16	261.6300	C	1.4875	1.4935	C2
17	349.2300	F	1.5016	1.4935	F2
18	293.6600	D	1.4969	1.4935	D2
19	329.6300	E	0.2089	0.1964	E16
20	415.3000	G#	0.3677	0.3773	G#8
21	261.6300	C	0.1990	0.1964	C16
22	440.0000	A	0.7271	0.7405	A4
23	293.6600	D	0.2048	0.1964	D16
24	392.0000	G	0.7325	0.7405	G4
25	369.9900	F#	0.7504	0.7405	F#4
26	440.0000	A	0.1981	0.1964	A16
27	392.0000	G	0.7390	0.7405	G4
28	261.6300	C	0.3802	0.3773	C8
29	369.9900	F#	2.9927	2.9960	F#1
30	277.1800	C#	0.3940	0.3773	C#8
31	293.6600	D	0.1842	0.1964	D16
32	392.0000	G	0.1840	0.1964	G16
33	293.6600	D	0.2050	0.1964	D16
34	293.6600	D	0.7207	0.7405	D4
35	349.2300	F	0.7557	0.7405	F4
36	349.2300	F	0.3708	0.3773	F8
37	329.6300	E	0.2011	0.1964	E16
38	293.6600	D	0.3782	0.3773	D8
39	293.6600	D	1.4790	1.4935	D2
40	349.2300	F	0.2059	0.1964	F16
41	440.0000	A	0.1872	0.1964	A16
42	293.6600	D	0.1949	0.1964	D16
43	329.6300	E	0.1795	0.1964	E16
44	261.6300	C	0.3689	0.3773	C8
45	440.0000	A	1.4888	1.4935	A2
46	392.0000	G	0.7529	0.7405	G4
47	311.1300	D#	0.2075	0.1964	D#16
48	415.3000	G#	0.3669	0.3773	G#8
49	369.9900	F#	0.1927	0.1964	F#16
50	349.2300	F	2.9993	2.9960	F1

5.10 Calculation of Note Durations

We calculated note durations of fifty notes by substracting the corresponding time locations of the two successive notes' physical onsets as stated in Section 3.12. We obtained fifty note durations which will be used in the note value types detection. We also provide these values in Table 5.1.

5.11 Note Value Types Detection

Until this step, we do not have any prior knowledge about the note value types of the notes segmented. We discussed how we try to approximate the note type values from the data of note durations in Section 3.13. In the detailed explanation of this approximation, we introduced a variable named $periPer$ whose default value is 0,8. This value is used while seeking the possible note durations in the entourage of a note centroid. This algorithm seems to resemble *K-Means* algorithm (although the contexts are different) in the sense that both require to be given the number of means to be detected, as an input [29]. Here we deal with the distinct note types and our K is five, apparently. As a result, we got two important new information which are distinct note value types and mapped note durations, respectively. The note value types detected for the file test-5-50-notes.wav are 2.9960, 1.4935, 0.7405, 0.3773, 0.1964, 0.0982, and 0.0491 seconds, in descending order. One important thing is that these are the generalized note types that we would like to map to the note durations found in the previous step instead of their values obtained from the measurement conducted between the successive onsets' time locations. Thus, the meaning of the mapped note durations are now revealed. At the moment, all note durations can be one of the five generalized and distinct note type values. Finally, we will use the mapped note durations in the final step of our implementation while we try to give a complete descriptive label to the notes segmented. See Figure 5.13 in order to investigate the five distinct note value types. Each note value type is exactly the half of the one previous note type. In the figure, red asteriks show the centroids calculated by taking the mean of the data points in its entourage. Blue asteriks show our current data points which are indeed the note durations calculated at the previous step. They are distributed in the plane as if they are coming from the different clusters. The case

of the centroid and its entourage data points can be seen better with a close-up plot (see Figure 5.12).

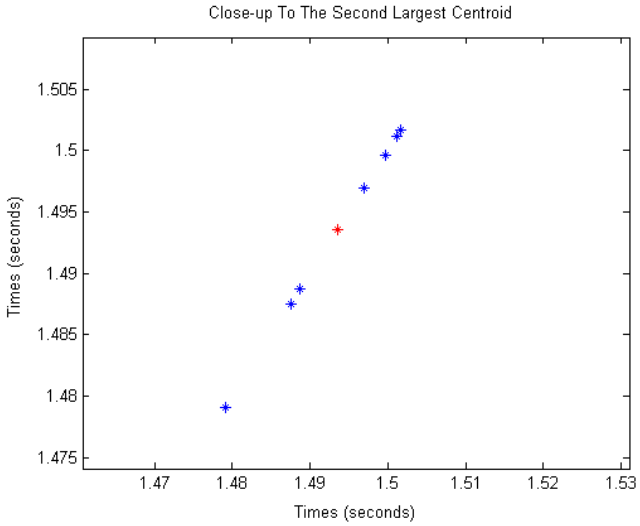

Figure 5.12: Second largest centroid zoomed.very near to 1,5 seconds.

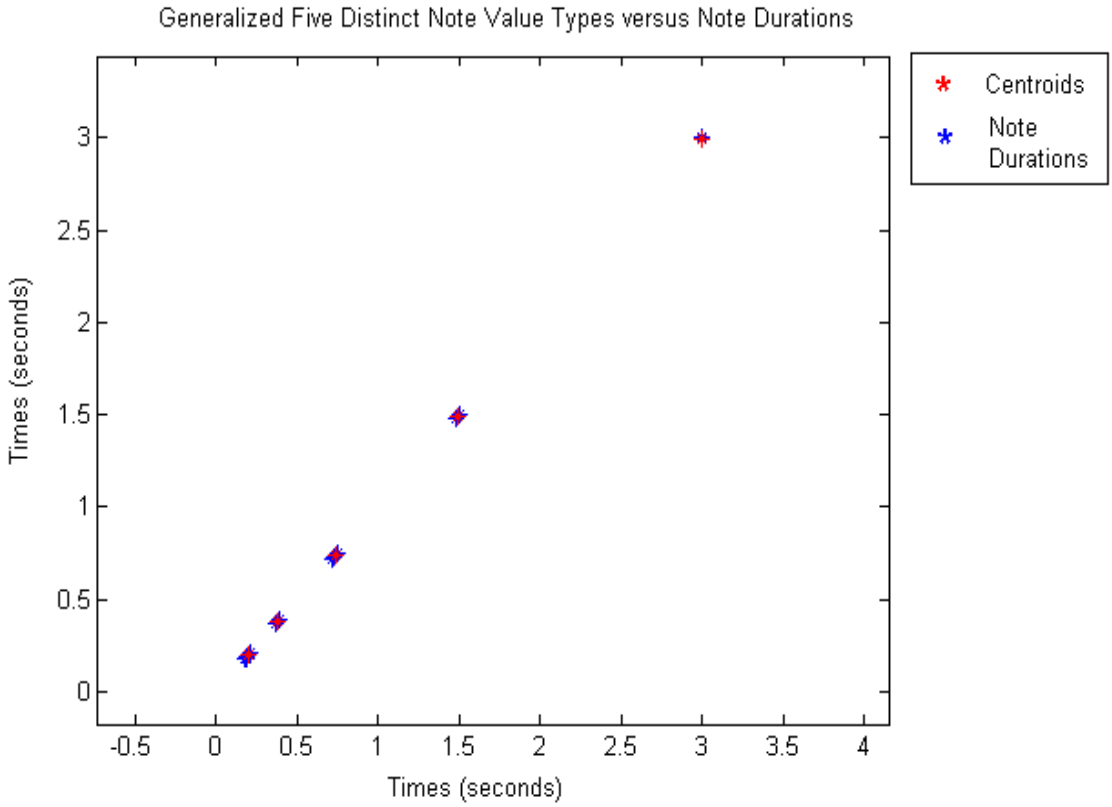

Figure 5.13: Distinct note value types-centroids seen with data points.

5.12 Note Labels Assignment

In Section 1.2.4.3, we have investigated different note value types that are the basis for our test data. In Section 1.2.7.2, we explained the rationale of the relative change of these note value types under the circumstances that tempo gets slower or faster. In Section 3.14, we have introduced the proposed approximation method of mapped note durations to absolute note value types in a given tempo. So, we give the assignment routine that map note durations and the names of notes found. It returns the fully qualified note name labeled with its note value type likewise the examples of E2, F16, D4, G8, F#1 which are all given for a different value type. We got fifty note labels in this format. All note labels can be seen in Table 5.1. We will use these note labels in calculating the accuracy of the algorithm that is validated for a given data test-5-50-notes.wav.

5.13 Performance Measurement

Performance measure of our proposed algorithm is its accuracy. We calculated the percentage of correctly identified notes. The accuracy can be given as the following

$$Accuracy = \frac{number\ of\ accurately\ segmented\ notes}{total\ number\ of\ notes} \qquad (5.1)$$

where $Accuracy \in \mathbb{R}\ and\ 0 \leq Accuracy \leq 100$. We obtained an accuracy value of 100% for the data file test-5-50-notes.wav. We have found fifty accurately segmented notes in our test data.

We tested the algorithm in an experiment of thousand randomly created wave files which contain the notes from middle octave and with first five note value types. As a result, our proposed transcription algorithm has an average accuracy of 97,6 with the tuned parameters (see Section 5.14). In this experiment, we successfully transcribed the music scores of 960 test files with one hundered per cent accuracy.

For 12 files, we got partial accuracies varying from 40% to 80%. The decrease in the accuracy is caused by the validation method of transcribed scores with the ground-truth data. If the segmentation of one note is skipped or one exces note could not be

eliminated during the spurious attack elimination, this situation affects linearly the validation of all subsequent transcribed scores. In this experiment, such a case occurs for the file $test - 0404.wav$ for which the algorithm yields an accuracy of 41,2. We show a fragment of the comparison process of ground-truth data with transcripted scores in Figure 5.14.

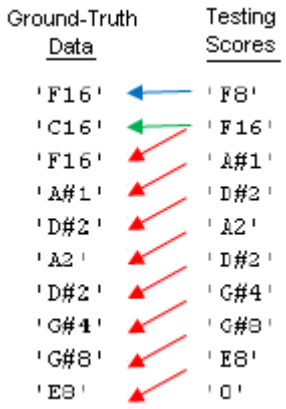

Figure 5.14: A fragment of the comparison matrix for the test file *test-0404.wav*.

Figure 5.14 explains the cases which decrease the accuracy. We first see that the note $F16$ was transcribed as $F8$. The correct note duration may not be properly mapped to the right centroid. All in all, this is a false transcription example even though the note name and its pitch were detected appropriately. Secondly, the algorithm skipped to detect $C16$ after $F16$ and that's why we add a padding zero to the end of the testing scores. This error causes a series of failures in the validation of testing data despite the rest was transcribed successfully (follow the red arrows). This massive decrease in the accuracy is due to the subsequent dependence of the algorithm pieces. If one fails, the subsequent pieces fail.

Beside these, we got zero acccuracy for 28 test files. One of the main reason for this bad result is randomness. Notes and their durations were created in a random fashion. This approach does not fit with the general music composition style [10]. In general, harmony, consonance and tonality are the main goal to reach for composers [11]. However, we discarded these aesthetic elements while we create our synthetic test data in order to force our algorithm. Another reason for the fall of accuracy is that the absence of one of more value types in note value type detection process. Lastly, for

some files, our algorithm parameters may be re-tuned even the majority of the test files we got 100% accuracy. For example, spurious attack elimination is a good example of the requirement for parameter tuning. If the threshold is not selected well, elimination fails to remove excess attacks which directly affects the performance. We explain the determination of parameters values in Section 5.14.

Another criterion to evaluate the performance of our algorithm is its squared error sum. This is computed for each test file used in an experiment. In Section 3.9, we segment K notes from a test file. Then, we eliminate the spurious notes in Section 3.10. The number of remaining notes is denoted by K'. For each detected note i, we calculate an experimental error e_i by substracting the look-up table frequency value (ltf) from the most powerful frequency value (f') obtained by the power spectrum analysis (see Section 3.11). For all remaining notes for a test file, we can formulate the squared error sum $b_{m,n}$ as follows

$$b_{m,n} = \sum_{i}^{K'} {e_i}^2 = \sum_{i}^{K'} ({f'}_i - ltf_i)^2 \tag{5.2}$$

where m is the number of current test file and n is the window size (see Table 5.3) and the jump amount (see Tables 5.4, 5.5 and 5.6) in the second and third experiments, respectively.

5.14 Experimental Determination of Parameters

The performance of the transcription algorithm is affected from the value of parameters used in various steps. We made some experiments to determine the crucial parameter values. One of these parameters is the coefficient value used to compute the threshold in the spurious attack elimination phase. The other one is the window size utilized during the construction of amplitude envelope. Thirdly, the jump amount affects the performance of the algorithm. We conducted an experiment to show how we calibrated the proportional ratio of window size and jump amount.

In our first experiment, we try to improve the performance of spurious attack elimination process. This process depends on to the value of a coefficient given during the runtime because we separately calculate a treshold value for each data file in order to eliminate the redundant attacks (see Section 3.10). The threshold is calculated via the formula $h = mc$ where h is threshold, m is the median of attacks lengths, and c is the coefficient. For randomly created 50 files (each consisting of 50 notes), we tested 10 different values of the coefficient which vary from 0.1 to 1.0 by an increment of 0.1. We got the best average accuracy of 98% while the coefficient value is 0.3 (see Table 5.2). After the coefficient value continues to increase, the average accuracy begins to decrease (see Figure 5.15). That's why we decided to use the coefficient value as 0,3 in the experiement where we got an accuracy of 97,6 for thousand files.

Table 5.2: Coefficient values and corresponding average accuracies.

Coefficient	Average Accuracies
0,1	68,451148
0,2	98
0,3	98
0,4	96,04
0,5	82,84
0,6	58,48
0,7	26,84
0,8	8,48
0,9	4,96
1	4,2

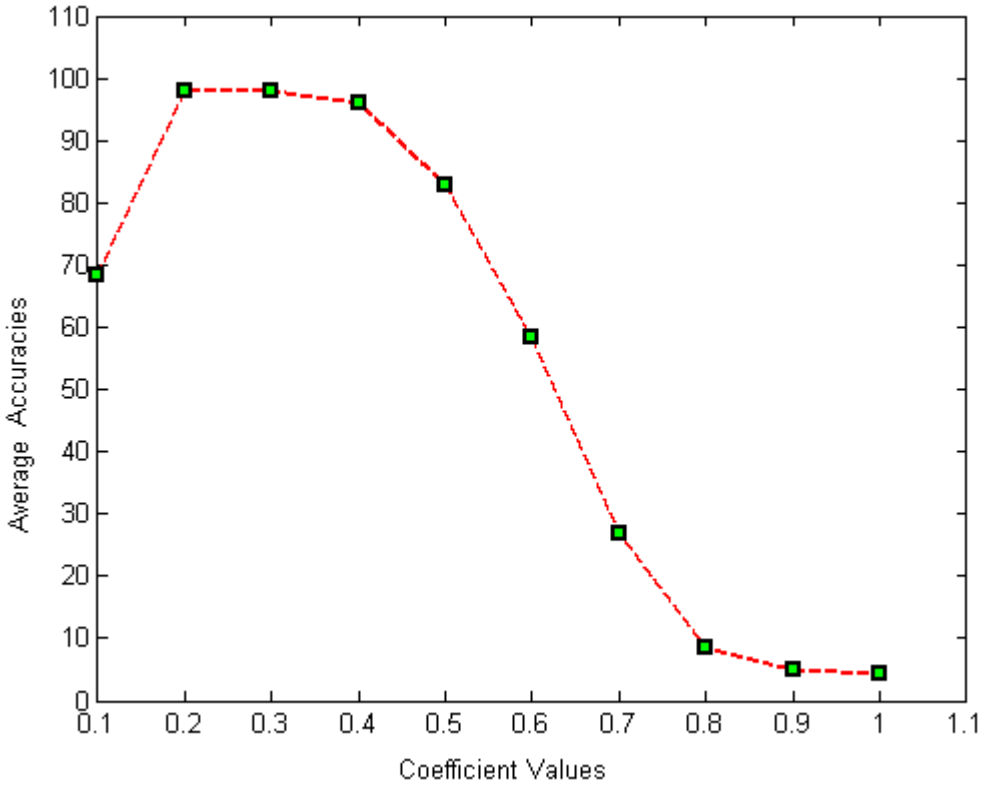

Figure 5.15: Average accuracies of 50 test files for 1st experiment.

In the second experiment, we try to understand how window size influences the average accuracy and squared error sum. We deal with the window size while we construct the amplitude envelope of the signal (see Section 3.4). In this phase, we take the frames of signal and we get the maximum signal value for each frame. We would like to know if the size of frame(in terms of samples) matters of not. We conducted the experiment with 18 different window sizes. They vary from 32 to 100 in an increment amount of four. We got the optimum result when the window size is equal to 56. For this value, we obtained an average accuracy of 98 for 50 randomly created test files. Concurrently, the average squared error sum must get the smallest value while the average accuracy must take the largest value. Error rate decreases in parallel with the increase of window size (see Figure 5.17). Approximately, the average squared error sum is 26,73 when window size is equal to 56 (see Table 5.3). After window size has once reached the value of 48, the accuracy did not decrease under the value of 93 (see Figure 5.16).

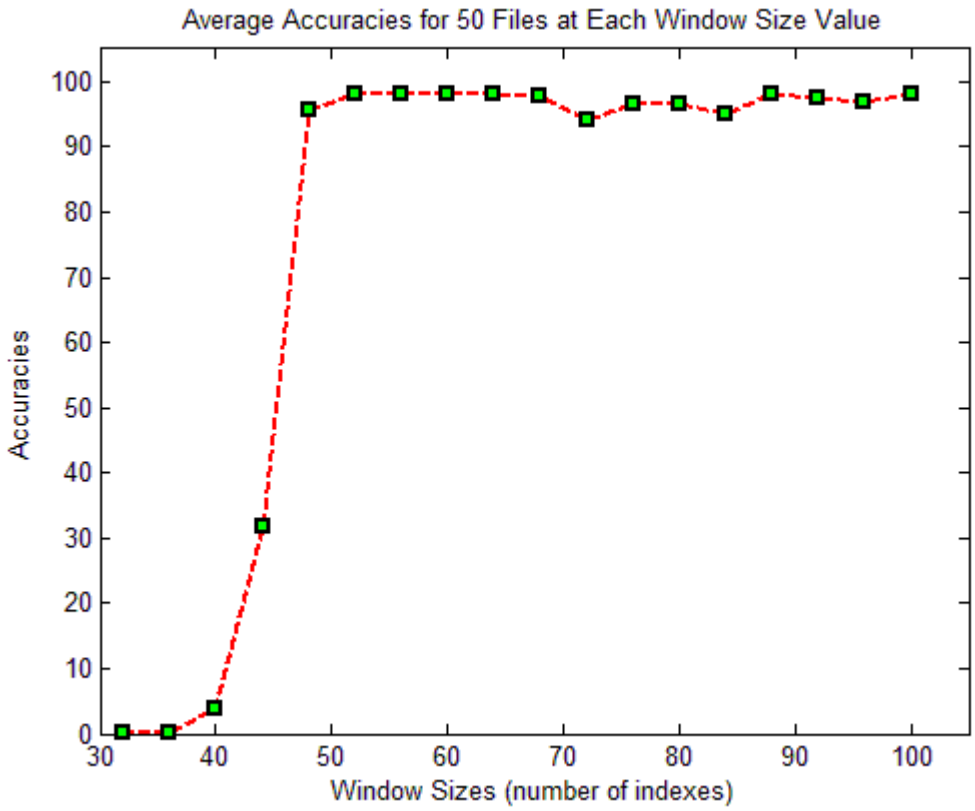

Figure 5.16: Average accuracies of 50 test files for 2nd experiment.

Table 5.3: Average accuracies and squared error sums for varying window sizes.

Window Sizes	Average Squared Error Sums	Average Accuracies
32	11212,0966620641	0,1358
36	199966,837415658	0,2443
40	1526,61161461678	3,7289
44	423,134603592385	31,6979
48	37,9286597202256	95,4902
52	27,3029560652964	98
56	26,7289881345325	98
60	27,4074577880511	98
64	28,2911981556218	98
68	28,1380003519404	97,72
72	32,5115328313859	93,9475
76	36,6055628080731	96,68
80	33,0921825399188	96,68
84	32,9430845575891	95,1114
88	31,6540005042485	98
92	30,4901939993109	97,3725
96	30,5439680754312	96,8627
100	32,3222016285271	98

We decided to use a window size value as 64 in the experiement where we got an accuracy of 97,6 for thousand files because the average accuracy still stands for 98% while the average squared error sum is very close to the value in the optimum case (see Figure 5.17). Computationally, algorithm may work relatively faster while window size is equal to 64 instead of 56.

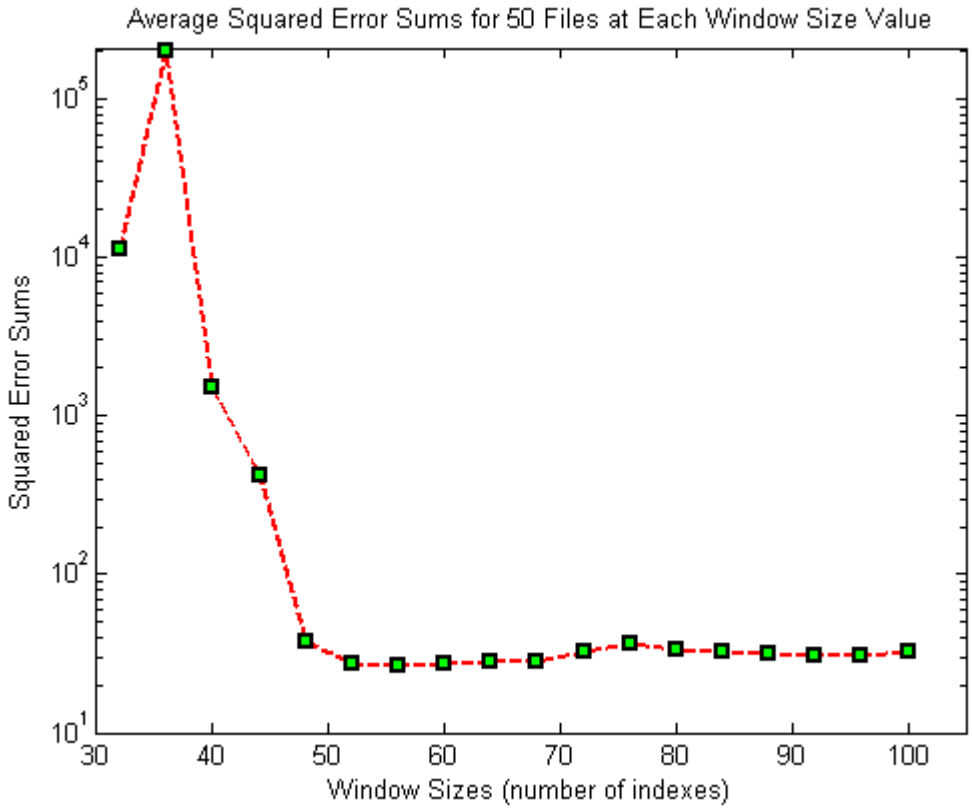

Figure 5.17: Average squared error sums of 50 test files for 2nd experiment.

In the third experiment, we investigated the relationship between the window size and jump amount. Both parameters are used in the construction of the amplitude envelope (see Section 3.4). We want to infer for which value of jump amount we can get the best result. For the three different window sizes including the optimum window size value of the previous experiment, we conducted twenty different jump amounts for fifty different randomly created test files. The window sizes are 50, 56, and 100, respectively. Jump amout vales vary from 5 to 100 by increments of five. For the window sizes 50 and 56, we tested the maximum jump amount values 50 and 55, respectively. The size of jump amount should not exceed the size of the window

size. For three different window sizes, we got three different best results which are provided in Tables 5.4, 5.5, and 5.6.

When window size is equal to 50, we got the optimum result while the size of jump amount is 20. In this case, the average accuracy for 50 files is 94,88 and the average squared error sums is 35, 25 (see Table 5.4). The secondary best result was obtained when the jump amount is equal to 15. In this case, we got an average accuracy of 94,08 which is lesser than the optimum result's one.

Table 5.4: Average accuracies and squared error sums for varying jump amount values when window size is 50.

Window Size=50		
Jump Amount	Average Accuracies	Avg. Squared Error Sums
5	0,2371	63783565,68
10	0,1329	28873072,93
15	94,08	26,97325893
20	94,88	35,25427532
25	92,1961	195081,7175
30	84,3184	195078,6058
35	37,0369	25,70871458
40	6,48	8,462197669
45	5,88	4,870550914
50	5,84	4,887836793
55	N/A	N/A
60	N/A	N/A
65	N/A	N/A
70	N/A	N/A
75	N/A	N/A
80	N/A	N/A
85	N/A	N/A
90	N/A	N/A
95	N/A	N/A
100	N/A	N/A

For this value of the window size, we can use the second optimum result whose average accuracy is equal to the most optimum solution's one. In addition, the average squared error sums is relatively smaller than the optimum solution. So, they can be used interchangeably. Thus, the ratio of the window size over jump amount is equal to 3,33 which is close to 4.

When window size is equal to 56, we got the optimum result while the size of jump amount is 15. In this case, the average accuracy for 50 files is 98 and the average squared error sums is 27, 1 (see Table 5.5). The secondary best result was obtained when the jump amount is equal to 20. In this case, we got an average accuracy of 95,582 which is lesser than the optimum result's one.

Table 5.5: Average accuracies and squared error sums for varying jump amount values when window size is 56.

Window Size=56		
Jump Amount	Average Accuracies	Avg. Squared Error Sums
5	0,208	59296361,37
10	0,0757	40381490,33
15	98	27,05006718
20	95,582	36,27277901
25	91,0361	195080,7427
30	86,3145	32,44605572
35	41,4533	25,66930479
40	6,28	7,08997871
45	5,88	390097,5907
50	5,72	4,954995228
55	5,32	4,813667549
60	N/A	N/A
65	N/A	N/A
70	N/A	N/A
75	N/A	N/A
80	N/A	N/A
85	N/A	N/A
90	N/A	N/A
95	N/A	N/A
100	N/A	N/A

For this value of the window size, the ratio of the window size over jump amount is equal to 3,73 which is also close to 4.

When window size is equal to 100, we got the optimum result while the size of jump amount is 25. In this case, the average accuracy for 50 files is 98 and the average squared error sums is 32,3 (see Table 5.6). The secondary best result was obtained when the jump amount is equal to 20. In this case, we got an average accuracy of 96,68 which is lesser than the optimum result's one.

Table 5.6: Average accuracies and squared error sums for varying jump amount values when window size is 100.

Window Size=100		
Jump Amount	Average Accuracies	Avg. Squared Error Sums
5	1,4335	17554593,9
10	0,84	1755644,673
15	4,4722	25357118,98
20	96,68	34,13469257
25	98	32,32220163
30	90,5498	31,87624639
35	48,7609	33,89132863
40	6,44	8,455331799
45	6,08	195051,3909
50	5,76	5,121477112
55	5,4	5,000779708
60	5,12	4,804608289
65	4,24	8,10920381
70	3,88	4,128848632
75	3,4	195050,0579
80	3,44	195049,3379
85	3,32	195049,2238
90	2,96	195048,823
95	3	2,167629159
100	2,84	2,092495119

For this value of the window size, the ratio of the window size over jump amount is equal to 4, exactly. Therefore, we can claim that when the ratio of the window size over jump amount approximates to 4, we get an average accuracy of 98 and an average square error sum of 29,7 ([27,1 + 32,3]/2). That's why we decided to use this ratio in the experiment we got an accuracy of 97,6 for thousand files by using a window size of 64 and a jump amount of 16. To conclude, we provide six figures to explain the situation of average accuracies and squared error sums in terms of the change in jump amount for three different window sizes (see Figures 5.18, 5.19, 5.20, 5.21, 5.22, and 5.23).

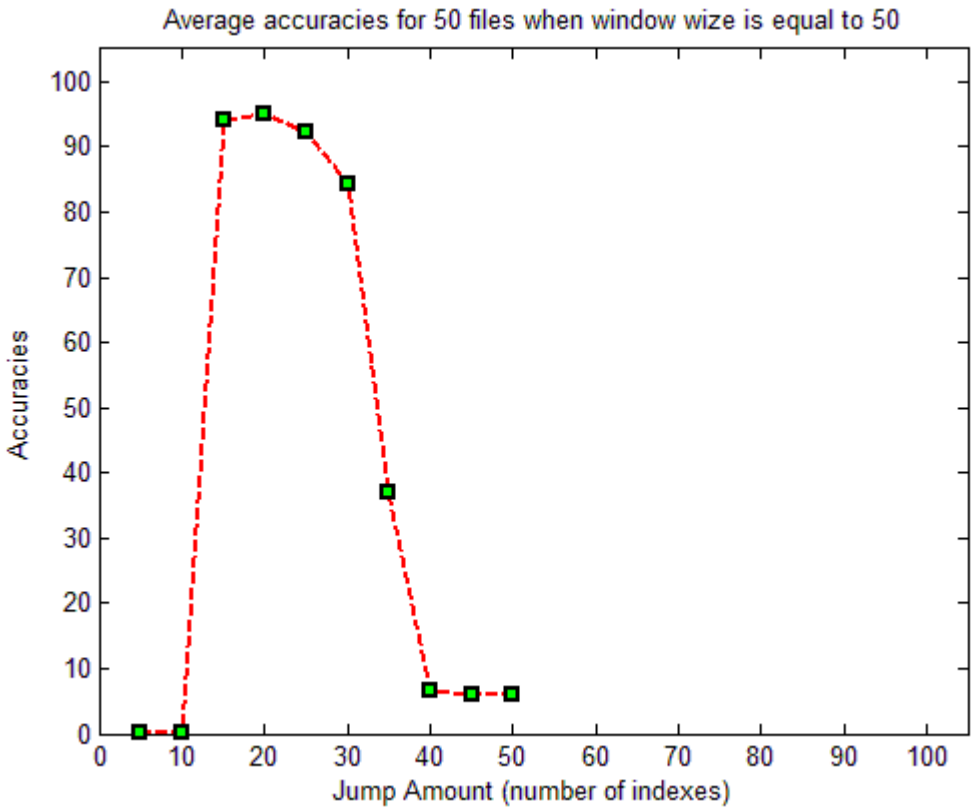

Figure 5.18: Average accuracies for 3rd experiment when window size is 50.

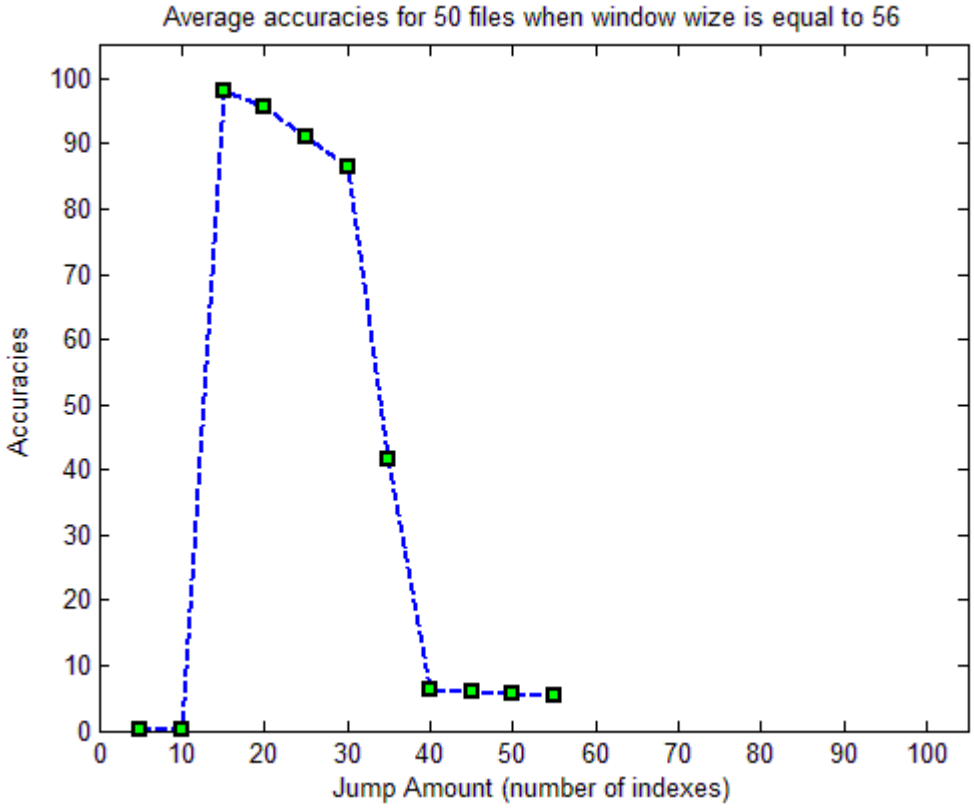

Figure 5.19: Average accuracies for 3rd experiment when window size is 56.

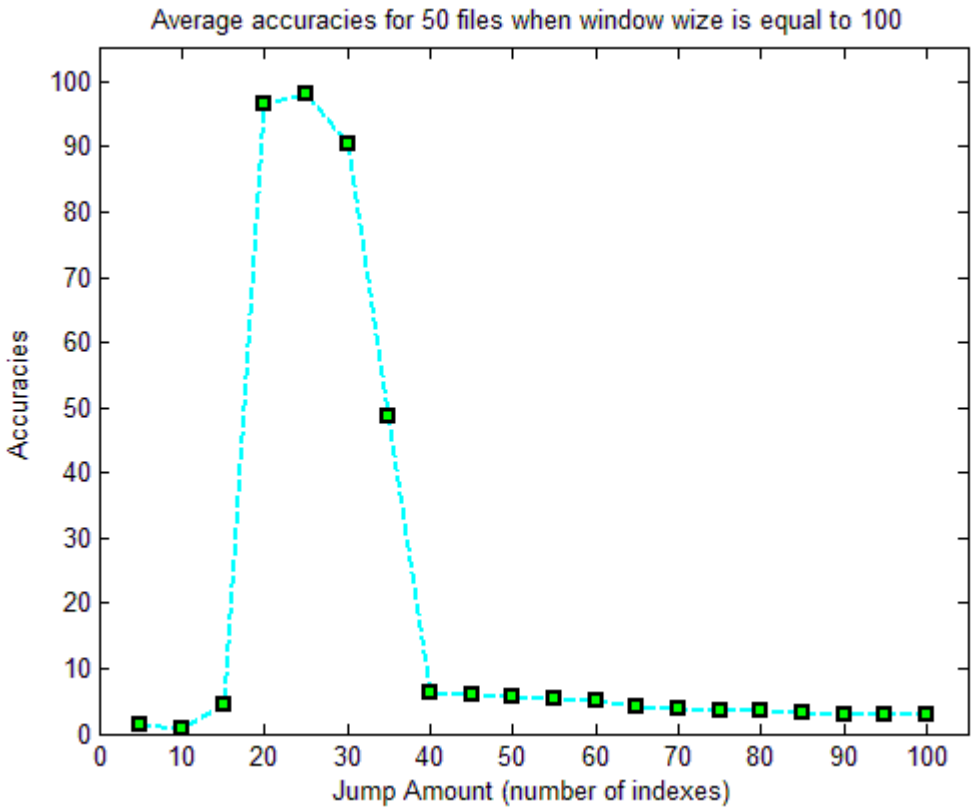

Figure 5.20: Average accuracies for 3rd experiment when window size is 100.

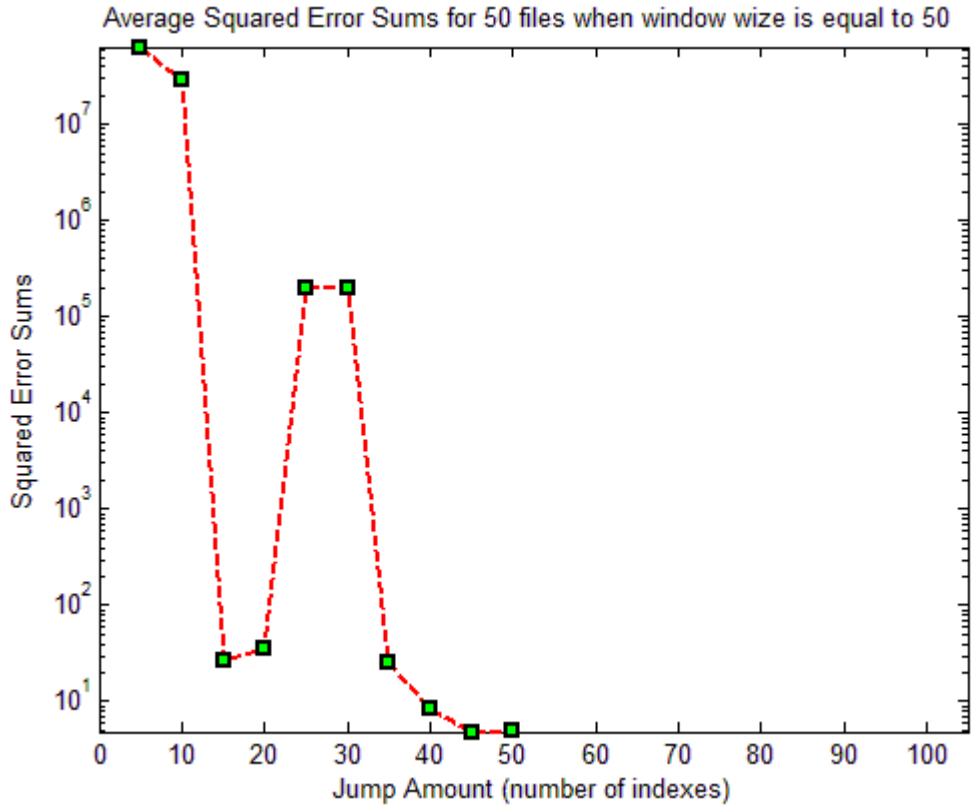

Figure 5.21: Average squared error sums for 3rd experiment when window size is 50.

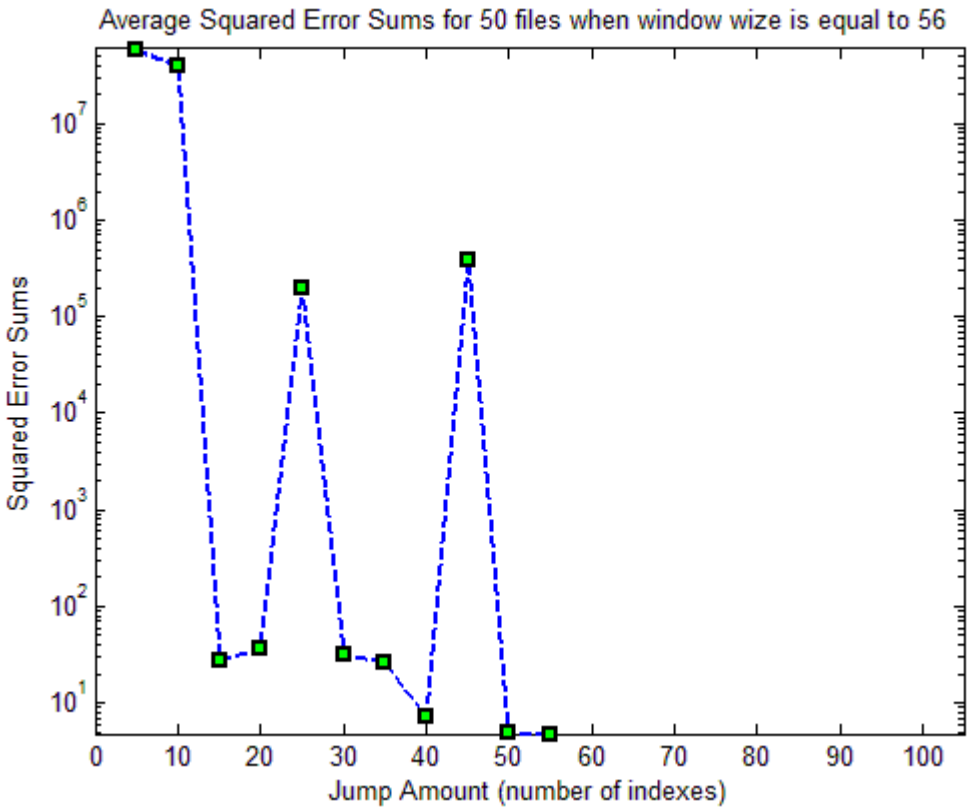

Figure 5.22: Average squared error sums for 3rd experiment when window size is 56.

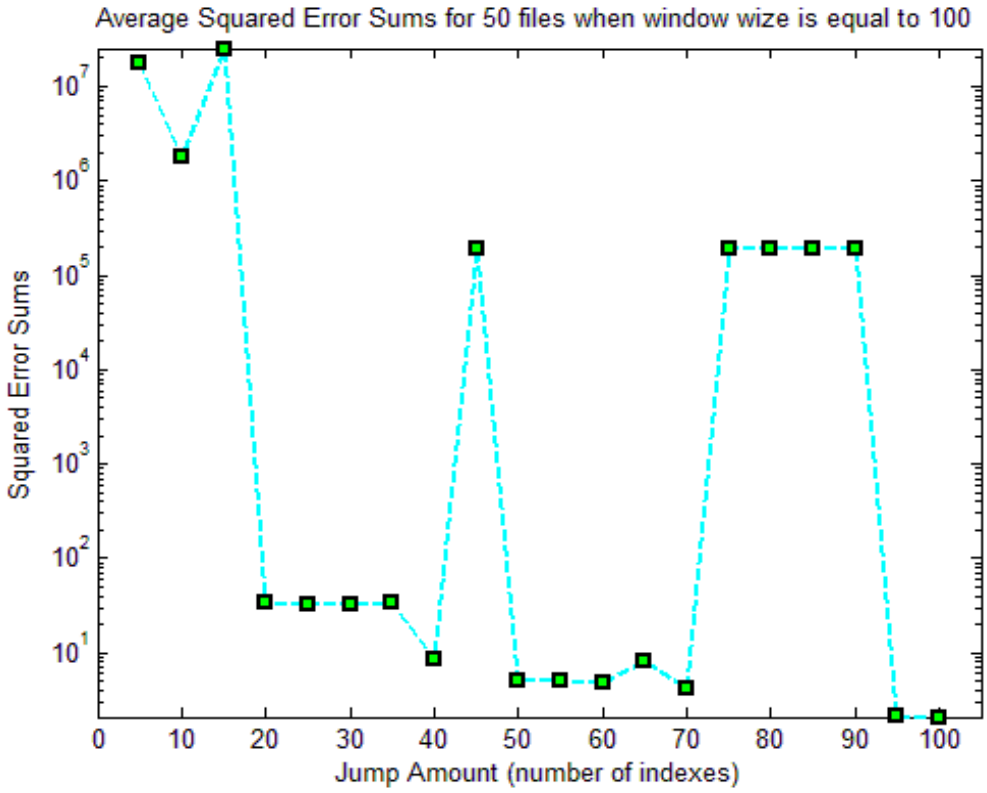

Figure 5.23: Average squared error sums for 3rd experiment when window size is 100.

Chapter 6
Conclusion

In this thesis, we have discussed automatic music transcription of music notes for an audio signal. We attempted to transcript percussive note onsets for the instrument piano. For this purpose, we used our data creator program in order to provide random data to our implementation. To transcript notes, we have used both temporal and frequency domain features.

Our aim was to detect music notes and estimate their physical attributes like onset and attack times, durations, labels, value types and their pitches. We have applied a series of techniques to obtain all the mentioned attributes. We believed that abrupt changes in the amplitude envelope could be a guide to find out the note events. Because the envelope had so much unrelated peaks, we needed to smooth it to get more comfortable data to work on. We benefitted from the sudden increase in the mean of slopes data. We detected these increases by calculating cumulative sums. By using the data obtained until this step, we segmented notes in audio signal. Some of them were truely related to music events but some were not. We eliminated these spurious ones from our data. Until right now, these were our temporal features. In order to obtain notes pitches, we made a frequency domain operation like FFT. Beside this, we got note names by only mapping their pitches into a look-up table of nine octaves frequencies for piano instrument. We estimated note durations from the differences of two successive onsets. Then, we have splitted note durations in some clusters. We used the mean of each cluster as a generalized note value type. Moreover, the mapped durations were assigned to the segmented notes. Each note got a label according to its mapped duration, e.g. F#4. As a result, we had a series of labeled notes like F#4.

Our proposed transcription algorithm has an average accuracy of 97,6 for randomly created thousand synthetic data with the tuned parameters. The performance of the algorithm is affected from the values of parameters in various steps. We provide experimental results to illustrate the tuning of parameters. Especially, the values of window size and jump amount in construction of amplitude envelope and the threshold used in spurious attack elimination have direct influence on the rate of accuracy. As a future work, the algorithm can be improved to detect 32^{nd} and 64^{th} notes in the middle octave, as well as the other octaves. Moreover, the algorithm can be tested on the real data in the future. This may well be more chanllenging for our algorithm.

References

[1] Poliner G. E. and Ellis D. P. W., “A Discriminative Model for Polyphonic Piano Transcription”, *EURASIP Journal on Advances in Signal Processing*, 2007.

[2] Plumbley M. D., Abdallah S. A., Bello J. P., Davies M. E., Monti G. and Sandler M. B., “Automatic Music Transcription and Audio Source Separation”, *Cybernetics and Systems*, vol. 33, no. 6, pp. 603-627, 2002.

[3] Klapuri Anssi P., “Automatic Music Transcription as We Know it Today”, *Journal of New Music Research*, vol. 33, no. 3, pp. 269-282, 2004.

[4] Bello J. P., Monti G., and Sandler M. B., “Techniques for Automatic Music Transcription”, *Proceedings of the International Symposium on Music Information Retrieval*, 2000.

[5] Brown J. C. and Zhang B., "Musical Frequency Tracking Using the Methods of Conventional and Narrowed Autocorrelation", *Journal of the Acoustical Society of America*, vol. 89, no. 5, pp. 2346-2354, 1991.

[6] de Cheveigné A. and Kawahara H., “YIN, A Fundamental Frequency Estimator for Speech and Music”, *Journal of Acoustical Society of America*, vol. 111, no. 4, pp. 1917-1930, 2002.

[7] Klapuri A. P., “Multiple Fundamental Frequency Estimation Based on Harmonicity and Spectral Smoothness”, *Speech and Audio Processing*, vol. 11, no. 6, pp. 804-816, 2004.

[8] Lahat M., Niederjohn R, and Krubsack D, “A Spectral Autocorrelation Method for Measurement of the Fundamental Frequency of Noise-Corrupted Speech”, *Acoustics, Speech and Signal Processing*, vol. 35, no. 6, pp. 741-750, 1987.

[9] Kunieda N., Shimamura T., and Suzuki J., "Robust Method of Measurement of Fundamental Frequency by ACLOS – Autocorrelation of log Spectrum", *In Proceedings of the Acoustics, Speech, and Signal Processing*, pp. 232–235, 1996.

[10] Dorrell P., *What Is Music? Solving A Scientific Mystery*, Philip Dorrell, 2005.

[11] Miller M., *The Complete Idiot's Guide to Music Theory*, Second Edition, Alpha Books, 2005.

[12] Benson D., *Music: A Mathematical Offering*, pp. 17, Cambridge University Press, 2006.

[13] Zhu Y., Kankanhalli M., and Gao S., "A Method for Solmization of Melody", *IEEE International Conference on Multimedia and Expo*, 2004.

[14] *Electronic Musical Instrument*, http://en.wikipedia.org/ wiki/Electronic_music al_instrument, 2010.

[15] *Musical Instrument Digital Interface*, http://en.wikipedia.org/wiki/MIDI, 2010.

[16] *Learn About MIDI*, http://www.midi.org/aboutmidi/index.php, 2010.

[17] *Standard MIDI File Format Specification 1.1*, International MIDI Association, http://duskblue.org/proj/toymidi/midiformat.pdf, 2010.

[18] *Standard MIDI Files*, http://www.omega-art.com/midi/mfiles.html, 2010.

[19] *MIDI File Format*, http://cs.fit.edu/~ryan/cse4051/projects/midi/midi.html,2010.

[20] *A Crash Course on the Standard MIDI Specification*, http://www.skytopia.com/ project/articles/midi.html, 2010.

[21] Richard S., "MIDI Programming", *A Complete Study*, http://www.petesqbsite. com/sections/express/issue18/midifilespart1.html, 2010.

[22] Irvine, T., "An Introduction to Music Theory", *Piano Page*, http://www.vibration data.com/piano.htm, 2010.

[23] Schloss, W. A., *On the Acoustic Transcription of Percussive Music From Acoustic Signal to High-Level Analysis*, Ph. D. Dissertation, Center for Computer Research in Music and Acoustics, Stanford University, 1985.

[24] Gordon, J. W., "The Perceptual Attack Time of Musical Tones", *Journal of Acoustical Society of America*, vol. 82(1), pp. 88-105, 1987.

[25] Young, R. W., "Terminology for Logarithmic Frequency Units", *Journal of the Acoustical Society of America*, vol. 11, pp. 134–139, 1939.

[26] Bello J. P., Daudet L., Abdallah S., Duxbury C., Davies M., and Sandler M. B., "A Tutorial on Onset Detection in Music Signals", *Speech and Audio Processing*, vol. 13, no. 5, pp. 1035-1047, 2005.

[27] McClellan J. H., Schafer R. W., and Yoder M. A., *Signal Processing First*, Pearson Prentice Hall, 2003.

[28] Oppenheim A. V., Willsky A. S., and Nawab S. H., *Signals and Systems*, Second Edition, pp. 321-322, and 384, 1997.

[29] Alpaydın E., *Introduction to Machine Learning*, MIT Press, pp. 29-31, 2004.

[30] Basseville M., “Detection of Jumps in Mean and Adaptive Filtering”, *Acoustics, Speech, and Signal Processing*, vol. 7, pp. 1092–1095, 1982

[31] Basseville M, “Detecting Changes in Signals and System – A Survey”, *Automatica*, vol. 24, no. 3, pp. 309-326, 1988.

[32] Klapuri A., “Sound Onset Detection by Applying Psychoacoustic Knowledge”, *Speech and Signal Processing*, in Proc. IEEE Int. Conf. Acoustics, pp. 115–118, 1999.

[33] Cooley, J. W. and Tukey J. W., "An Algorithm for the Machine Computation of the Complex Fourier Series", *Mathematics of Computation*, vol. 19, pp. 297-301.

[34] Biran A. and Breiner M., *Matlab 6 for Engineers*, Third Edition, pp. 634-638, Prentice Hall, 2002.

[35] Maddage N. C., "Automatic Structure Detection for Popular Music," *IEEE Multi Media*, vol. 13, no. 1, pp. 65-77, 2006.

[36] Garlan D. and Shaw M., *An Introduction to Software Architecture***,** CMU Software Engineering Institute Technical Report, CMU-CS-94-166, 1994.

Appendix A Variable Length Writing and Reading

A.1 Pseudocode of Variable Length Writing

```
WRITE-VARIABLE-LENGTH( value, file )
00      # initialize and define an integer R by making bitwise AND with 0x7F
01      R ← value & 0x7F
02
03      # bitwise RIGHT-SHIFT operation: division by 128
04      L ← value >> 7
05
06      # Check L > 0
07      if L > 0 then
08              do
09                      # bitwise LEFT-SHIFT: multiplication by 256
10                      K ← R << 8
11
12                      # bitwise AND operation for L with 0x7F
13                      I ← L & 0x7F
14
15                      # bitwise OR operation for I with 0x80
16                      temp ← I | 0x80
17
18                      # bitwise OR operation for K with temp
19                      data ← K | temp
20
21                      # L is assigned to value
22                      value ← L
23
24                      # loop condition will be tested with the value of L
25                      L ← value >> 7
26              while L > 0
27      else
28              # bitwise AND operation for value with 0x7F
29              data ← value & 0x7F
30
31      # writing data into the file
32      while TRUE
33              # writing data as one-byte character inside a file
34              write( data, file )
35
36              # if data's seven bit is set
37              if ( data & 0x80 )
38                      # divide data by 256 and assign the result to itself
39                      data ← data >> 8
40              else
41                      break
42      end while
```

Figure A.1: Algorithm of variable length writing.

A.2 Pseudocode of Variable Length Reading

```
READ-VARIABLE-LENGTH( value, file )
00      # we read an hexadecimal number from the file
01      V← read(file)
02
03      # bitwise AND operation with 0x80
04      R ← V & 0x80
05
06      # check R >= 1
07      if  R >= 1 then
08
09              # bitwise AND operation with 0x7F
10              R ← V & 0x7F
11              V ← R
12
13              do
14                      # V is multiplied by 128
15                      L ← V << 7
16
17                      # a new value is read from file
18                      Y ← read(file)
19
20                      # bitwise AND for Y with 0x7F
21                      R← Y & 0x7F
22
23                      # V is assigned to the addition of L and R
24                      V ← L + R
25
26                      # loop condition will be tested with the value of temp
27                      temp ← Y & 0x80
28
29              while temp > 0
30
31      else
32              # bitwise AND operation for V with 0x7F
33              R ← V & 0x7F
34              V ← R
35
36      return V
```

Figure A.2: Algorithm of variable length reading.

Appendix B Note Value Types Detection

B.1 Pseudocode of Round#1: Finding Centroids

```
FIND-CENTROIDS( NDur, periPer, disType )
00
01      # Initialization of groupCount and centroids array C
02      groupCount ← 1
03      C ← array[1 × disType]
04
05      # Iteration to calculate note value types
06      for   q ← 1  to  disType
07
08              # If current iteration is 1
09              if   q==1   then
10                      rightB  ← max(NDur)
11                      leftB ← rightB  × periPer
12              else
13                      leftB ← tempC  × periPer
14                      rightB ← tempC + tempC  × (1 − periPer)
15
16              # Notes around participates a mutual group
17              NAround ← NDur(NDur  ≥ leftB & NDur  ≤ rightB )
18
19              # Check number of elements if it is zero or not
20              if  s(NAround) == 0   then
21
22                      # No notes were found at this  qth iteration
23                      C_groupCount ← tempC
24
25                      # Pseudo centroid is calculated to advance
26                      tempC ← C_groupCount / 2
27                      groupCount ← groupCount  +  1
28
29              else
30                      # There are some notes around
31                      C_groupCount ← mean( NAround )
32
33                      # Pseudo centroid is calculated to advance
34                      tempC ← C_groupCount / 2
35                      groupCount ← groupCount  +  1
36      end for
37
38      # Return centroids
39      return  C
```

Figure B.1: Algorithm of finding centroids.

B.2 Pseudocode of Round#2: Mapping Note Durations

```
MAP-NOTE-DURATIONS( Ndur, C )
00
01      # Initialization mapped note durations
02      MapDur ← array[ 1 × length(NDur) ]
03
04      # Assignment of generalized durations to real durations
05      for  counter ← 1 to length(NDur)
06
07              # Finding most nearest centroid value to jth note duration
08              [row, column] ← MNV( C, NDur_j )
09
10              # Assign the centroid value as note value type to this note duration
11              MapDur( counter ) ← C(column)
12
13      end for
14
15      # Return mapped note durations
16      return  MapDur
```

Figure B.2: Algorithm of mapping note durations.

Appendix C Pseudocode of Note Label Assignment

```
ASSIGN-NOTE-LABELS( MapDur, NName )
00
01      # Initialization
02      Extensions ← {'1', '2', '4', '8', '16'}
03      Labels ← array[ 1 × length(NName) ]
04
05      # Loop produces note labels according to its mapped durations
06      for  j ← 1  to  length(NName)
07
08              # If mapped durations is one of the descendingly ordered note value type
09              switch( MapDur_j )
10
11                      # Case corresponds to whole note
12                      case  C_1
13                              # suffix to append is ` 1 '
14                              Labels(j) ← cat( NName_j , Extensions(j) )
15
16                      # Case corresponds to half note
17                      case  C_2
18                              # suffix to append is ` 2 '
19                              Labels(j) ← cat( NName_j , Extensions(j) )
20
21                      # Case corresponds to quarter note
22                      case  C_3
23                              # suffix to append is ` 4 '
24                              Labels(j) ← cat( NName_j , Extensions(j) )
25
26                      # Case corresponds to eighth note
27                      case  C_4
28                              # suffix to append is ` 8 '
29                              Labels(j) ← cat( NName_j , Extensions(j) )
30
31                      # Case corresponds to sixteenth note
32                      case  C_5
33                              # suffix to append is ` 16 '
34                              Labels(j) ← cat( NName_j , Extensions(j) )
35              end  switch
36      end  for
37
38      # Return fully qualified note labels i.e: G#16
39      return  Labels
```

Figure C.1: Algorithm of note label assignment.

Curriculum Vitae