FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Symbolic Harmonic Change Detection in the Tonal Interval Space

José Alexandre da Silva Macedo

MASTER DISSERTATION

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

Supervisor: Gilberto Bernardes

October 6, 2022

Resumo

Nesta dissertação, é proposto um modelo computacional para detetar mudanças de harmonia numa sequência musical simbólica. Um modelo computacional para a detecção de mudanças de harmonia torna-se fundamental para o campo da recuperação de informação musical, bem como para uma vasta área de outras aplicações musicais. Uma vez que uma grande parte da investigação é realizada utilizando áudio, a investigação com música simbólica é ainda muito reduzida e embrionária, uma vez que não existem muitos datasets organizados de musica simbólica e que a conversão de datasets de outros formatos para simbólico pode ser um processo moroso. Fazendo uso do *Tonal Interval Space* (TIS) [1], a aplicação de uma redução das notas que compõem os acordes, numa representação hierárquica em árvore, pretende eliminar ornamentos e notas não fundamentais à estrutura musical como pré processamento. Esta representação constitui a base da tarefa de detecção de mudanças de harmonia, onde o início de cada novo acorde é inferido, utilizando a versão atualizada da função de deteção de alterações harmónicas proposta por Ramoneda e Bernardes [2]. A avaliação do modelo proposto foi realizada através da aplicação sobre cinco conjuntos de música clássica de épocas e estilos distintos. Os resultados obtidos estão em linha com os valores encontrados para a função de detecção de mudanças de harmonia para áudio reportadas em [2] e [3], com os valores alcançados a rondarem os 60-80%, embora estes diminuam ligeiramente quando o processo de redução é aplicado. Os códigos, em Python, para a computação do modelo proposto com e sem redução harmónica estão disponíveis nos apêndices para efeitos de replicação e disseminação.

Keywords: Harmonic Change Detection, Symbolic Music, Harmonic Segmentation, Music Information Retrieval

Abstract

In this dissertation, a computational model for harmonic change detection in a symbolic musical sequence is proposed. A computational model for detecting harmonic changes is fundamental to the field of musical information retrieval, and to a wide area of other musical applications. Related research has been conducted to address musical audio signals. Adopting symbolic notation to the task is at an embryonic stage, since there are not many organized symbolic music datasets, and converting datasets from other formats to symbolic can be a time-consuming process. Making use of the *Tonal Interval Space* (TIS) [1], we adopt a strategy to reduce the component notes of a chord, in a hierarchical tree representation, to eliminate ornaments and notes that are not fundamental to the musical structure as a preprocessing stage. This representation forms the basis of an harmonic change detection model, where the beginning of each new chord is inferred, using the altered version of the harmonic change detection function proposed by Ramoneda and Bernardes [2]. The proposed model was evaluated on five datasets of classical music from different eras and styles. The results are in line with the values found for the harmonic change detection function for audio reported in [2] and [3], with the values achieved being around 60-80%, although these decrease slightly when the reduction process is applied. The codes, in Python, for computing the proposed model with and without harmonic reduction are available in the appendices for replication and dissemination purposes.

Acknowledgements

The biggest acknowledgement goes to my supervisor, Gilberto Bernardes, and to Pedro Ramoneda. Without their help and input, this dissertation would not have a successful completion. I can't thank both enough.

To my colleagues Daniel Ferreira Diogo and Noémia Cardoso Ferreira, whose mutual interest in the field of Music Information Retrieval led us to search in a similar quest on a similar direction. From all the struggles, aids and funny moments that we had, we built a good partnership and, most important of all, a very strong friendship.

To Ana, who helped me through the worst of times and became a pillar during those.

To all my friends at TEUP - Tuna de Engenharia da Universidade do Porto, for all the stories and the brotherhood.

Last but not least, a big "Thank You" to all my family, friends and colleagues at FEUP for their support and patience.

José Macedo

"The dawn of reason lights your eyes, with the key you realise to the kingdom of the wise"

Eric Woolfson & Alan Parsons

Contents

List of Figures

List of Tables

Abbreviations and Symbols

FEUP	Faculty of Engineering of the University of Porto
INESC TEC	Institute of Systems and Computer Engineering, Technology and Science
MIDI	Musical Instrument Digital Interface
SMC	Sound and Music Computing
MIR	Music Information Retrieval
ACR	Automatic Chord Recognition
BPS-FH	Beethoven Piano Sonata with Function Harmony
CAD	Computer-Aided Design
HCDF	Harmonic Change Detection Function
HCD	Harmonic Change Detection
TIV	Tonal Interval Vector
TIS	Tonal Interval Space
RN	Roman Numeral
BTC	Bi-directional Transformer (for Chord Recognition)
HT	Harmony Transformer
DFT	Discrete Fourier Transform
STFT	Short-Time Fourier Transform
CQTC	Constant-Q Transform Chromagram
PCP	Pitch Class Profiles
HPCP	Harmonic Pitch Classes Profiles
NNLS	Non-Negative Least-Squares
MHA	Multi-Head Attention
GTTM	Generative Theory of Tonal Music
FFN	Feed-Forward Networks
DHD	Directional Hamming Distance
TISMIR	Transactions of the International Society for Music Information Retrieval
MIREX	Music Information Retrieval Evaluation eXchange

Chapter 1

Introduction

1.1 Context

Harmonic Change Detection (HCD) is the task that studies the detection of chord boundaries (onset or chords in the musical surface). It is framed within the Music Information Retrieval (MIR) and Sound and Music Computing (SMC) fields, two of the most prolific areas of the last few decades in Music Computing's domain.

1.2 Motivation

The importance of harmony in the music spectrum is undeniable. In the computational side of things, there is a wide range of musical areas where harmony is an integral part of them, with HCD being one of the main tasks in MIR, covering many applications, like Structural Segmentation, Genre Classification or Music Generation, to cite a few.

The HCD topic has been studied in greater depth in recent years, mainly in SMC and MIR fields, where it can be used as a preprocessing strategy to detect chord boundaries in musical representations, and it has also been instrumental to Automatic Chord Recognition (ACR), which has been growing as a topic of interest in the field of MIR, not only for its commercial applications, but also for future and more advanced music analysis.

From a computational stand, HCD is the first step to a better understanding and creation of computed harmony, as the given information at the change of a chord yields the harmonic relationships.

While this task has been most prominent in audio processing, the growing interest in processing symbolically encoded music has given new horizons to this area and expanded its possibilities to other ones (e.g. musicology and data recovery).

The major difference between audio ACR and symbolic ACR is related to the processing of the data: audio contains expressive information (e.g. timbre) and it is usually represented in spectrograms and chromagrams (based on Discrete Fourier Transform (DFT) functions, e.g. Short-Time Fourier Transform (STFT)); while symbolic music contains a more abstract view of

music representation (e.g. a MIDI file generally delivers its own information given through a letter, which tells the note, and a number, which tells the octave).

Recently, the increasing evolution of the technologies for artificial intelligence and for machine and deep learning has provided newer ways to improve the efficiency on many computational music tasks, also bringing new horizons for this and other related tasks in the future.

1.3 Objectives

The main objective of this thesis is to define an harmonic change detection function inspired by [2] to be applied to symbolic music manifestations, seeking to outperform existing state-of-the-art models.

In greater detail, the following three main objectives will be pursued:

1. To advance a model for pitch distances between the component notes of a chord. It ultimately aims at reducing the symbolic musical surface to a minimal and constant chordal texture (e.g., triads), thus excluding all embellishments from the structure;

2. To propose a compatible and efficient method for detecting harmonic changes in symbolic music manifestations, based on a critical assessment of HCDF model (proposed by Ramoneda and Bernardes [2]) and its parameterization;

3. To evaluate the proposed method, based on standard evaluation metrics (e.g., *F-score*), and assess the future improvements to the model.

1.4 Methodology

To attain objective 1, which is to create a model for pitch distances, it is necessary to find rating reduction strategies in order to eliminate embellishment notes and reduce structure to its essential minimum (e.g. triads), in order to reduce the amount of information and avoid false positives in HCDF. For that matter, several distances within the TIS will be used, like the inter-note distance and dissonance, as well as the distance of the note to the entire chord cluster. The regional context (i.e., key) on TIS will also be valuable at this point.

Regarding objective 2, it will be necessary to adapt the HCDF proposed by Ramoneda and Bernardes [2] to process symbolic music notation. The main challenges here are the definition of a parametrization that is aligned with a reduced number of data entry points (in comparison to audio processing signals). Defining the otimal parametrization will lead through an extensive grid search to the set of parameters that best fit the problem at hand.

In relation to objective 3, a search for a varied and conclusive gathering of datasets with annotated symbolic data has to be made. In order to evaluate the obtained results and draw conclusions about them, a set of evaluation metrics is designed. The global evaluation will be made by *standard* metrics like *F-Score* and *Precision* and *Recall* [7]. Its results, based on the comparison between

dataset data and evaluation data, can lead to certain conclusions that can direct the work produced for future application developments or even the work itself (e.g. higher *F-Score* measures provide a more balanced prediction of string boundaries).

1.5 Preliminary Work

All the preliminary work was carried out in order to obtain the greatest amount of knowledge on the subject and on what has already been achieved in this field, besides getting acquainted with which innovations are being prepared today and which can be created in the future.

However, before starting with the coding itself, there is some preliminary work that can already be done. Such tasks have to do with a proper understanding of this procedure and how to promote satisfactory results. Therefore, a proper installation and execution of Python libraries like `TIVlib` [1] [8], `music21` [9] and `scipy` [10] and `libfmp` [6] [11] [12] [13] are mandatory for the smooth running of the final program.

1.6 Workplan

The proposed workplan for this dissertation will be based on the one presented in [2], but with substantial differences: for example, changing the type of input promotes significant changes in the work plan formed. These differences will be covered under the innovations portrayed in [3].

A tentative Workplan has been created, in order to represent a formal map for the dissertation: it serves basically as a guiding document for its future endeavours.

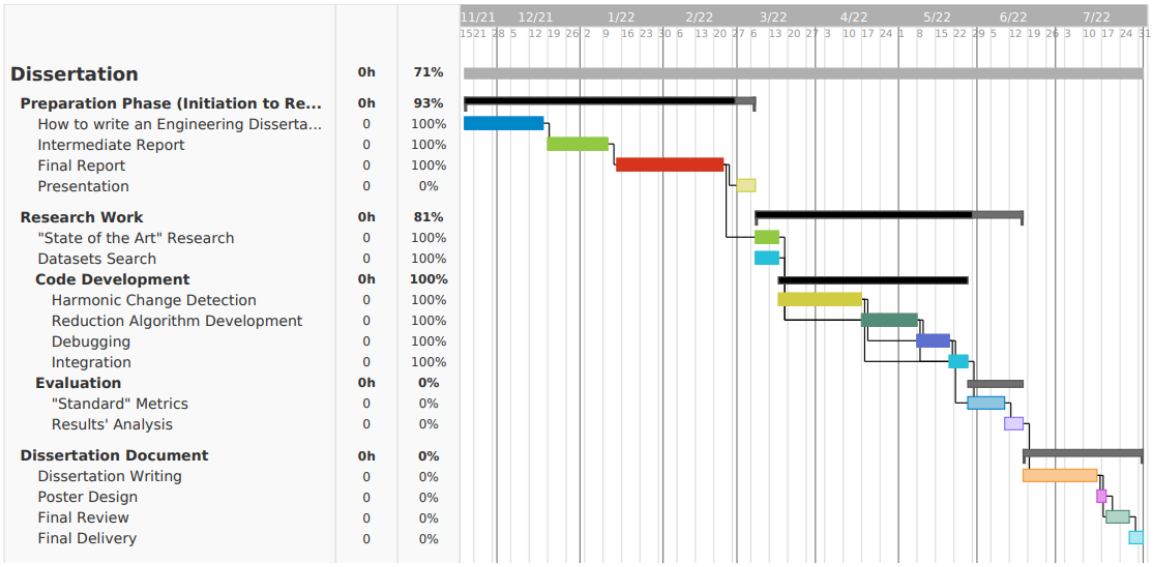

Figure 1.1: Initial Workplan

.

Its various groups and subgroups will be explained in more detail in the following paragraphs:

- **Preparation Phase**: Comprehends the period related to the course "Initiation to Research". In this course there was a seminar (How to Write an Engineering Dissertation), where the rules and precautions to be taken in writing a thesis, as well as to undertake a literature review for relevant information were addressed. In addition, two reports were written on the thesis proposal: an intermediate and a final one, the latter being the object of a presentation;
- **Research Phase**: It refers to the period of efficient search for relevant information to the writing of the dissertation, from the initial search and understanding of the state of the art to the creation of an effective algorithm that positively supports all the stages defined in the methodology.

In the case of this dissertation, the "Code Development" part requires some important notes:

- **Harmonic Change Detection**: It is the main task of this thesis. It will pick up the proposed variation of Harte's HCDF [5] by Ramoneda and Bernardes [2] and, based on the latter, promote a new one (in this case for symbolic music representations);
- **Reduction Algorithm Development**: TIS will take a pivotal role, assuming the task of reducing the symbolic surface of the music used;
- **Evaluation**: According to the standard metrics, the comparison of the obtained results and the datasets results will form a global opinion about the developed work and its results, as well as a thought for future improvements.

The last part refers to the actual writing of the thesis, as well as certain subsequent actions (e.g. Poster Design), only being subject to scrutiny in the final phase of its writing.

1.7 Structure of the Dissertation

This dissertation is organized as follows:

- This initial Chapter introduces the subject of the dissertation and presents the methodology employed to carry out the study. It ends with a graphical depict of all stages defined in advance to establish a tentative workplan;
- Chapter 2 renders a closer look at the work previously done in the field of study of harmonic change detection. This "State-of-the-art" provides a backbone for all the background on which this thesis is based on;
- Chapter 3 provides a detailed illustration of each step of the direction followed according to its primary objective;

- Chapter 4 brings to the fore a critical evaluation of the obtained results, as well as possible explanations for unexpected behaviours;
- Finally, Chapter 5 renders the main conclusions, making a constructive assessment of all the work developed and the results obtained, and suggesting possible future changes or additions that can help to improve the constructed method in future endeavours.

Chapter 2

Harmonic Change Detection: An Extensive Review

2.0.1 Fundamentals of Harmony

2.0.1.1 Harmony

The most basic and central concept of this study is **harmony**, which is defined as the simultaneous combination of notes, which leads to the formation of chords, and, in consequence, to the production of chord progressions [14]. Harmony and its computational processing has been a topic of interest in computer music, and largely addressed by the sound and music computing community.

2.0.1.2 Intervals

An interval can be described as a pitch difference (in semitones) between two notes [15]. Table 2.1 shows the nomenclature adopted in Western music traditions for all intervals within an octave.

2.0.1.3 Chord

In Western Tonal Harmony, a **chord** is defined as a group of two or more notes sounded together, with each existing type named according to the intervals formed by the constituent notes [16].

The most frequently encountered type of chord is a **triad**, which is designed as a set of three notes which are arranged to form two thirds [17]. There are two basic types of triads: **major**, if the lower third is major and the upper is minor (e.g. C-E-G), and **minor**, if the aforementioned sequence is reversed (lower 3rd minor and upper third major, e.g. A-C-E). Two variations can arise from these definitions: **augmented**, if the chord is composed by two major thirds (e.g. C-E#-G), and **diminished**, if the chord is composed by two minor thirds (e.g. A-C-C♭) [17]. There are also **tetrads**, which are basically "four note-chords", or, in order to relate the two definitions, they are triads with an additional note: this addition promotes two new chord types: **Major Seventh** (e.g. C-E-G-B), and **Minor Seventh** (A-C-E-G). But these are not the only existing types of chords: its

Number of Semitones	Interval
0	Unison
1	Minor Second/Augmented Unison
2	Major Second/Diminished Third
3	Minor Third/Augmented Second
4	Major Third/Diminished Fourth
5	Perfect Fourth/Augmented Third
6	Diminished Fifth/Augmented Fourth
7	Perfect Fifth/Diminished Sixth
8	Minor Sixth/Augmented Fifth
9	Major Sixth/Diminished Seventh
10	Minor Seventh/Augmented Sixth
11	Major Seventh/Diminished Octave
12	Perfect Octave/Augmented Seventh

Table 2.1: Relationships between Number of Semitones and its corresponding Interval Name

.

type depends on the intervals between the notes. The variety of existing harmonies is multiplied with the addition of notes in the chord.

The computational description of a chord is typically made with Roman Numeral (RN) analysis, which represents chords as RNs to show the relation between the root note of the chord and its tonic: that relationship forms the functional harmony [18].

2.0.1.4 Tonality

Since this study revolves around Western tonal harmony, it is fundamental to understand the notion of tonality. Tonality is defined as an organization of pitch relationships which helps creating a musical articulation in time [19]. Its two main components are **melody** (sequential pitch structures) and **harmony** (synchronous pitch structures) and its intervals are considered to be two important perceptual tonal descriptors in the Tonal Pitch Space.

2.0.2 Symbolic Digital Music Notation

Symbolic digital music manifestations, such as scores, ABC notation, MIDI, and MusicXML files, have been recently growing in both public and private datasets. Given its nature, a greater interest in studying this descriptive form of music has been pointed out by musicologists and music theorists, who work primarily with musical representations. In this context, due to greater ease to obtain audio content rather than symbolic representations, the study of the latter is still quite scarce. However, recent studies show that it can produce results similar to those obtained with audio [3].

2.1 Tonal Pitch Spaces

Tonal pitch spaces are algebraic or geometric representations of a pitch, where distances between its relations are perceived with a proximity among pitches, chords or keys [20]. One of the earliest pitch spaces, *Tonnetz*, was proposed by Euler [4]. Euler proposed an equal-tempered pitch organization based on interval relations, especially perfect fifths and major and minor thirds.

Figure 2.1: Excerpt of Original Tonnetz Graphic by Euler [4]

.

Figure 2.1 shows a part of the Tonnetz diagram proposed by Euler [4]. It is possible to notice some formed relationships between pitches, with the two coloured triads (one major and one minor), along with the possible relations formed in each direction (Fifths are formed horizontally from left to right, minor thirds are formed diagonally from bottom left to top right, and major thirds are also formed diagonally from top left to bottom right). A more extensive diagram will cover more relationships between the various pitches.

The Tonnetz was the inspiration for many contributions on this domain [21][5][1]. They are considerably more complex, since they encompass, in their description, the modeling of more and more harmonic hierarchies:

- **Chew Spiral Array**: Based on [22], is basically the Euler's *Tonnetz* representation, but in a 2-dimensional helix, with pitches being mapped into a continuous spiral, which allows the modelling of abstract concepts (e.g. chords and keys) and the construction of spatial relations between pitches, chords and keys;
- **Tonal Centroid Space**: Tonal Centroid Space, proposed by Harte [5], introduced **Enharmonic Equivalence** (Chew Spiral Array model assumed all possible notes and its notations (e.g. assumes that **F*b*** is not the same that **E**) so, to perpetuate this behaviour, it did not assume enharmonic equivalence) and **Octave Equivalence**, reducing all possible notes to a set of 12 pitch classes, categorized into a "6-D Hypertorus" space: each chord can be described by its respective 6-D centroid present in the space;
- **Tonal Interval Space**: Proposed by Bernardes [1] and inspired in the aforementioned hypotheses, its characterization allows significant improvements in relation to the previously mentioned base models: the characterization of the various possible relationships between pitches makes an improvement relative to Euler's base model, which only characterized major and minor triads; the existence of Chroma Vectors and its function promotes a more easy representation than Spiral Array by Chew [21]. Lastly, by considering all pitch relationships in only one octave, TIV has an upgrade upon Harte's [5] method.

2.1.1 Tonal Interval Space

In this study, TIS, a 12-dimension pitch space, where each dimension corresponds to a harmonic quality and has some links to the interval content of a sonority, will be adopted. To map the pitch into the TIS, a pitch class profile vector (also referred to as a chroma vector, namely when driven from audio) is first aggregated, which is then projected into the space as Tonal Interval Vectors (TIV).

Figure 2.2: TIV - Sequence [1]

.

In audio processing, chroma vectors are 12-element vectors, with each dimension referring to a pitch class, that indicates how much of the energy of each of the 12 distinct pitch classes of the octave is actually present in the signal [23]. For symbolic music, chroma vector refers to the cardinality of each pitch class.

In this matter, it is also worthwhile to refer Pitch Class Vectors (PCV), remmant of Pitch Class Profiles [24] [25]: the definition is pretty much similar, but the profiles are only binary or integers.

Chroma Vector Position	0	1	2	3	4	5	6	7	8	9	10	11
Pitch Class	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
Chroma Vector Value	1	0	0	0	1	0	0	1	0	0	0	0

Table 2.2: Example of a Chroma vector $c(n)$ representation (C major chord)

.

For each chroma vector, $c(n)$, a TIV, $T(k)$, is computed, as its normalized and weighted discrete Fourier transform (DFT), such that:

$$T(k) = w(k) \sum_{n=0}^{N-1} \overline{c}(n) e^{-\frac{j2\pi kn}{N}}, k = (0,...,6) \tag{2.1}$$

$$\overline{c}(n) = \frac{c(n)}{\sum n = 0^{N-1} c(n)} \tag{2.2}$$

where $\overline{c}(n)$ is the normalized chroma vector, $N = 12$ is the dimension of the chroma vector and $w_s(k)$ = [2, 11, 17, 16, 19, 7] is a set of weights constructed from empirical consonance ratings using dyads.

The final result is a 12-dimension TIV $T(k)$, where each dimension (or coefficient) has an interpretation as a harmonic quality and a weaker relation to musical intervals. It is important to note that, as the last six dimensions will be symmetric to the first six, only these first six are worth exploring, as referred in 2.1.

With TIV, all the major and minor chords have the same magnitude (i.e. distance to the centre of TIS), since the interval structure is the same, but they have all different angles), which can create the assumption that they all have the same interval content. Moreover, the phase gives information about the pitch class content of the chord.

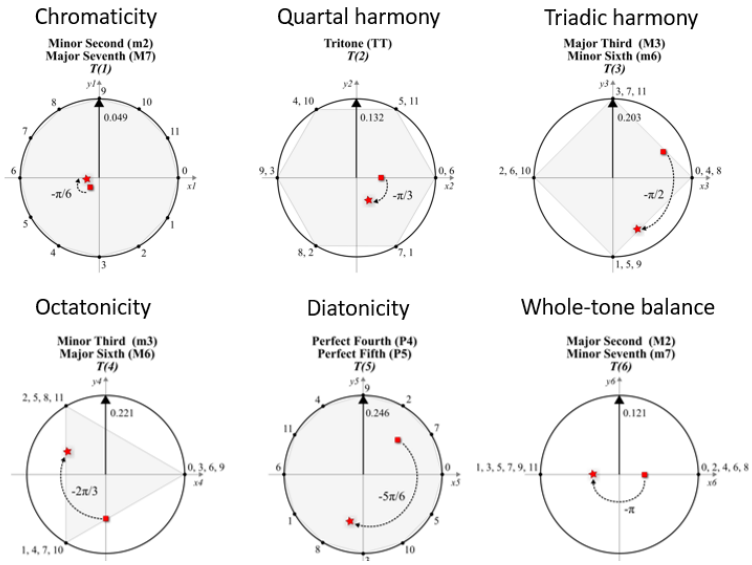

Figure 2.3: C chord looked through the first 6 dimensions of the Tonal Interval Space [1]

.

TIV is the basis of reduction processes inscribed in certain methods for various types of themes: e.g. `MusikVerb` [26], a novel digital reverberation which, using a spectral filtering technique, 'tunes' the reverberation output, according to the harmonic content of the input; and `Sostenante Pedal` [27], a sustain pedal used for digital pianos, where certain notes of the chords, according to the relation of their positions in the TIS, are released, resulting in a more consonant and less dissonant final chord.

2.2 Harmonic Change Detection Methods

The Harmonic Change Detection Function (HCDF), originally proposed by Christopher A. Harte in [5], has widely been considered the base of the "train of thought" that must be implemented in the study of this theme, given to be successful in detect harmonic changes such as chord boundaries in polyphonic audio recordings. HCDF involves a model that creates a space formed by equal tempered Pitch Classes, mapping 12-element Chroma Vectors to the interior of a 6-D polytope visualized as three circles, mapping also 6-D Centroid Vectors to represent the existing harmonic relations. To measure a possible harmonic change occurred in a certain frame n, a calculated euclidean distance between frames $n-1$ and $n+1$ is needed. Its results can render various conclusions (e.g. a small result of a euclidean distance denotes closer harmonic relations). A peak occurred in the detection function shows a transition from a harmonically stable region to another.

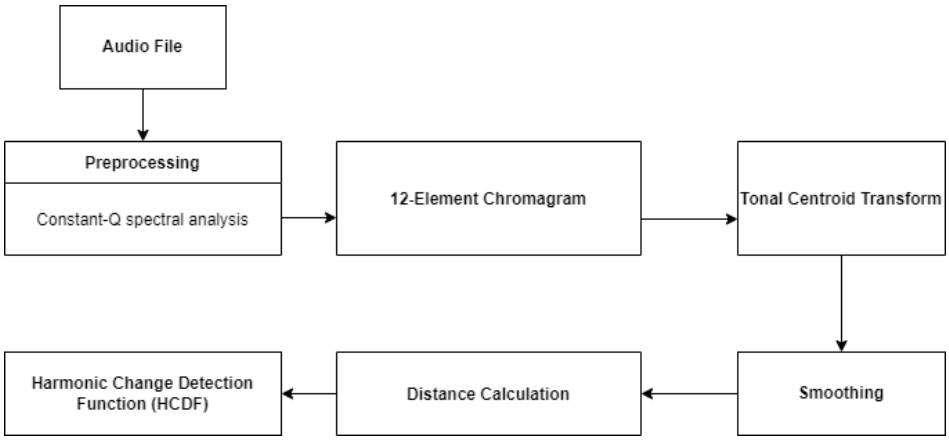

Figure 2.4: Original HCDF Sequence, proposed by Harte [5]

.

In this dissertation, it will be employed the same method as in [2]. Although using symbolic music representations instead of audio as input, it will be created an improved way of computing HCDF, aiming at detecting harmonic transitions in musical signals. This is a vital condition for the ACR task, as a revision (and betterment) of each component block of HCDF, relying on more recent advances, promotes a more efficient and adapted HCDF when compared to the original [5].

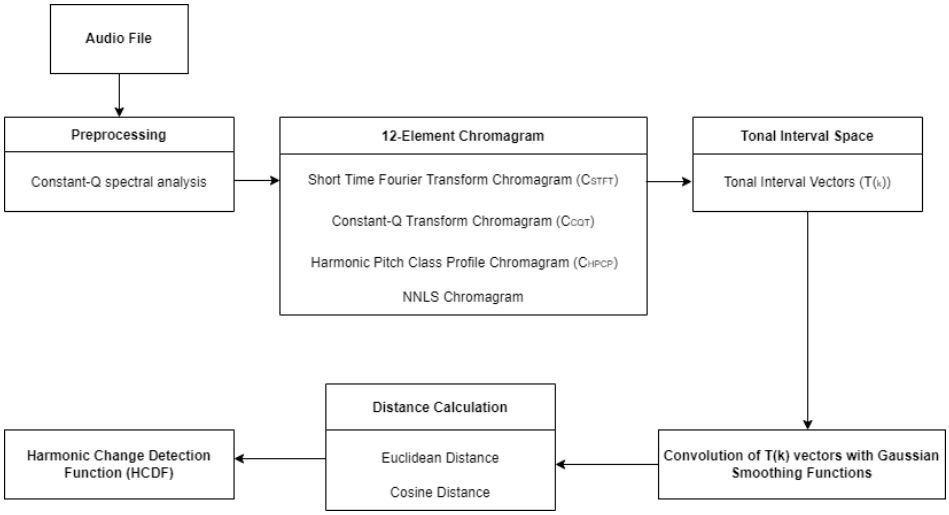

Figure 2.5: A changed HCDF sequence [2]

.

As referred earlier, the aforementioned changes and inclusions are the result of a "block-by-block" revision of the original HCDF methodology, with each block retaining the most efficient algorithm (based in the recent literature), creating the most effective HCDF algorithm possible, as follows:

- **Preprocessing**: Creates a frequency-based representation of the digital input waveform (audio). To that end, and considering future chromagrams computation requirements, Constant-Q spectral analysis is made;

- **12-Element Chromagram**: To create a 12-element chroma vector, compressing the sum of all the energy of each pitch class (of all the 12 notes), various algorithms have been put to test: from the earliest methods (e.g. **Short Time Fourier Transform Chromagram** (C_{STFT}) [28], **Constant-Q Transform Chromagram** (C_{CQTC}) [29], **Pitch Class Profiles** (PCP) [25]), to some more advanced ones (e.g. **Harmonic Pitch Classes Profiles** (HPCP) [24] and **NNLS Chromagrams** (C_{NNLS}) [30]). To this day, NNLS seems the best way to create the most balanced chroma vector.

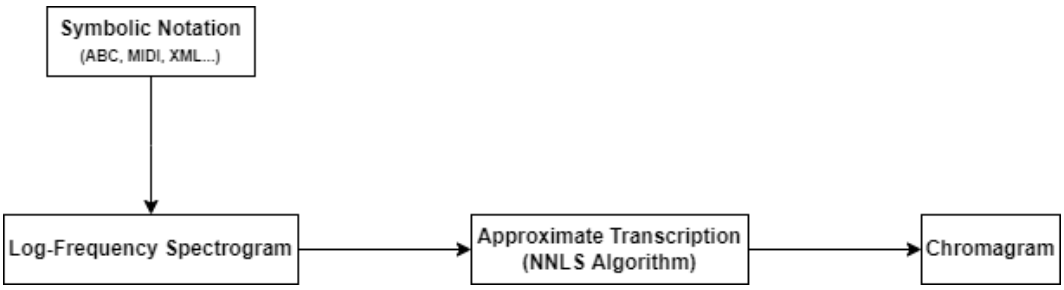

Figure 2.6: NNLS Chromagram (C_{NNLS}) Pipeline Diagram

.

A descriptive diagram of the creation of NNLS Chromagrams is presented in figure 2.6. In it, the two main steps for the realization of this chroma are visible: **Log-Frequency Spectrogram**, in which is computed the spectrogram that indicates the distribution of the signal's energy across the various signal's pitches, and **Approximate Transcription**, a tentative transcription of the information contained on the spectrogram is computed in a new spectrum, with this to be inferred using a Non-Negative Least Squares (NNLS) algorithm. At last, two types of chromagram are created: a treble chroma for higher frequencies, and a bass chromagram in lower frequencies (a third one can also be computed, being this a mixture of both);

- **Tonal Interval Space**: The great addition to this revamped algorithm is the Tonal Interval Space [1]. Exhaustively described in the "Background" section, it provides a good solution for the creation of TIV, a 12-element vector formed from each chroma vector relative to each of the 12 existing notes. The corresponding space will especially represent the most important parameters of Western Tonal Music: Pitch, Chord and Key will be included in unique and notable locations in the space;

- **Smoothing and Distance Calculation**:

  - A convolution between TIV ($T(k)$) with a Gaussian smoothing function is constructed, in order to minimize the effect caused by noise and transient frames;

  - To take some conclusions about the relationships created between notes, "distances" between $T(k)$ vectors are calculated: the Euclidean Distance, to state movements between two sounds, and Cosine Distances, to evaluate a supposed "mix" of the aforementioned sounds;

- **Harmonic Change Detection Function (HCDF)**:
  - Given the smoothed $T(k)$ vectors resulting from the *Smoothing* step, HCDF retreats an overall rate of change between consecutive frames (frames $n-1$ and $n+1$), with peak picking applied in the end to highlight harmonic stable regions [31]. The code used for TIV computation has been provided by [1];

According to [2], the aforementioned improvement results in a 5.57% (*F-Score* evaluation metric) progress of the ability to detect harmonic changes, when compared to other methods. Although there is an improvement in results when compared to previous methods, an amendment from a type of audio input to symbolic music takes a significant change in the entire process, from data processing to the creation of TIV's.

An important ally is [3], which promotes harmonic analysis with symbolic music and its possibilities of use in the Automatic Chord Recognition (ACR) task.

In this dissertation, two recent models related to ACR are portrayed: Bi-directional Transformer for Chord Recognition (BTC) [32] and the Harmony Transformer (HT), with this dissertation comparing both, namely in chord symbol recognition and functional harmony recognition quality. Both methods are built upon the Transformer model [33] and its two main computation blocks (Multi-Head Attention (MHA), used to create global dependencies between inputs and outputs, and the Feed-forward Networks (FFN)). Both methods share great ability to chord segmentation. These revamped versions demonstrate the capacity of improvement that the "multi-head attention" mechanism introduces in the performance of ACR, as described below:

- **Bi-directional Transformer for Chord Recognition (BTC)**: Originally created just for audio file datasets, it only uses the encoder part of Transformer and aims to capture harmonic dependencies in the chord sequences provided;
- **Harmony Transformer (HT)**: Created for both audio and symbolic music datasets, it uses the full extent of Transformer's architecture and aims to estimate chord boundaries, and then, according to the information present in the segmented chord, recognise the chord.

The introduction of intra/inter-MHA's (according to the type of input of the *Attention* function), uni/bi-directional MHA's (according to which positions of a sequence are valid to be applied the attention function) and FFN's blocks provide several adaptions from the original methods.

2.2.1 Reduction-based Musical Structures

A part of this thesis' main goal is to reduce the complexity of the structure of symbolic music representation. To achieve that, musical structures based on reduction are used.

Various forms of reducing musical structures have been proposed: the most known, and considered a classic in the field, is Generative Theory of Tonal Music (GTTM), proposed by Lerdahl and Jackendoff [34], where four hierarchically reduced structures (Grouping Structure, Metrical

Structure, Time-span Reduction and Prolongational Reduction), are presented as possible ways of modeling a person's musical intuition. Dominant regions contain a smaller number of subordinate elements and equal elements can exist contiguously within a particular hierarchical level [34]. Largely based on GTTM, is other reduction-based method proposed by Nakamura [35]. It revolves around a probabilistic formulation of music language, forming a tree-representation model, probabilistic context-free grammar, based on time-span tree proposed in GTTM. More recently, other methods, based on GTTM, have been proposed ([36] [37]), with the second reference being an updated version of the first. Both follow a very similar path:

- At first, each chord of the musical sequence is segmented;
- Once the segmentation is performed, the most relevant note of each chord is identified.

Carrying out this process on all elements of the input sequence, this whole path is summarized in a tree structure, containing all the aforementioned reduction levels.

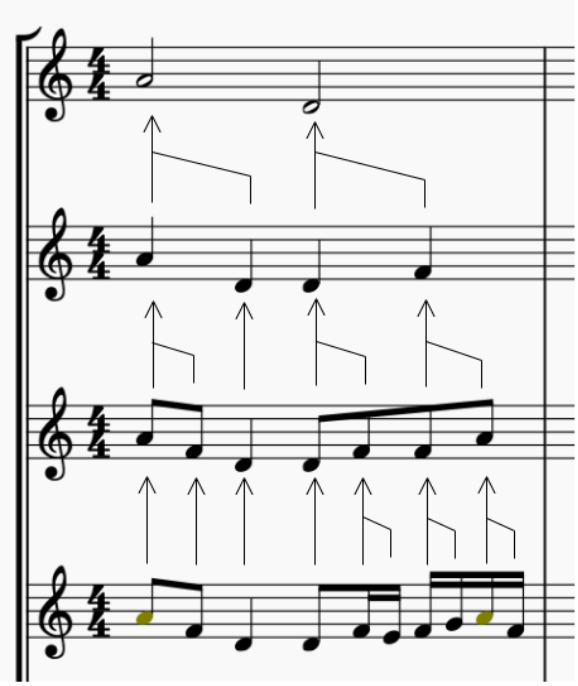

Figure 2.7: An Exemplified Reduction of a Structured Musical Sequence

.

In this dissertation it will be also developed a tree-based representation, where the notes of the analysed piece of music will be ranked into various levels of importance, between the original sequence (i.e., leaf notes) and the root, in a form of a tree (hence the term *"tree-based representation"*: the lowest level being the whole chord and whole harmony to be scrutinized and the highest level of reduction being only the most noticeable note of the chord. The resulting tree will be the input to the harmonic change detection task, where the boundaries of the chords will be deducted. This process will be implemented using TIS and its reduction characteristics.

2.2.2 Evaluation Metrics

The evaluation metrics used in this dissertation are considered *standard* evaluation metrics [7] generally used to evaluate Music Information Retrieval Evaluation eXchange `MIREX` Tasks and researches released in Transactions of the International Society for Music Information Retrieval `TISMIR` to indicate performances measures of the algorithm:

- **Precision** (p): Percentage of correct hits to detected changes in the harmony of the chord:

$$p = \frac{tp}{tp + fp} \cdot 100(\%)$$

- **Recall** (r): Percentage of correct hits to harmonic changes according to ground truth annotations:

$$p = \frac{tp}{tp + fn} \cdot 100(\%)$$

- ***F-Score***: The most usual evaluation metric, represents an harmonic mean value of the Precision and Recall indicators:

$$p = \frac{2 * p * r}{p + r}$$

Note that *tp*, *fp* and *fn* are acronyms for true positives, false positives and false negatives (respectively). A true positive represents a correct hit of a harmonic change by the HDCF function; a false positive is a correct hit in the HCDF function, but is not a harmonic change in the symbolic representation. At last, a false negative is a hit that has not been identified by the HDCF function as a harmonic change, although there's actually an harmonic change in the representation.

In [3], the segmentation quality (SQ) is calculated using Directional Hamming Distance (DHD) [30]:

$$SQ = 1 - MAX[DHD(\mathbf{S}, \mathbf{S'}), DHD(\mathbf{S'}, \mathbf{S})] \tag{2.3}$$

$$DHD(\mathbf{S}, \mathbf{S'}) = \frac{\sum_{i=1}^{\text{size}(N)} (|\mathbf{S}_n| - MAX_n |\mathbf{S}_n \cap \mathbf{S'}_n|)}{\sum_{i=1}^{\text{size}(N)} |\mathbf{S}_n|} \tag{2.4}$$

S_n refers to the frames of the n^{th} segment of the annotated segmentation $\mathbf{S}$ and $S'n$ denotes the frames of the n^{th} segment of the predicted segmentation $\mathbf{S'}$. The SQ value (which goes from 0 to 1 only) reflects the similarity of two segmentations: the higher the SQ coefficient, the higher the quality of the segmentation produced: a null value represents an uneven segmentation between the two parts, without any kind of correlation of parts between them; a unit value indicates complete equality between them: they mean the same representation.

Chapter 3

Symbolic Harmonic Change Detection in the Tonal Interval Space

This Chapter defines the proposed Harmonic Change Detection Function (HCDF) for symbolic music manifestations. The reasoning behind the multiple components of the HCDF are detailed along with the computational methods adopted.

3.1 Harmonic Change Detection Function

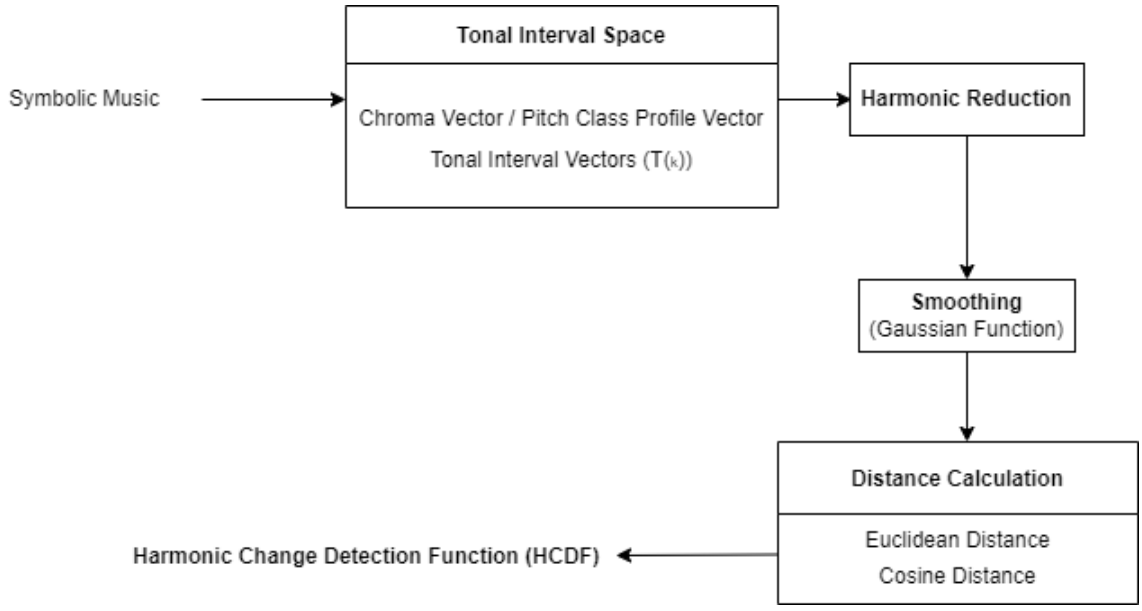

Figure 3.1: Diagram Sequence of the various steps involved on the HCDF process

.

Figure 3.1 shows the pipeline to compute the HCDF for symbolic music manifestations. It features four prominent modules that will be detailed in the following sections. In the resulting function, peaks indicate the changes in the harmonic content of the symbolic music input, i.e.,

the onset of chords. A novelty to the pipeline introduced in Ramoneda and Bernardes [2] is the adoption of a preprocessing harmonic reduction module, where the component notes of chords are inspected and potentially filtered to avoid ornament and embellishment notes which are not fundamental to the harmonic structure and typically introduce false positive detection in the HCDF.

A GitHub repository containing the Python scripts for this work was created, to save all development steps that ended up morphing into the final code [1].

3.1.1 Symbolic Input

The basic format used to describe symbolic music was MIDI, for its handling in storing onset times and notes duration, fundamental for the later comparison with the groundtruth values of each chord annotations, generally contained on CSV or XLSX extensions. For this matter, all files from the considered datasets were converted, from a machine-readable format (MuseScore format), to MIDI file [2]. Another reason for the choice of this format is its easy interconnection with certain Python libraries, such as `music21` [9], a very important library in the MIR area, and `TIVlib` [1] [8], necessary for this work.

For each MIDI file, a number of relevant descriptors for the non-reduction and reduction processes are used, as the following information can detail:

- Onset Tick - Starting time of each note;
- Midi Pitch - Integer number that represents a note's pitch;
- Offset - Starting time of each note (with quarter note as unit and the first note has 0);
- Duration - Duration of each note, (with quarter note as unit).

3.1.2 Tonal Interval Vectors

The harmonic content of a symbolic music manifestation is represented by Tonal Interval Vectors, $T(k)$, computed as the discrete Fourier transform of chroma vector or pitch class profile vectors. The resulting vector is a 12-dimension vector (2.1).

Chroma or pitch class vectors play an important role in music processing, as they represent the energy (in audio) or the cardinality (in symbolic) of each pitch class.

For this work, chroma vectors (or harmonic pitch class profiles [25]) were created by the *get.chroma* function of the `pretty_midi` library to each MIDI file, with its final chromagram, i.e., a temporal sequence of chroma vectors in time, being segmentated in columns spaced apart by $1/fs$ seconds (in this case, fs is the resolution value). Figures 3.2 and 3.3 show two possible chromagram representations for the Beethoven's 19th Sonata: one using `pretty_midi` [41] and `librosa` libraries, and the other using `libfmp` library [6].

[1] The GitHub repository can be accessed in https://github.com/ZMacedo/HCDF-Symbolic_Music.git

[2] All datasets can be found as submodules of the AugmentedNet repository, which can be accessed in https://github.com/napulen/AugmentedNet.git

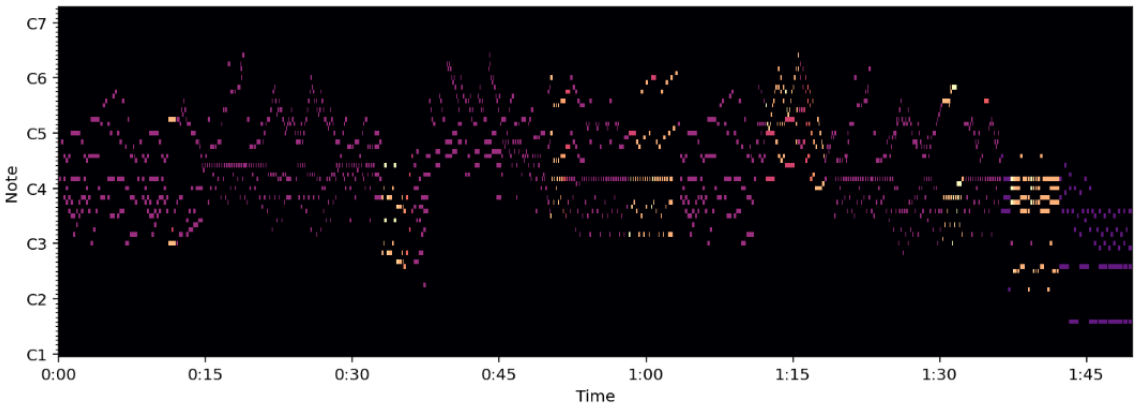

Figure 3.2: Chromagram of Beethoven's 19th Sonata by `pretty_midi` and `librosa` libraries

.

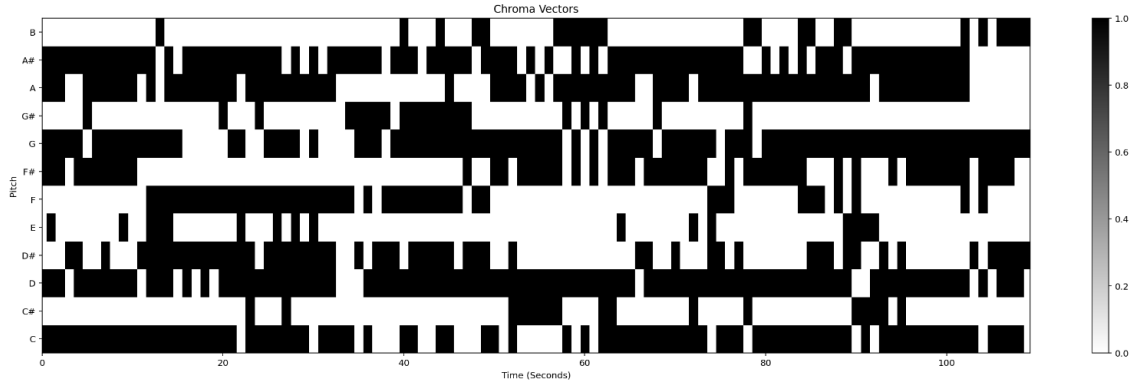

Figure 3.3: Chromagram of Beethoven's 19th Sonata by `libfmp` library [6]

.

After applying the DFT, the resulting Tonal Interval Vector, $T(k)$, exist in a space, where distances equate with the the cognitive distance of the pitch configurations [1]. $T(k)$ vectors can represent any pitch configuration from a single note to chords with varying number of notes and scales. In greater detail, pitch configurations that are close in space are understood by humans as related, such as the interval of fifth. Conversely, a minor second will have a larger distance, as a result of commonly being harmonically unrelated within the tonal music context.

For each chroma vector (or pitch class profile vector), a 12-dimensional TIV can be computed using the **from_pcp** function of the `TIVlib` library (using the base parameters). The final TIV can also be plotted with **plot_TIV** function.

3.1.3 Smoothing

Tonal Interval Vectors, $T(K)$, are convoluted with a Gaussian filter to avoid possible transient frames, at later peak picking stages in the system's pipeline, a process named **Smoothing**. The degree of convolution will be measured from a sigma parameter, σ. The impact of high-convoluted *T(k)* vectors (i.e. convolution made with major σ values) is to provide a blurred convolution,

whereas with null or minor σ it can be denoted a more clear convolution operation. This step is constructed using the Gaussian filter function by the **scipy.ndimage** package of the `scipy` library [10].

3.1.4 Distance Calculation

Equations 3.1 and 3.2 define the **Euclidean** and **cosine** distance metrics adopted in the HCDF calculation. For a given event in time n the HCDF value is defined by the distance between $T(k)_{n-1}$ and $T(k)_{n+1}$ vectors.

$$\zeta_n^{eucl} = \|\zeta'_{n+1} - \zeta'_{n-1}\| \tag{3.1}$$

$$\zeta_n^{cos} = \frac{<\zeta'_{n+1}, \zeta'_{n+1}>}{\|\zeta'_{n+1}\|\|\zeta'_{n+1}\|} \tag{3.2}$$

While the Euclidean distance accounts for both interval and pitch class content properties of the pitch configurations, cosine distance only accounts for the latter. In other words, while Euclidean distance accounts for similar intervals and common tones between two pitch configurations, the cosine distance only considers the latter.

3.1.5 HCDF

All above steps lead to the HCDF, from which we extract the chord changes in a symbolic music manifestation. Peaks, or local maxima, in the HCDF indicate the onset of chord changes. Figure 3.4 illustrates the detected chord changes (in green) of a HCDF (in blue).

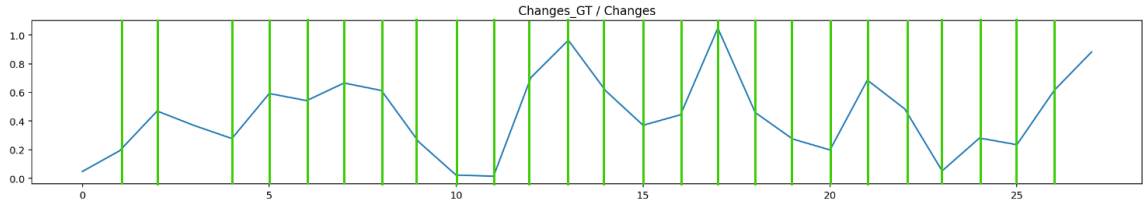

Figure 3.4: A Perfect HCDF Plot (with Bach Chorale example)

3.2 Harmonic Reduction

As identified in the literature, one of the main drawbacks of the HCDF is the large amount of false positive it tends to feature. In symbolic music manifestations, these false positive may be driven by ornaments or embellishments notes that typically occur in the musical surface. To account for latter and aiming to diminish their impact on the HCDF, we propose three harmonic reduction strategies: 1) tree-note chord, 2) threshold and reduction, and 3) distance threshold.

3.2.1 Tree-note Chord Reduction

This reduction strategy aims to transform any chord with four or more component notes to a triad, while retaining the root, third and fifth. To this end, the computational reduction method is fourfold: 1) identify the chords with 4 or more component notes; 2) compute the TIV ($T(k)$) of the chord; 3) compute the distance of each component note to the chord $T(k)$ vector; and 4) retain the 3 notes that have smaller distance.

A multi-part score can be represented in a single part, result of slicing the symbolic music score into "salami slices", i.e., temporal segments sliced vertically at every new onset in the musical score. Furthermore, the multi-part score is reduced to a single part score composed of a sequence of chords per slice. This transformation, using the **Chordify** function of the `music21` library, is illustrated in 3.5.

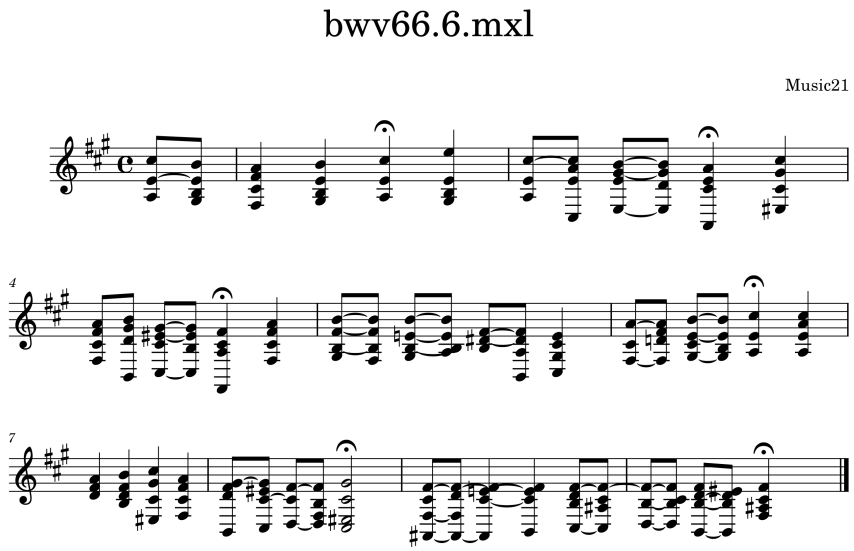

Figure 3.5: Example of a "Chordified" Score (Bach Chorale)

.

The reduction process is only applied to chords with four or more unique notes, therefore it ignores duplicated notes. After removing duplicate notes (as shown in Figure 3.6), the reduction steps can be applied. The output of this harmonic reduction process is a new score which can be parsed into a symbolic music notation format, such as MusicXML or MIDI. To verify the process, a score showing both filtered (red note-heads) and unfiltered (black note-heads) notes can be produced as shown in Figure 3.7.

In Figure 3.7, red note-heads indicate note to be removed from the harmonically reduced score. The method behaved as intended. For example, looking at the first chord that undergoes reduction, it consists of E, D, G#, B, which allows us to understand what forms a chord dominant seventh of E (E7 major). If the objective is to reduce the chord to its basic form (triad, in this case E major), the note to remove would be the most dissonant note, which promotes the dominant seventh nature of the chord (in this case, it would be the note D).

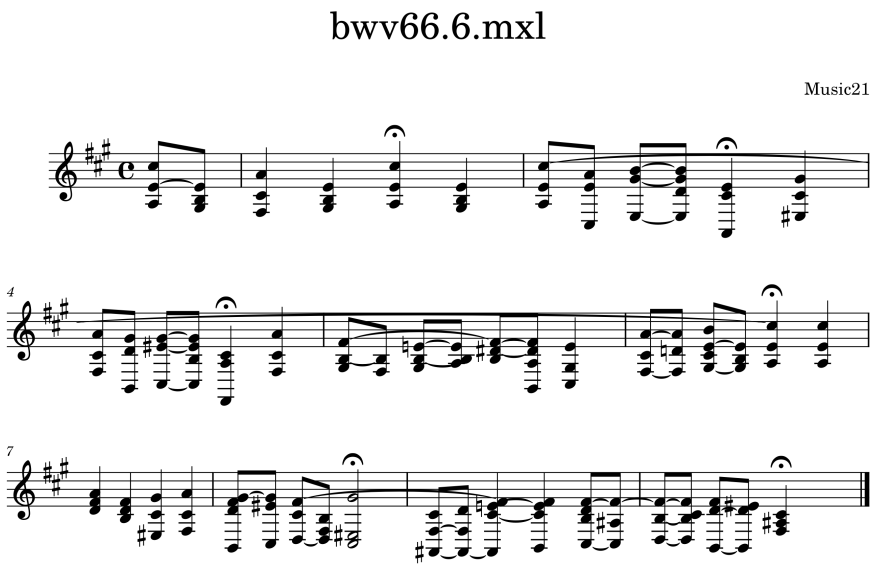

Figure 3.6: Bach Chorale with a Non-duplicated "Chordified" Score

.

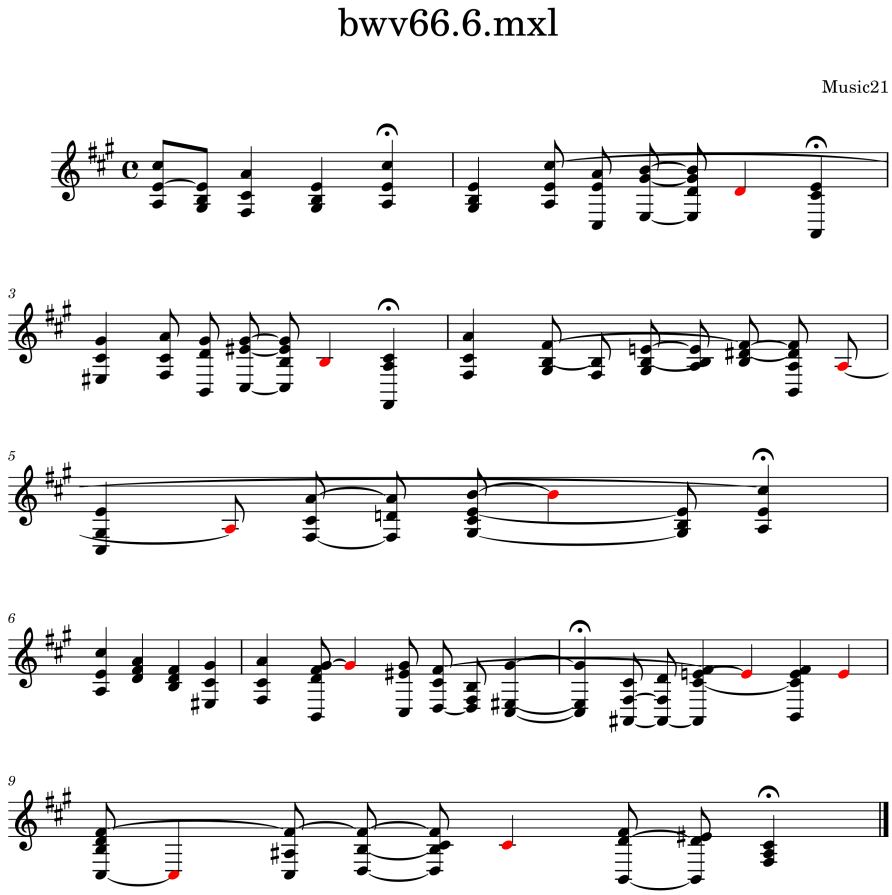

Figure 3.7: Non-duplicated "Chordified" Bach Chorale Score with Painted Removed Notes

.

Figure 3.8 shows the resulting score after the harmonic reduction process. The main problem with this method is that `music21` library does not provide a good conversion from a "chordified" score to MIDI. This conversion always calls *stripTies()* but, when this function is called, notes of the chord that are not strip-tied to the next chord can be lost, which leads to these notes being lost when converted to MIDI, in addition to the reduced notes themselves, which is not desirable.

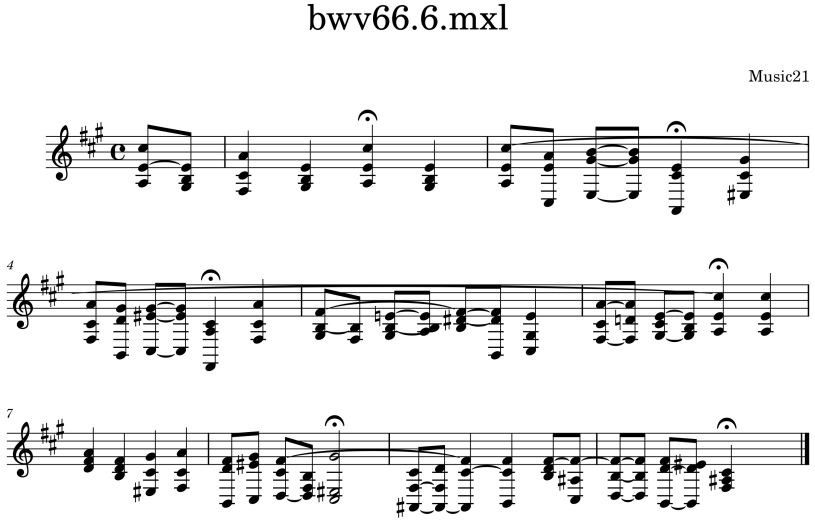

Figure 3.8: Final Reduced Non-duplicated "Chordified" Bach Chorale Score

.

3.2.2 Other Reduction Methods

In order to try new approaches and to expand the model presented to the reduction process, two more processes were created, with each one revolving around a threshold value:

3.2.2.1 "Threshold&Reduction"

Along the reduction method presented in the subsection before, a second reduction phase, based on a threshold value defined by the distance between each note and the centroid point of the TIV of the chord on its TIS representation, is added: above this value, all notes will be reduced, with an auxiliary reduction in case of having chords still not on its triad form. To that end, every MIDI file in each dataset will be reduced according to threshold values between 30 and 35, writing down the HCDF values for each case to find the best threshold value for the effect. It is expected that, with the increase of the threshold value, and the consequent decrease of the number of notes to reduce, the HCDF result will be closer to the results obtained without any kind of threshold.

3.2.2.2 Only Threshold

Another way would be just to look at the threshold values and only reduce notes that would be on a higher distance from the TIV center of the chord, not caring if the new chord would be a triad or not, but once again, the behaviour was the same as the other reduction processes.

Chapter 4

Evaluation

This Chapter presents the evaluation of the proposed HCDF for symbolic music manifestations in detecting chord changes.

The evaluation will be based on an objective and quantitative comparison between the onset times of ground truth annotation data (in seconds) of chord onsets. Namely, by assessing how well the predicted changes from the HCDF temporally align with the ground truth data. To quantify this alignment we adopt the standard evaluation metrics *F-Score*, *Precision* and *Recall*, already described in subsection 2.2.2. These are calculated using the `mir_eval` library [38], specifically its **mir_eval.onset.f_measure** function, which, from the **Precision** and **Recall** metrics, calculates the **F-score** value, in terms of correct and incorrectly predicted onsets. The correctness of this calculation is determined by a small window (since this work is being done in seconds, the value used for the window is 0.628).

4.1 Datasets

To test the developed model, four datasets will be adopted: `BPS-FH Dataset` (composed by symbolic music data and functional harmony annotations of the 1st movements from Beethoven's 32 piano sonatas), `Bach Preludes Dataset` (which consists of 24 preludes to Bach's *Well-Tempered Clavier* first book), `TAVERN` dataset [39] (formed by 17 works by Beethoven and 10 by Mozart for a total of 27 works and 281 Variations, 100 from Mozart and 181 from Beethoven), and `ABC` dataset [40], which consists of harmonic analysis of all Beethoven string quartets. To an easier extraction of information contained on the collected datasets, all symbolic data was converted to MIDI files, as previously explained in 3.1.1.

4.2 Evaluating HCDF

In this section a description is provided about the evaluation performed on the HCDF process for both non-reduction and reduction methods, as well as providing the results obtained and possible explanations for them.

4.2.1 HCDF Evaluation

To evaluate the HCDF function on symbolic musical data, a comparison of the time onsets of each chord change from the HCDF with the ground truth annotations was conducted. To this end, the parameterization of the following threefold HCDF variables was defined or assessed:

- Sigma - Parameter for the **Smoothing** process;
- Distance - Measure of the distance (considering Euclidean and Cosine metrics) between smoothed TIV vectors. Its overall value defines HCDF.

4.2.2 Results

To define the resolution of the MIDI files, the `pretty_midi` [41] library was used. This library creates a container for MIDI data in a format that allows changing its parameter values, such as resolution. For the evaluation, a resolution of 28 ticks per unit is adopted. The defined resolution is much lower than the initial resolution provided by the container (when no file or resolution value is provided, the container initialize with 220). The initial *tempo* of the file can also be controlled, although the value used for this parameter was the basic value given by the container (i.e. 120 BPM). Every score, before its conversion to MIDI, was altered to only have one single *tempo* throughout the whole piece (it was decided to maintain the 120 BPM basic value) for a correct read of the MIDI file by the **pretty_midi** library.

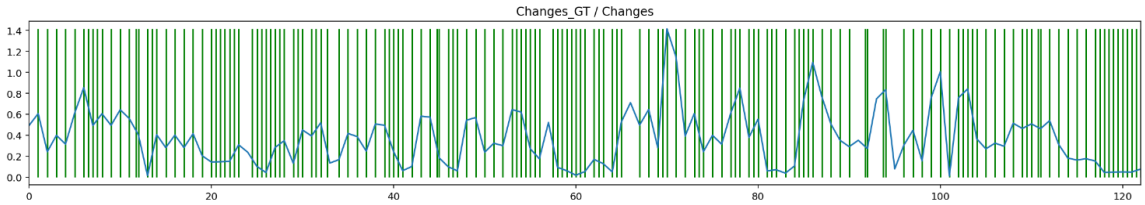

Figure 4.1: Beethoven's 19th Sonata's HCDF, with σ 10 and Euclidean Distance

.

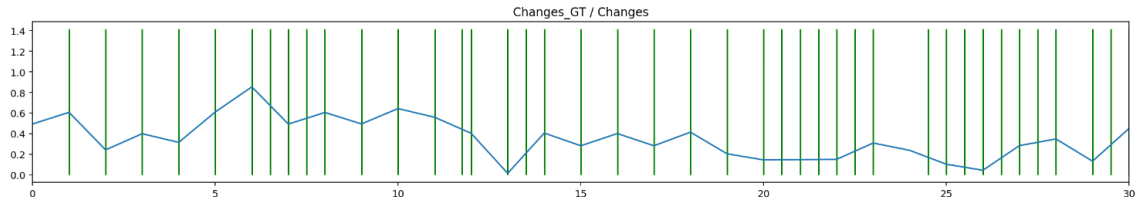

Figure 4.2: First 30 seconds of Beethoven 19th Sonata's HCDF

.

Figure 4.1 and 4.2 show HCDF plots from a symbolic music score. These figures show a clear difference in HCDF plots for audio and for symbolic music: for audio, it is expected to have angular peaks representing harmonic changes [2]; in symbolic music, as figures 4.1 and 4.2 can suggest, the peaks are sharp, almost as if the diagram consisted exclusively of line segments.

For a better understanding of the evolution of the evaluation metrics with the various possible values of σ, a graph is also computed that shows how the σ parameter performs, aiming to find the best σ value. Given that σ controls the Gaussian blur filter in the HCDF, it is expected to see a "inverted parable behavior" for *F-Score* and *Precision*, i.e., low values for low σ values, a substantial increase to median values and a decrease for major values, but a sharp and sustained growth for *Recall* values, behavior translated on the increase of σ, as can be seen in figures 4.3 and 4.4. Considering this behavior, we proceeded to search for the sigma value that promotes the best values for the mentioned metrics: as 4.3 and 4.4 suggest, the best σ value, and the one that would be adopted in the evaluation tests, is 10.

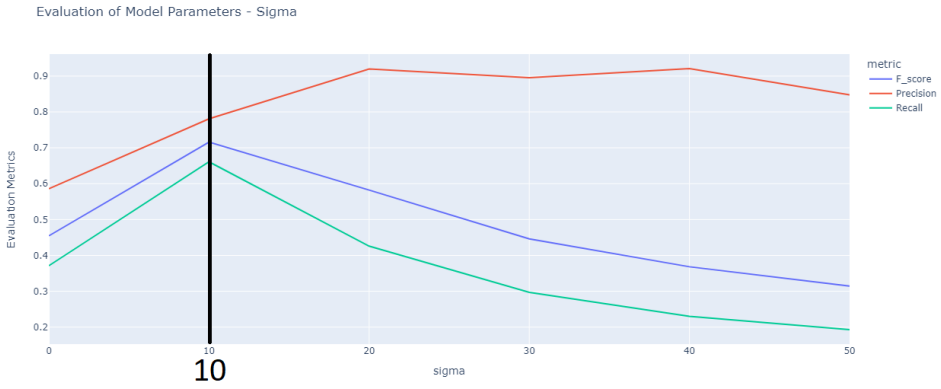

Figure 4.3: Evaluation Metrics Behaviour with σ increase (BPS-FH Dataset, Euclidean Distance)

.

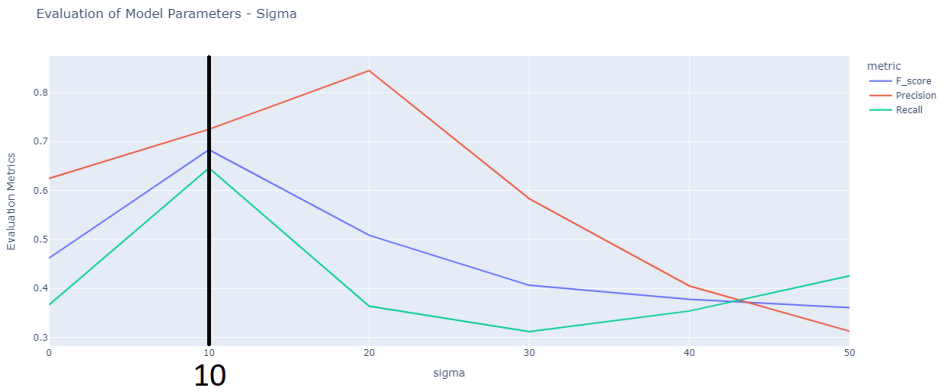

Figure 4.4: Evaluation Metrics Behaviour with σ increase (BPS-FH Dataset, Cosine Distance)

.

Tables 4.3 and 4.4 show that all non-reduction values stand in the 60-80% range, just like the values referred in the symbolic version of [2] and [3] (these values are represented in 4.1 and 4.2). Also worth of notice is the closeness of results between the Euclidean and cosine distances, denoting no difference between their adoption.

	F-Score	Precision	Recall
Haydn Op20 Dataset	67.65%	83.52%	59.08%

Table 4.1: Ramoneda's results for HCDF

.

BPS-FH					
Model	**Chord Symbol Recognition**		**Functional Harmony Recognition**		
	Accuracy	Segmentation	Key	Roman numeral	Segmentation
HT*	**83.98**$\pm_{1.08}$	**85.09**$\pm_{0.96}$	**79.07**$\pm_{2.70}$	**41.74**$\pm_{2.63}$	**75.50**$\pm_{1.72}$

Bach Preludes					
Model	**Chord Symbol Recognition**		**Functional Harmony Recognition**		
	Accuracy	Segmentation	Key	Roman numeral	Segmentation
HT*	**78.54**$\pm_{2.06}$	**83.86**$\pm_{2.24}$	**56.28**$\pm_{2.53}$	**25.95**$\pm_{1.67}$	**73.60**$\pm_{1.80}$

Table 4.2: Chen and Su's results for symbolic chord and functional harmony recognition

.

The results of the reduction process are inconclusive: in some cases they perform better, but it is mostly aligned with the results of the non-reduced scores, being slightly lower than these. In a way, it can be because of possible data being lost in the MIDI conversion in the *Chordify* function by `music21` library. In theory, when removing embellishment notes and dissonant notes, the performance of the HCDF would increase, but that is only true for the `Bach Preludes` Dataset, composed by simple scores (only two-part voices at maximum), with low chord reduction requirements.

Without Reduction Sigma=10 Distance = Euclidean	**Datasets** / **Metric**	**BPS-FH**	**ABC (Beethoven Quartets)**	**Bach Preludes**	**Tavern**			
					Beethoven		**Mozart**	
					A	**B**	**A**	**B**
	F-Score	62.8%	63.7%	72.6%	71.9%	71.8%	66.5%	66.5%
	Precision	62.4%	71.2%	82.8%	70.7%	70.6%	60.6%	61.1%
	Recall	67.6%	65.3%	70.3%	78.7%	78.9%	78.9%	78.3%

Without Reduction Sigma=10 Distance = Cosine	**Datasets** / **Metric**	**BPS-FH**	**ABC (Beethoven Quartets)**	**Bach Preludes**	**Tavern**			
					Beethoven		**Mozart**	
					A	**B**	**A**	**B**
	F-Score	60.3%	61.5%	70.4%	70.2%	69.8%	64.3%	64.8%
	Precision	60.0%	68.3%	81.0%	68.7%	68.3%	59.0%	59.8%
	Recall	65.2%	63.7%	67.6%	77.2%	76.9%	76.0%	76.0%

Table 4.3: Results without Reduction (for both distances)

.

To obtain the best σ value for these evaluations, as was done for HCDF without reduction, an observation of the progress of the evaluation metrics with various possible values of σ, for the tree-note chord reduction process, was also made, as it can be seen in figures 4.5 and 4.6. It can be concluded, from these figures, that the σ value that can provide higher results is also 10.

With Reduction Sigma = 10 Distance = Euclidean	Datasets / Metric	BPS-FH	ABC (Beethoven Quartets)	Bach Preludes	Tavern			
					Beethoven		Mozart	
					A	B	A	B
	F-Score	61.5%	63.2%	73.0%	70.3%	70.1%	66.4%	66.3%
	Precision	60.9%	70.9%	82.7%	69.3%	69.1%	61.0%	61.3%
	Recall	66.6%	64.3%	71.6%	76.7%	76.7%	78.6%	77.9%

With Reduction Sigma = 10 Distance = Cosine	Datasets / Metric	BPS-FH	ABC (Beethoven Quartets)	Bach Preludes	Tavern			
					Beethoven		Mozart	
					A	B	A	B
	F-Score	58.9%	61.0%	71.8%	68.3%	68.2%	64.3%	64.4%
	Precision	58.0%	67.8%	81.4%	66.1%	66.0%	58.6%	59.2%
	Recall	64.8%	62.9%	69.8%	76.1%	76.2%	76.8%	76.4%

Table 4.4: Results with Reduction (for both distances)

.

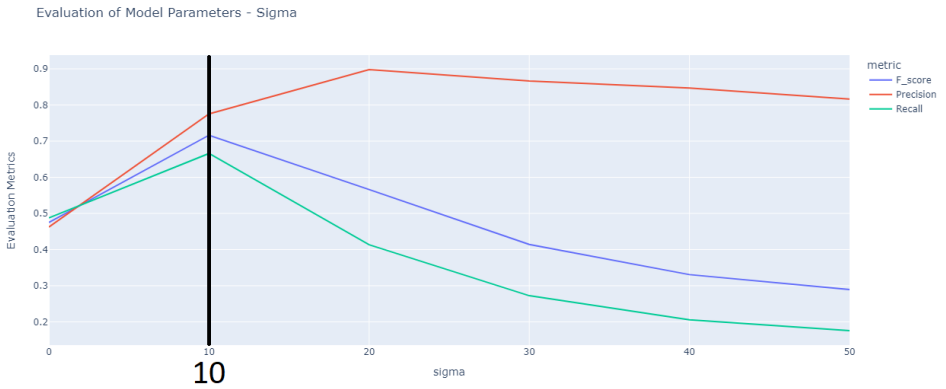

Figure 4.5: Evaluation Metrics Behaviour with σ increase (BPS-FH Reduced Dataset, Euclidean Distance)

.

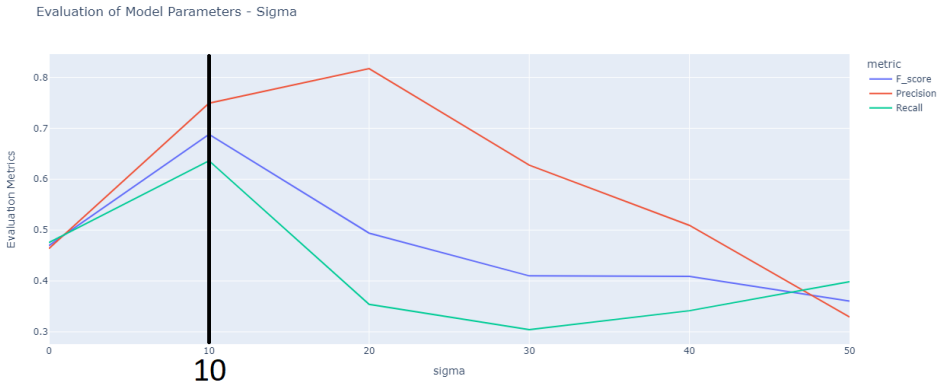

Figure 4.6: Evaluation Metrics Behaviour with σ increase (BPS-FH Reduced Dataset, Cosine Distance)

.

Finally, an important assessment in our evaluation is the comparison of the non-reduction methods with the different methods for harmonic reduction: the non-reduction method obtains better results than all three reduction programs, with the original reduction method being the best of them. Although, in theory, the presented reduction processes seem to be methods that may generate better results, one can not forget that, with TIV, the most dissonant notes will also be the most distant, being the first to be cut in the reduction process, so the "Threshold&Reduction" method will, in essence, give the same form of the original reduction previously made, so the results are similar in both cases: if a threshold value above 35 (the maximum value of the proposed interval of threshold values, as referred in 3.2.2.1) is used, since notes with a distance greater than this value to the TIV center of their respective chord are rare, so this method becomes a repetition of the initial reduction method. In relation to the case with only the threshold, the optimal threshold value, according to the obtained results for each dataset and using both methods for all possible threshold values, would be 33.

With Reduction Threshold=33 Sigma = 10 Distance = Euclidean	Datasets / Metric	BPS-FH	ABC (Beethoven Quartets)	Bach Preludes	Tavern			
					Beethoven		Mozart	
					A	B	A	B
	F-Score	61.5%	63.2%	73.0%	70.3%	70.1%	66.4%	66.3%
	Precision	60.9%	70.9%	82.7%	69.3%	69.1%	61.0%	61.1%
	Recall	66.6%	64.3%	71.6%	76.7%	76.7%	78.6%	77.9%

With Reduction Threshold=33 Sigma = 10 Distance = Cosine	Datasets / Metric	BPS-FH	ABC (Beethoven Quartets)	Bach Preludes	Tavern			
					Beethoven		Mozart	
					A	B	A	B
	F-Score	58.9%	61.0%	71.8%	68.3%	68.2%	64.3%	64.5%
	Precision	58.0%	67.7%	81.4%	66.0%	66.0%	58.6%	59.2%
	Recall	64.8%	62.9%	69.8%	76.1%	76.2%	76.8%	76.4%

Table 4.5: "Threshold&Reduction" results for all datasets (for both distances)

.

Threshold=33 Sigma=10 Distance = Euclidean	Datasets / Metric	BPS-FH	ABC (Beethoven Quartets)	Bach Preludes	Tavern			
					Beethoven		Mozart	
					A	B	A	B
	F-Score	61.7%	63.4%	73.4%	70.2%	70.0%	66.3%	66.1%
	Precision	61.0%	70.9%	82.9%	69.3%	69.0%	60.7%	61.1%
	Recall	66.9%	64.8%	71.9%	76.7%	76.7%	78.7%	77.8%

Threshold=33 Sigma=10 Distance = Cosine	Datasets / Metric	BPS-FH	ABC (Beethoven Quartets)	Bach Preludes	Tavern			
					Beethoven		Mozart	
					A	B	A	B
	F-Score	59.0%	61.1%	71.6%	68.4%	68.3%	64.5%	64.5%
	Precision	58.1%	67.6%	81.4%	66.2%	66.0%	58.6%	59.2%
	Recall	64.9%	63.3%	69.6%	76.6%	76.6%	77.0%	76.5%

Table 4.6: Maximum Threshold (without previous Tree-Note Chord reduction) for all datasets (for both distances)

.

The results obtained with the two types of representations mentioned are comparably similar, and, in certain cases, those involving symbolic music are better, which has allowed a greater use of these in studies in the area.

Finally, it can be concluded from the aforementioned lists of results and set of tables that the reduction methods generally do not improve the maximum values achieved for HCDF in previous works.

Chapter 5

Conclusions and Future Work

In this dissertation, the latest advances in harmonic description of symbolic music, namely the use of the Tonal Interval Vectors [1] were adopted to compute an HCDF for symbolic music representations. The proposed method detects harmonic changes in symbolic music manifestations, based on a critical revision on the HCDF model proposed by Ramoneda and Bernardes [2] and its parameterization. The proposed model and its parameterization were objectively evaluated using four datasets (BPS-FH, Bach Preludes, Tavern and ABC datasets) and *standard* F-score, precision and recall metrics.

The motivation to pursue this study is driven by the scarce research on symbolic music representations, when compared with musical audio signals.

At the core of this method lies Tonal Interval Space, where a 12-dimension Vector with each dimension containing the cardinality of each of the twelve pitch class [1]. From the Tonal Interval Vector, distances were calculated: the shorter the distance, the closer two notes were between each other. This line of thought formed the basis of the proposed model, namely the HCDF.

The main contribution of this dissertation is the addition of a new concept to this task: the harmonic reduction step, where the embellishment notes of the chord were removed. Three reduction models were proposed: one, more based on musical theory and based on the reduction of the chord to its basic harmonic form (triad), and the two others, more based on the properties of the chord's structure, which revolve around a threshold value (based on distances between notes on TIS), and which remove notes with distances above this base value.

All the objectives proposed in the introductory section of this dissertation were successfully achieved, with the creation and testing of methods that promote the detection of harmonic changes in the structure of each constituent chord of each symbolic music file, as well as the reduction of each chord either to its basic form (triad) or to a threshold value.

5.1 Main Results

The results of our evaluation on the proposed HCDF without harmonic reduction method are in line with those proposed in [2] and [3], with a close similarity between the values provided by both distances used (Euclidean and Cosine).

Regarding the harmonic reduction methods, they did not improve the performance of the HCDF in relation to the non-reduction harmonic method in most datasets, which was a non expected behaviour. However, it should be noted that the tree-note chord reduction method manages to capture the essential part of each chord, without changing its original harmonic structure, this being one of the main objectives to be achieved in the construction of the reduction process. For the other two methods, the results are similar, with the "Only Threshold" methods having the best performance values for `Bach Preludes`, results that are aligned with the ones obtained for the same dataset by Chen and Su [3], being the former a more transparent and computationally effective model.

One of the main purposes of this dissertation was to understand how the hierarchical reduction of the harmonic structure of a chord would improve the efficiency of the detection of harmonic changes. Emerging from the results shown earlier, it can be said that reduction, although only presenting better results on a small number of cases, can be a useful tool to obtain a more direct way to analyse and detect harmonic changes, as it provides a more concise chord with its original structure intact.

5.2 Future Work

As for the work that can be done in the future in the HCDF computation, some points need to be further explored: for this work, the only *tempo* of each piece is the same for all of its duration, 120 BPM, which forces the rewriting of the musical staff and the reduction of its various *tempos* to this single value. A future improvement would be promoting an efficient HCDF without these *tempo* limitations, in order to correctly point out the chord changes in the original, unchanged file, still containing all the different *tempos* and feels.

Another point where this work can be improved is to apply the method to music of different genre and style. Other datasets were to be included, like `Haydn20` [42], composed by String Quartets from Joseph Haydn but, due to tight schedule of the dissertation, it was not developed further to enter on the main options for this thesis, but it can be considered on future endeavours or in other future works. Most of the datasets used in Automatic Chord Recognition (ACR) are based on audio, a substantially more developed part than that of symbolic music, so the annotation of new datasets to symbolic music or the correct conversion of audio datasets to symbolic ones would also be a great improvement to future studies.

On a broader level, and noting the wide range of themes inserted on MIR field (e.g. ACR), and the number of new possible approaches (e.g Deep Learning was used in [3]), this work can also be applied to several of these, expanding this research.

Appendix A

HCDF Code (without Reduction)

In this appendix, the Python code for computing HCDF without reduction is presented:

```
import plotly.express as px
import sys
import numpy as np
import glob
import pretty_midi
import libfmp.b
import libfmp.c1
import libfmp.c3
import mir_eval
from TIVlib import TIV
import matplotlib.pyplot as plt
import pandas as pd
from scipy.spatial.distance import cosine, euclidean
from scipy.ndimage import gaussian_filter

# # TIS&TIV - Tonal Interval Space & Tonal Interval Vectors
# A truncated version of TIV library [1].
#[1] - Ramires, A., Bernardes, G., Davies, M.E., & Serra, X. (2020). TIV.lib: an
    open-source library for the tonal description of musical audio. ArXiv, abs
    /2008.11529.

#The full TIV library isn't importing correctly to the program, so here is a part
    of the TIV library.
class TIV:
    weights_symbolic = [2, 11, 17, 16, 19, 7]
    weights_audio = [3, 8, 11.5, 15, 14.5, 7.5]

    def __init__(self, energy, vector):
        self.energy = energy
        self.vector = vector

    def abs_vector(self):
        return np.abs(self.vector)
```

```
31
32      def phases_vector(self):
33          return np.angle(self.vector)
34
35      def get_vector(self):
36          return np.array(self.vector)
37
38      def diss(self): #Compute dissonance
39          return 1 - (np.linalg.norm(self.vector) / np.sqrt(np.sum(np.dot(self.
    weights_symbolic, self.weights_symbolic))))
40
41      def coeffs(self, coef): #Compute coefficient
42          return self.abs_vector()[coef] / self.weights_symbolic[coef]
43
44      def chromaticity(self): #Compute chromaticity
45          return self.abs_vector()[0] / self.weights_symbolic[0]
46
47      def dyads(self): #Compute dyadicity
48          return self.abs_vector()[1] / self.weights_symbolic[1]
49
50      def triads(self): #Compute triadicity (triads)
51          return self.abs_vector()[2] / self.weights_symbolic[2]
52
53      def d_q(self): #Refers a possible diminished quality
54          return self.abs_vector()[3] / self.weights_symbolic[3]
55
56      def diatonal(self): #Compute diatonicity
57          return self.abs_vector()[4] / self.weights_symbolic[4]
58
59      def tone(self): #Define wholetoneness
60          return self.abs_vector()[5] / self.weights_symbolic[5]
61
62      @classmethod
63      def from_pcp(cls, pcp, symbolic=True):
64          #       Get TIVs from pcp, as the original method
65          #      :param pcp: 12xN vector containing N pcps
66          #      :return: TIVCollection object
67          #      """
68          if pcp.shape[0] == 12:
69              fft = np.fft.rfft(pcp, n=12)
70              energy = fft[0]
71              vector = fft[1:7]
72              if symbolic:
73                  vector = ((vector / energy) * cls.weights_symbolic)
74              else:
75                  vector = ((vector / energy) * cls.weights_audio)
76              return cls(energy, vector)
77          else:
78              return cls(complex(0), np.array([0, 0, 0, 0, 0, 0]).astype(complex))
```

```
79
80      def plot_TIV(self):
81          titles = ["m2/M7", "TT", "M3/m6", "m3/M6", "P4/P5", "M2/m7"]
82          TIVector = self.vector / self.weights_symbolic
83          i = 1
84          for tiv in TIVector:
85              circle = plt.Circle((0, 0), 1, fill=False)
86              plt.subplot(2, 3, i)
87              plt.subplots_adjust(hspace=0.4)
88              plt.gca().add_patch(circle)
89              plt.title(titles[i - 1])
90              plt.scatter(tiv.real, tiv.imag)
91              plt.xlim((-1.5, 1.5))
92              plt.ylim((-1.5, 1.5))
93              plt.grid()
94              i = i + 1
95          plt.show()
96
97      def hchange(self):
98          tiv_array = self.vector
99          results = []
100         for i in range(len(tiv_array)):
101             distance = TIV.euclidean(tiv_array[i + 1], tiv_array[i])
102             results.append(distance)
103         return results
104
105 # # Auxiliary Functions
106 # By Pedro Ramoneda in "Harmonic Change Detection from Musical Audio"
107 def gaussian_blur(centroid_vector, sigma):
108     centroid_vector = gaussian_filter(centroid_vector, sigma=sigma)
109     return centroid_vector
110
111 def get_peaks_hcdf(hcdf_function, rate_centroids_second, symbolic=True):
112     changes = [0]
113     hcdf_changes = []
114     last = 0
115     for i in range(2, hcdf_function.shape[0] - 1):
116         if hcdf_function[i - 1] < hcdf_function[i] and hcdf_function[i + 1] <
    hcdf_function[i]:
117             hcdf_changes.append(hcdf_function[i])
118             if not symbolic:
119                 changes.append(i / rate_centroids_second)
120             else:
121                 changes.append(i)
122             last = i
123     return np.array(changes), np.array(hcdf_changes)
124
125 #Distance Calculation (Euclidean and Cosine)
126 def distance_calc(centroid_point, distance):
```

```
    dist = []
    if distance == 'Euclidean':
        for j in range(1, centroid_point.shape[1] - 1):
            aux = 0
            for i in range(0, centroid_point.shape[0]):
                aux += ((centroid_point[i][j + 1] - centroid_point[i][j - 1]) ** 2)
            aux = np.math.sqrt(aux)
            dist.append(aux)

    if distance == 'Cosine':
        for j in range(1, centroid_point.shape[1] - 1):
            cosine_distance = cosine(centroid_point[:, j - 1], centroid_point[:, j
    + 1])
            dist.append(cosine_distance)
    dist.append(0)

    return np.array(dist)

#Now we will need to take information from TIV. So we will need some additional
    functions
def all_zero(vector):
    for element in vector:
        if element != 0:
            return False
    return True

def real_imag(TIVector):
    aux = []
    for i in range(0, TIVector.shape[1]):
        real_vector = []
        imag_vector = []
        for j in range(0, TIVector.shape[0]):
            real_vector.append(TIVector[j][i].real)
            imag_vector.append(TIVector[j][i].imag)
        aux.append(real_vector)
        aux.append(imag_vector)
    return np.array(aux)

def tonalIntervalSpace(chroma, symbolic=True):
    centroid_vector = []
    for i in range(0, chroma.shape[1]):
        each_chroma = [chroma[j][i] for j in range(0, chroma.shape[0])]
        each_chroma = np.array(each_chroma)
        if all_zero(each_chroma):
            centroid = [0. + 0.j, 0. + 0.j, 0. + 0.j, 0. + 0.j, 0. + 0.j, 0. + 0.j]
        else:
            tonal = TIV.from_pcp(each_chroma, symbolic)          #Calculate the TIV
     for each chroma
```

```
            #tonal.plot_TIV() #PLOT TIV for each chroma -> too expensive in terms of program's space
            centroid = tonal.get_vector()
        centroid_vector.append(centroid)
    return real_imag(np.array(centroid_vector))

def harmonic_change(chroma: list, window_size: int=2048, symbolic: bool=True, sigma: int = 5, dist: str = 'euclidean'):
    chroma = np.array(chroma).transpose()
    centroid_vector = tonalIntervalSpace(chroma, symbolic=True)

    # Blur
    centroid_vector_blurred = gaussian_blur(centroid_vector, sigma)

    # Harmonic Distance Calculation - Euclidean or Cosine
    harmonic_function = distance_calc(centroid_vector_blurred, dist)

    changes, hcdf_changes = get_peaks_hcdf(harmonic_function, window_size, symbolic=True)

    return changes, hcdf_changes, harmonic_function

np.set_printoptions(threshold=sys.maxsize)

# # Piano Roll Representations
def midi_pianoRoll(file):
    midi_data = pretty_midi.PrettyMIDI(file)
    score = libfmp.c1.midi_to_list(midi_data)
    libfmp.c1.visualize_piano_roll(score, figsize=(8, 3), velocity_alpha=True);

#Chroma Vectors
def chromagram(midi_file):
    midi_data = pretty_midi.PrettyMIDI(midi_file)
    score = libfmp.c1.midi_to_list(midi_data)
    df = pd.DataFrame(score, columns=['Start', 'Duration', 'Pitch', 'Velocity', 'Instrument'])
    array_time = np.array(df[['Start']]) #It's in seconds
    array_pitch = np.array(df[['Pitch']])
    df_array = np.column_stack((array_time, array_pitch))
    chromagram = libfmp.b.b_sonification.list_to_chromagram(score, df_array.shape[0], 1)
    chroma = libfmp.b.b_plot.plot_chromagram(chromagram, xlim = (0, array_time[-1]), figsize=(16, 6))
    print(df['Start'].max())

    plt.xlabel("Time (Seconds)")
    plt.ylabel("Pitch")
    plt.title("Chroma Vectors")
    plt.show()
```

```
215
216     return chroma
217
218 #HCDF
219 def hcdf_changes_gt(csv_file):
220     if csv_file.endswith(".xlsx"):
221         df = pd.read_excel(csv_file, header=None)
222     elif csv_file.endswith(".csv"):
223         df = pd.read_csv(csv_file, header=None)
224     else:
225         print("Not a valid excel format")
226     beat_chord_onset = df[0].to_numpy() * 60 / 120
227     return beat_chord_onset
228
229 def HCDF(file, csv_file, sigma=int, distance='Euclidean', resolution = 28):
230     f_measure_results, precision_results, recall_results = [], [], []
231     # read midi
232     midi_vector = pretty_midi.PrettyMIDI(file, resolution, initial_tempo=120)
233
234     # Compute chroma
235     chroma_vector = midi_vector.get_chroma(resolution).transpose()
236
237     # Predicted harmonic changes
238     changes, hcdf_changes, harmonic_function = harmonic_change(chroma=chroma_vector
        , symbolic=True, sigma=sigma, dist = distance)
239     changes = changes / resolution
240
241     # Ground truth harmonic changes
242     changes_ground_truth = hcdf_changes_gt(csv_file)
243
244     #Plot
245     #plt.figure(figsize=(10, 7))
246     #plt.plot(hcdf_changes)
247     #plt.vlines(x=changes_ground_truth, ymin=0, ymax=max(hcdf_changes), colors='
        green')
248     #plt.title('Changes_GT / Changes')
249
250     #  evaluation
251     f_measure, precision, recall = mir_eval.onset.f_measure(changes_ground_truth,
        changes, window=0.628)
252     f_measure_results.append(f_measure)
253     precision_results.append(precision)
254     recall_results.append(recall)
255
256     return np.mean(np.array(f_measure_results)), np.mean(np.array(precision_results
        )), np.mean(np.array(recall_results))
257
258 def tune_sigma_plot(evaluation_result):
259     sigma_list = []; type_metric = []; metrics = []
```

```
    for s, v in evaluation_result.items():
        f, p, r = v
        # F-Measure
        sigma_list.append(s)
        type_metric.append("F_score")
        metrics.append(f)
        # Precision
        sigma_list.append(s)
        type_metric.append("Precision")
        metrics.append(p)
        # Recall
        sigma_list.append(s)
        type_metric.append("Recall")
        metrics.append(r)
    df_dict = {
        "sigma": sigma_list,
        "metric": type_metric,
        "Evaluation Metrics": metrics
    }

    df = pd.DataFrame(df_dict)
    fig = px.line(df, x="sigma", y="Evaluation Metrics", color="metric",
    render_mode="svg")
    fig.show()

def compute_hcdf(lst1, lst2, distance, sigma, resolution):
    f_sc_results = []
    prec_results = []
    rec_results = []
    for file, file2 in zip(lst1, lst2):

        hcdf = HCDF(file, file2, sigma=sigma, distance=distance, resolution=
    resolution)
        f_sc_results.append(hcdf[0])
        prec_results.append(hcdf[1])
        rec_results.append(hcdf[2])

    return np.mean(np.array(f_sc_results)), np.mean(np.array(prec_results)), np.
    mean(np.array(rec_results))

def results(lst1,lst2, resolution):
    for file, file2 in zip(lst1, lst2):
        results_euclidean = {
            sigma: HCDF(file, file2, sigma=sigma, distance='Euclidean',resolution =
     resolution) for sigma in range(0, 50, 10)}
        results_cosine = {
            sigma: HCDF(file, file2, sigma=sigma, distance='Cosine',resolution =
    resolution) for sigma in range(0, 50, 10)}
    return results_euclidean, results_cosine
```

```
304
305 # #HCDF in BPS Dataset
306 path_score_BPS = './Datasets/BPS'
307 file_list_BPS = glob.glob(path_score_BPS + '/*.mid')
308 file_csv_BPS = glob.glob(path_score_BPS + '/*.xlsx')
309
310 lst1_bps = list()
311 lst2_bps = list()
312 for file in file_list_BPS:
313     lst1_bps.append(file)
314 for file in file_csv_BPS:
315     lst2_bps.append(file)
316
317 print("BPS - Euclidean")
318 f_sc_bps_e, p_bps_e, r_bps_e = compute_hcdf(lst1_bps,lst2_bps, 'Euclidean', 10,
        resolution = 28)
319 print(f_sc_bps_e, p_bps_e, r_bps_e)
320 print("BPS - Cosine")
321 f_sc_bps_c, p_bps_c, r_bps_c = compute_hcdf(lst1_bps,lst2_bps, 'Cosine', 10,
        resolution = 28)
322 print(f_sc_bps_c, p_bps_c, r_bps_c)
323
324 results_euclidean_BPS, results_cosine_BPS = results(lst1_bps,lst2_bps,resolution =
        28)
325
326 tune_sigma_plot(results_euclidean_BPS)
327 tune_sigma_plot(results_cosine_BPS)
328
329 # #HCDF in Tavern Dataset
330 #TAVERN consists of three types of files for each musical phrase for each annotator
       (A and B)
331
332 path_Tavern = './Datasets/Tavern'
333 lst_midi_beethoven = list()
334 lst_midi_mozart = list()
335 for file in glob.glob(path_Tavern + './Beethoven/*.mid'):
336     lst_midi_beethoven.append(file)
337 for file in glob.glob(path_Tavern + './Mozart/*.mid'):
338     lst_midi_mozart.append(file)
339
340 lst_csv_beethovenA = list()
341 lst_csv_beethovenB = list()
342 lst_csv_mozartA = list()
343 lst_csv_mozartB = list()
344
345 for file in glob.glob(path_Tavern + './Beethoven/*A.csv'):
346     lst_csv_beethovenA.append(file)
347 for file in glob.glob(path_Tavern + './Beethoven/*B.csv'):
348     lst_csv_beethovenB.append(file)
```

```
for file in glob.glob(path_Tavern + './Mozart/*A.csv'):
    lst_csv_mozartA.append(file)
for file in glob.glob(path_Tavern + './Mozart/*B.csv'):
    lst_csv_mozartB.append(file)

# #Beethoven with Annotator A
print("Beethoven with Annotator A - Euclidean")
f_sc_beethovenA_e, p_beethovenA_e, r_beethovenA_e = compute_hcdf(lst_midi_beethoven
    ,lst_csv_beethovenA, 'Euclidean', 10,resolution = 28)
print(f_sc_beethovenA_e, p_beethovenA_e, r_beethovenA_e)
print("Beethoven with Annotator A - Cosine")
f_sc_beethovenA_c, p_beethovenA_c, r_beethovenA_c = compute_hcdf(lst_midi_beethoven
    ,lst_csv_beethovenA, 'Cosine', 10,resolution = 28)
print(f_sc_beethovenA_c, p_beethovenA_c, r_beethovenA_c)

results_euclidean_TAVERN_Beethoven_A, results_cosine_TAVERN_Beethoven_A= results(
    lst_midi_beethoven,lst_csv_beethovenA,resolution = 28)

tune_sigma_plot(results_cosine_TAVERN_Beethoven_A)
tune_sigma_plot(results_cosine_TAVERN_Beethoven_A)

# #Beethoven with Annotator B
print("Beethoven with Annotator B - Euclidean")
f_sc_beethovenB_e, p_beethovenB_e, r_beethovenB_e = compute_hcdf(lst_midi_beethoven
    ,lst_csv_beethovenB, 'Euclidean', 10,resolution = 28)
print(f_sc_beethovenB_e, p_beethovenB_e, r_beethovenB_e)
print("Beethoven with Annotator B - Cosine")
f_sc_beethovenB_c, p_beethovenB_c, r_beethovenB_c = compute_hcdf(lst_midi_beethoven
    ,lst_csv_beethovenB, 'Cosine', 10,resolution = 28)
print(f_sc_beethovenB_c, p_beethovenB_c, r_beethovenB_c)

results_euclidean_TAVERN_Beethoven_B, results_cosine_TAVERN_Beethoven_B= results(
    lst_midi_beethoven,lst_csv_beethovenB,resolution = 28)

tune_sigma_plot(results_cosine_TAVERN_Beethoven_B)
tune_sigma_plot(results_cosine_TAVERN_Beethoven_B)

# #Mozart with Annotator A
print("Mozart with Annotator A - Euclidean")
f_sc_mozartA_e, p_mozartA_e, r_mozartA_e = compute_hcdf(lst_midi_mozart,
    lst_csv_mozartA, 'Euclidean', 10,resolution = 28)
print(f_sc_mozartA_e, p_mozartA_e, r_mozartA_e)
print("Mozart with Annotator A - Cosine")
f_sc_mozartA_c, p_mozartA_c, r_mozartA_c = compute_hcdf(lst_midi_mozart,
    lst_csv_mozartA, 'Cosine', 10,resolution = 28)
print(f_sc_mozartA_c, p_mozartA_c, r_mozartA_c)

results_euclidean_TAVERN_MozartA, results_cosine_TAVERN_MozartA= results(
    lst_midi_mozart,lst_csv_mozartA,resolution = 28)
```

```

tune_sigma_plot(results_euclidean_TAVERN_MozartA)
tune_sigma_plot(results_cosine_TAVERN_MozartA)

# #Mozart with Annotator B
print("Mozart with Annotator B - Euclidean")
f_sc_mozartB_e, p_mozartB_e, r_mozartB_e = compute_hcdf(lst_midi_mozart,
    lst_csv_mozartB, 'Euclidean', 10,resolution = 28)
print(f_sc_mozartB_e, p_mozartB_e, r_mozartB_e)
print("Mozart with Annotator B - Cosine")
f_sc_mozartB_c, p_mozartB_c, r_mozartB_c = compute_hcdf(lst_midi_mozart,
    lst_csv_mozartB, 'Cosine', 10,resolution = 28)
print(f_sc_mozartB_c, p_mozartB_c, r_mozartB_c)

results_euclidean_TAVERN_MozartB, results_cosine_TAVERN_MozartB= results(
    lst_midi_mozart,lst_csv_mozartB,resolution = 28)

tune_sigma_plot(results_euclidean_TAVERN_MozartB)
tune_sigma_plot(results_cosine_TAVERN_MozartB)


# #HCDF in Bach's Preludes (First Book of Well Tempered Clavier Preludes)
path_Bach_Preludes = './Datasets/Bach_Preludes'
midi_bach = list()
csv_bach = list()
for file in glob.glob(path_Bach_Preludes + './*.mid'):
    midi_bach.append(file)
for file in glob.glob(path_Bach_Preludes + './*.csv'):
    csv_bach.append(file)

print("Bach Preludes - Euclidean")
f_sc_bach_preludes_e, p_bach_preludes_e, r_bach_preludes_e = compute_hcdf(midi_bach
    ,csv_bach, 'Euclidean', 10,resolution = 28)
print(f_sc_bach_preludes_e, p_bach_preludes_e, r_bach_preludes_e)
print("Bach Preludes - Cosine")
f_sc_bach_preludes_c, p_bach_preludes_c, r_bach_preludes_c = compute_hcdf(midi_bach
    ,csv_bach, 'Cosine', 10,resolution = 28)
print(f_sc_bach_preludes_c, p_bach_preludes_c, r_bach_preludes_c)

results_euclidean_Bach_Prelude, results_cosine_Bach_Prelude= results(midi_bach,
    csv_bach,resolution = 28)

tune_sigma_plot(results_euclidean_Bach_Prelude)
tune_sigma_plot(results_cosine_Bach_Prelude)


# #HCDF with Beethoven Quartets (ABC Dataset)
path_ABC_Beethoven_Quartets = './Datasets/ABC(Beethoven_Quartets)'
midi_beeQ = list()
```

```
csv_beeQ = list()
for file in glob.glob(path_ABC_Beethoven_Quartets + './*.mid'):
    midi_beeQ.append(file)
for file in glob.glob(path_ABC_Beethoven_Quartets + './*.csv'):
    csv_beeQ.append(file)

print("Beethoven Quartets (ABC) - Euclidean")
f_sc_beeQ_e, p_beeQ_e, r_beeQ_e = compute_hcdf(midi_beeQ,csv_beeQ, 'Euclidean', 10,
    resolution = 28)
print(f_sc_beeQ_e, p_beeQ_e, r_beeQ_e)
print("Beethoven Quartets (ABC) - Cosine")
f_sc_beeQ_c, p_beeQ_c, r_beeQ_c = compute_hcdf(midi_beeQ,csv_beeQ, 'Cosine', 10,
    resolution = 28)
print(f_sc_beeQ_c, p_beeQ_c, r_beeQ_c)

results_euclidean_Beethoven_Quartets, results_cosine_Beethoven_Quartets = results(
    midi_beeQ,csv_beeQ,resolution = 28)

tune_sigma_plot(results_euclidean_Beethoven_Quartets)
tune_sigma_plot(results_cosine_Beethoven_Quartets)
```

Appendix B

HCDF Code (with Reduction)

In this appendix, the Python code for computing HCDF with reduction process is presented:

```
from music21 import *
from collections import Counter
import plotly.express as px
import sys
import numpy as np
import glob
import pretty_midi
from scipy import spatial
import mir_eval
import matplotlib.pyplot as plt
import pandas as pd
from scipy.spatial.distance import cosine, euclidean
from scipy.ndimage import gaussian_filter

# The full TIV library isn't importing correctly to the program, so here is a part
    of the TIV library.
class TIV:
    weights_symbolic = [2, 11, 17, 16, 19, 7]
    weights_audio = [3, 8, 11.5, 15, 14.5, 7.5]

    def __init__(self, energy, vector):
        self.energy = energy
        self.vector = vector

    def abs_vector(self):
        return np.abs(self.vector)

    def phases_vector(self):
        return np.angle(self.vector)

    def get_vector(self):
        return np.array(self.vector)

```

```
    def diss(self):  # Compute dissonance
        return 1 - (np.linalg.norm(self.vector) / np.sqrt(np.sum(np.dot(self.
    weights_symbolic, self.weights_symbolic))))

    def coeffs(self, coef):  # Compute coefficient
        return self.abs_vector()[coef] / self.weights_symbolic[coef]

    def chromaticity(self):  # Compute chromaticity
        return self.abs_vector()[0] / self.weights_symbolic[0]

    def dyads(self):  # Compute dyadicity
        return self.abs_vector()[1] / self.weights_symbolic[1]

    def triads(self):  # Compute triadicity (triads)
        return self.abs_vector()[2] / self.weights_symbolic[2]

    def d_q(self):  # Refers a possible diminished quality
        return self.abs_vector()[3] / self.weights_symbolic[3]

    def diatonal(self):  # Compute diatonicity
        return self.abs_vector()[4] / self.weights_symbolic[4]

    def tone(self):  # Define wholetoneness
        return self.abs_vector()[5] / self.weights_symbolic[5]

    @classmethod
    def from_pcp(cls, pcp, symbolic=True):
        #       Get TIVs from pcp, as the original method
        #      :param pcp: 12xN vector containing N pcps
        #      :return: TIVCollection object
        #      """
        if pcp.shape[0] == 12:
            fft = np.fft.rfft(pcp, n=12)
            energy = fft[0]
            vector = fft[1:7]
            if symbolic:
                vector = ((vector / energy) * cls.weights_symbolic)
            else:
                vector = ((vector / energy) * cls.weights_audio)
            return cls(energy, vector)
        else:
            return cls(complex(0), np.array([0, 0, 0, 0, 0, 0]).astype(complex))

    def plot_TIV(self):
        titles = ["m2/M7", "TT", "M3/m6", "m3/M6", "P4/P5", "M2/m7"]
        TIVector = self.vector / self.weights_symbolic
        i = 1
        for tiv in TIVector:
            circle = plt.Circle((0, 0), 1, fill=False)
```

```
            plt.subplot(2, 3, i)
            plt.subplots_adjust(hspace=0.4)
            plt.gca().add_patch(circle)
            plt.title(titles[i - 1])
            plt.scatter(tiv.real, tiv.imag)
            plt.xlim((-1.5, 1.5))
            plt.ylim((-1.5, 1.5))
            plt.grid()
            i = i + 1
        plt.show()

    def hchange(self):
        tiv_array = self.vector
        results = []
        for i in range(len(tiv_array)):
            distance = TIV.euclidean(tiv_array[i + 1], tiv_array[i])
            results.append(distance)
        return results

np.set_printoptions(threshold=sys.maxsize)

##Auxilary Functions
#Auxiliary TIV Functions:
def all_zero(vector):
    for element in vector:
        if element != 0:
            return False
    return True

def real_imag(TIVector):
    aux = []
    for i in range(0, TIVector.shape[1]):
        real_vector = []
        imag_vector = []
        for j in range(0, TIVector.shape[0]):
            real_vector.append(TIVector[j][i].real)
            imag_vector.append(TIVector[j][i].imag)
        aux.append(real_vector)
        aux.append(imag_vector)
    return np.array(aux)

def tonalIntervalSpace(chroma, symbolic=True):
    centroid_vector = []
    for i in range(0, chroma.shape[1]):
        each_chroma = [chroma[j][i] for j in range(0, chroma.shape[0])]
        each_chroma = np.array(each_chroma)
        if all_zero(each_chroma):
            centroid = [0. + 0.j, 0. + 0.j, 0. + 0.j, 0. + 0.j, 0. + 0.j, 0. + 0.j]
        else:
```

```
130             tonal = TIV.from_pcp(each_chroma, symbolic)          #Calculate the TIV
     for each chroma
131             #tonal.plot_TIV() #PLOT TIV for each chroma -> too expensive in terms
    of program's space
132             centroid = tonal.get_vector()
133         centroid_vector.append(centroid)
134     return real_imag(np.array(centroid_vector))
135
136 def harmonic_change(chroma: list, window_size: int=2048, symbolic: bool=True, sigma
    : int = 5, dist: str = 'euclidean'):
137     chroma = np.array(chroma).transpose()
138     centroid_vector = tonalIntervalSpace(chroma, symbolic=True)
139
140     # Blur
141     centroid_vector_blurred = gaussian_blur(centroid_vector, sigma)
142
143     # Harmonic Distance Calculation - Euclidean or Cosine
144     harmonic_function = distance_calc(centroid_vector_blurred, dist)
145
146     changes, hcdf_changes = get_peaks_hcdf(harmonic_function, window_size, symbolic
    =True)
147
148     return changes, hcdf_changes, harmonic_function
149
150 #Auxiliary HCDF Functions:
151 #By Pedro Ramoneda in "Harmonic Change Detection from Musical Audio"
152 def gaussian_blur(centroid_vector, sigma):
153     centroid_vector = gaussian_filter(centroid_vector, sigma=sigma)
154     return centroid_vector
155
156 def get_peaks_hcdf(hcdf_function, rate_centroids_second, symbolic=True):
157     changes = [0]
158     hcdf_changes = []
159     last = 0
160     for i in range(2, hcdf_function.shape[0] - 1):
161         if hcdf_function[i - 1] < hcdf_function[i] and hcdf_function[i + 1] <
    hcdf_function[i]:
162             hcdf_changes.append(hcdf_function[i])
163             if not symbolic:
164                 changes.append(i / rate_centroids_second)
165             else:
166                 changes.append(i)
167             last = i
168     return np.array(changes), np.array(hcdf_changes)
169
170 #Distance Calculation (Euclidean and Cosine)
171 def distance_calc(centroid_point, distance):
172     dist = []
173     if distance == 'Euclidean':
```

```
        for j in range(1, centroid_point.shape[1] - 1):
            aux = 0
            for i in range(0, centroid_point.shape[0]):
                aux += ((centroid_point[i][j + 1] - centroid_point[i][j - 1]) ** 2)
            aux = np.math.sqrt(aux)
            dist.append(aux)

    if distance == 'Cosine':
        for j in range(1, centroid_point.shape[1] - 1):
            cosine_distance = cosine(centroid_point[:, j - 1], centroid_point[:, j
    + 1])
            dist.append(cosine_distance)
    dist.append(0)

    return np.array(dist)

#Reduction Functions:
# Just to order list of TIV_Redux
def sort_dist(dist):
    return dist[1]

def dist_calc(pitches_chord, dist):
    dist_list = []
    hist = Counter(pitches_chord)
    pitch_list = [0] * 12
    for pitch in hist:
        index = pitch % 12
        pitch_list[index] += hist[pitch]
    TIV_vector = TIV.from_pcp(np.array(pitch_list))

    for i in range(len(pitches_chord)):
        each_pitch = pitches_chord[i]
        aux_vector = [0] * 12
        aux_vector[each_pitch % 12] += 1
        aux_vector = TIV.from_pcp(np.array(aux_vector))
        distance = dist(TIV_vector.vector, aux_vector.vector)
        dist_list.append((i, distance))
    return dist_list

#NOW WITHOUT DUPLICATED NOTES ON THE CHORD:
def non_duplicated_stream(stream):
  for chord in stream.flatten().getElementsByClass('Chord'):
      chord.removeRedundantPitchNames(inPlace=True)
  return stream

#The aim is to only show the notes that are going to be cut by the reduction step
#(Taking into account that the representation of each chord are only up to 3 notes
    maximum)
def TIV_Redux(pitches_chord, dist):
```

```
221     dist_list = dist_calc(pitches_chord, dist)
222     dist_list.sort(key=sort_dist)
223     dist_reduced = dist_list[3:] #only show the notes that are going to be cut,
        reducing the chord
224     return dist_reduced
225
226 #Reduction Process
227 def info_chord(stream):
228   midi_pitches = []
229   duration_pitches = []
230   offset_pitches = []
231
232   for chord in stream.flatten().getElementsByClass('Chord'):
233     midi_pitches.append([n.pitch.midi for n in chord.notes])
234     duration_pitches.append([n.quarterLength for n in chord.notes])
235     offset_pitches.append([chord.offset for n in chord])
236
237   print(midi_pitches)
238   print(duration_pitches)
239   print(offset_pitches)
240   return midi_pitches, duration_pitches, offset_pitches
241
242 def red_notes_quantifier(lst):
243   reduced_notes_lst = []
244   for s in lst:
245     red_notes = TIV_Redux(s, spatial.distance.euclidean)
246     e = [x[0] for x in red_notes]
247     n_name = [note.Note(s[i]).nameWithOctave for i in e]
248     if n_name:
249         reduced_notes_lst.append(n_name)
250   return reduced_notes_lst
251
252 def times_info(offset_pitches):
253   times_offset = []
254   for value in offset_pitches:
255      if len(value) >= 4:
256          times_offset.append(value[0])
257   return times_offset
258
259 def stream_painted_notes(stream1 ,notes_lst, times_offset):
260   final_reduced_list = zip(notes_lst, times_offset)
261   zipped_reduced_list = list(final_reduced_list)
262   zipped_reduced_list.sort(key=lambda x: x[1])
263
264   for i in range(len(zipped_reduced_list)):
265     for j in range(len(zipped_reduced_list[i][0])):
266       if len(zipped_reduced_list[i][0]) > 0:
267         note_zipped = note.Note(zipped_reduced_list[i][0][j])
268         note_zipped.style.color = 'red'
```

```
        stream1.insert(zipped_reduced_list[i][1], note_zipped)

  stream1 = stream1.flat
  return stream1

def reduced_stream(stream):
  for chord in stream.flatten().getElementsByClass('Chord'):
    chord_new = chord
    n = [n.pitch.midi for n in chord_new.notes]
    n_pitch = [nt for nt in chord_new.pitches] #All pitches from each chord
    red_chord = TIV_Redux(n, spatial.distance.euclidean) #Putting TIV on the
    equation
    e = [x[0] for x in red_chord]
    n_to_cut = [n_pitch[i] for i in e] #The pitches of notes that are going to be
    cutted from the chord
    for element in n_to_cut:
      if element in n_pitch:
        for i in chord_new.pitches:
          if i == element:
            chord_new.remove(i)
  stream.replace(chord,chord_new)
  return stream

#Functions for computing HCDF
def hcdf_changes_gt(csv_file):
    if csv_file.endswith(".xlsx"):
        df = pd.read_excel(csv_file, header=None)
    elif csv_file.endswith(".csv"):
        df = pd.read_csv(csv_file, header=None)
    else:
        print("Not a valid excel format")
    beat_chord_onset = df[0].to_numpy() * 60 / 120
    return beat_chord_onset

def HCDF(file, csv_file, sigma=int, distance='Euclidean', resolution = 28):
    f_measure_results, precision_results, recall_results = [], [], []

    # Read Midi File
    midi_vector = pretty_midi.PrettyMIDI(file, resolution, initial_tempo=120)

    # Compute chroma
    chroma_vector = midi_vector.get_chroma(resolution).transpose()

    # Predicted harmonic changes
    changes, hcdf_changes, harmonic_function = harmonic_change(chroma=chroma_vector
    , symbolic=True, sigma=sigma, dist = distance)
    changes = changes / resolution

    # Ground truth harmonic changes
```

```
315     changes_ground_truth = hcdf_changes_gt(csv_file)
316 
317     #Plot
318     #plt.figure(figsize=(10, 7))
319     #plt.plot(hcdf_changes)
320     #plt.vlines(x=changes_ground_truth, ymin=0, ymax=max(hcdf_changes), colors='
        green')
321     #plt.title('Changes_GT / Changes')
322 
323     # Evaluation
324     f_measure, precision, recall = mir_eval.onset.f_measure(changes_ground_truth,
        changes, window=0.628)
325     f_measure_results.append(f_measure)
326     precision_results.append(precision)
327     recall_results.append(recall)
328 
329     return np.mean(np.array(f_measure_results)), np.mean(np.array(precision_results
        )), np.mean(np.array(recall_results))
330 
331 def tune_sigma_plot(evaluation_result):
332     sigma_list = []; type_metric = []; metrics = []
333     for s, v in evaluation_result.items():
334         f, p, r = v
335         # F-Measure
336         sigma_list.append(s)
337         type_metric.append("F_score")
338         metrics.append(f)
339         # Precision
340         sigma_list.append(s)
341         type_metric.append("Precision")
342         metrics.append(p)
343         # Recall
344         sigma_list.append(s)
345         type_metric.append("Recall")
346         metrics.append(r)
347     df_dict = {
348         "sigma": sigma_list,
349         "metric": type_metric,
350         "Evaluation Metrics": metrics
351     }
352 
353     df = pd.DataFrame(df_dict)
354     fig = px.line(df, x="sigma", y="Evaluation Metrics", color="metric",
        render_mode="svg")
355     fig.show()
356 
357 def compute_hcdf(lst1, lst2, distance, sigma, resolution):
358     f_sc_results = []
359     prec_results = []
```

```
    rec_results = []
    for file, file2 in zip(lst1, lst2):
        hcdf = HCDF(file, file2, sigma=sigma, distance=distance, resolution=
    resolution)
        f_sc_results.append(hcdf[0])
        prec_results.append(hcdf[1])
        rec_results.append(hcdf[2])

    return np.mean(np.array(f_sc_results)), np.mean(np.array(prec_results)), np.
    mean(np.array(rec_results))

def results(lst1,lst2, resolution):
    for file, file2 in zip(lst1, lst2):
        results_euclidean = {
            sigma: HCDF(file, file2, sigma=sigma, distance='Euclidean',resolution =
     resolution) for sigma in range(0, 50, 10)}
        results_cosine = {
            sigma: HCDF(file, file2, sigma=sigma, distance='Cosine',resolution =
    resolution) for sigma in range(0, 50, 10)}
    return results_euclidean, results_cosine

# #HCDF in BPS Dataset
path_score_BPS = './Chordify/BPS'
file_list_BPS = glob.glob(path_score_BPS + '/*.mid')
file_csv_BPS = glob.glob(path_score_BPS + '/*.xlsx')

lst1_bps = list()
lst2_bps = list()
for file in file_list_BPS:
    lst1_bps.append(file)
for file in file_csv_BPS:
    lst2_bps.append(file)

print("BPS - Euclidean")
f_sc_bps_e, p_bps_e, r_bps_e = compute_hcdf(lst1_bps,lst2_bps, 'Euclidean', 10,
    resolution = 28)
print(f_sc_bps_e, p_bps_e, r_bps_e)
print("BPS - Cosine")
f_sc_bps_c, p_bps_c, r_bps_c = compute_hcdf(lst1_bps,lst2_bps, 'Cosine', 10,
    resolution = 28)
print(f_sc_bps_c, p_bps_c, r_bps_c)

results_euclidean_BPS, results_cosine_BPS = results(lst1_bps,lst2_bps,resolution =
    28)

tune_sigma_plot(results_euclidean_BPS)
tune_sigma_plot(results_cosine_BPS)
```

```
# #HCDF in Tavern Dataset
#TAVERN consists of three types of files for each musical phrase for each annotator (A and B)
path_Tavern = './Chordify/Tavern'
lst_midi_beethoven = list()
lst_midi_mozart = list()
for file in glob.glob(path_Tavern + './Beethoven/*.mid'):
    lst_midi_beethoven.append(file)
for file in glob.glob(path_Tavern + './Mozart/*.mid'):
    lst_midi_mozart.append(file)

lst_csv_beethovenA = list()
lst_csv_beethovenB = list()
lst_csv_mozartA = list()
lst_csv_mozartB = list()

for file in glob.glob(path_Tavern + './Beethoven/*A.csv'):
    lst_csv_beethovenA.append(file)
for file in glob.glob(path_Tavern + './Beethoven/*B.csv'):
    lst_csv_beethovenB.append(file)
for file in glob.glob(path_Tavern + './Mozart/*A.csv'):
    lst_csv_mozartA.append(file)
for file in glob.glob(path_Tavern + './Mozart/*B.csv'):
    lst_csv_mozartB.append(file)

# #Beethoven with Annotator A
print("Beethoven with Annotator A - Euclidean")
f_sc_beethovenA_e, p_beethovenA_e, r_beethovenA_e = compute_hcdf(lst_midi_beethoven,lst_csv_beethovenA, 'Euclidean', 10,resolution = 28)
print(f_sc_beethovenA_e, p_beethovenA_e, r_beethovenA_e)
print("Beethoven with Annotator A - Cosine")
f_sc_beethovenA_c, p_beethovenA_c, r_beethovenA_c = compute_hcdf(lst_midi_beethoven,lst_csv_beethovenA, 'Cosine', 10,resolution = 28)
print(f_sc_beethovenA_c, p_beethovenA_c, r_beethovenA_c)

results_euclidean_TAVERN_Beethoven_A, results_cosine_TAVERN_Beethoven_A= results(lst_midi_beethoven,lst_csv_beethovenA,resolution = 28)

tune_sigma_plot(results_cosine_TAVERN_Beethoven_A)
tune_sigma_plot(results_cosine_TAVERN_Beethoven_A)

# #Beethoven with Annotator B
print("Beethoven with Annotator B - Euclidean")
f_sc_beethovenB_e, p_beethovenB_e, r_beethovenB_e = compute_hcdf(lst_midi_beethoven,lst_csv_beethovenB, 'Euclidean', 10,resolution = 28)
print(f_sc_beethovenB_e, p_beethovenB_e, r_beethovenB_e)
print("Beethoven with Annotator B - Cosine")
f_sc_beethovenB_c, p_beethovenB_c, r_beethovenB_c = compute_hcdf(lst_midi_beethoven,lst_csv_beethovenB, 'Cosine', 10,resolution = 28)
```

```
445 print(f_sc_beethovenB_c, p_beethovenB_c, r_beethovenB_c)
446
447 results_euclidean_TAVERN_Beethoven_B, results_cosine_TAVERN_Beethoven_B= results(
        lst_midi_beethoven,lst_csv_beethovenB,resolution = 28)
448
449 tune_sigma_plot(results_cosine_TAVERN_Beethoven_B)
450 tune_sigma_plot(results_cosine_TAVERN_Beethoven_B)
451
452 # #Mozart with Annotator A
453 print("Mozart with Annotator A - Euclidean")
454 f_sc_mozartA_e, p_mozartA_e, r_mozartA_e = compute_hcdf(lst_midi_mozart,
        lst_csv_mozartA, 'Euclidean', 10,resolution = 28)
455 print(f_sc_mozartA_e, p_mozartA_e, r_mozartA_e)
456 print("Mozart with Annotator A - Cosine")
457 f_sc_mozartA_c, p_mozartA_c, r_mozartA_c = compute_hcdf(lst_midi_mozart,
        lst_csv_mozartA, 'Cosine', 10,resolution = 28)
458 print(f_sc_mozartA_c, p_mozartA_c, r_mozartA_c)
459
460 results_euclidean_TAVERN_MozartA, results_cosine_TAVERN_MozartA= results(
        lst_midi_mozart,lst_csv_mozartA,resolution = 28)
461
462 tune_sigma_plot(results_euclidean_TAVERN_MozartA)
463 tune_sigma_plot(results_cosine_TAVERN_MozartA)
464
465 # #Mozart with Annotator B
466 print("Mozart with Annotator B - Euclidean")
467 f_sc_mozartB_e, p_mozartB_e, r_mozartB_e = compute_hcdf(lst_midi_mozart,
        lst_csv_mozartB, 'Euclidean', 10,resolution = 28)
468 print(f_sc_mozartB_e, p_mozartB_e, r_mozartB_e)
469 print("Mozart with Annotator B - Cosine")
470 f_sc_mozartB_c, p_mozartB_c, r_mozartB_c = compute_hcdf(lst_midi_mozart,
        lst_csv_mozartB, 'Cosine', 10,resolution = 28)
471 print(f_sc_mozartB_c, p_mozartB_c, r_mozartB_c)
472
473 results_euclidean_TAVERN_MozartB, results_cosine_TAVERN_MozartB= results(
        lst_midi_mozart,lst_csv_mozartB,resolution = 28)
474
475 tune_sigma_plot(results_euclidean_TAVERN_MozartB)
476 tune_sigma_plot(results_cosine_TAVERN_MozartB)
477
478
479 # #HCDF in Bach's Preludes (First Book of Well Tempered Clavier Preludes)
480 path_Bach_Preludes = './Chordify/Bach_Preludes'
481 midi_bach = list()
482 csv_bach = list()
483 for file in glob.glob(path_Bach_Preludes + './*.mid'):
484     midi_bach.append(file)
485 for file in glob.glob(path_Bach_Preludes + './*.csv'):
486     csv_bach.append(file)
```

```
487
488 print("Bach Preludes - Euclidean")
489 f_sc_bach_preludes_e, p_bach_preludes_e, r_bach_preludes_e = compute_hcdf(midi_bach
        ,csv_bach, 'Euclidean', 10,resolution = 28)
490 print(f_sc_bach_preludes_e, p_bach_preludes_e, r_bach_preludes_e)
491 print("Bach Preludes - Cosine")
492 f_sc_bach_preludes_c, p_bach_preludes_c, r_bach_preludes_c = compute_hcdf(midi_bach
        ,csv_bach, 'Cosine', 10,resolution = 28)
493 print(f_sc_bach_preludes_c, p_bach_preludes_c, r_bach_preludes_c)
494
495 results_euclidean_Bach_Prelude, results_cosine_Bach_Prelude= results(midi_bach,
        csv_bach,resolution = 28)
496
497 tune_sigma_plot(results_euclidean_Bach_Prelude)
498 tune_sigma_plot(results_cosine_Bach_Prelude)
499
500
501 # #HCDF with Beethoven Quartets (ABC Dataset)
502 path_ABC_Beethoven_Quartets = './Chordify/ABC_(Beethoven_Quartets)'
503 midi_beeQ = list()
504 csv_beeQ = list()
505 for file in glob.glob(path_ABC_Beethoven_Quartets + './*.mid'):
506     midi_beeQ.append(file)
507 for file in glob.glob(path_ABC_Beethoven_Quartets + './*.csv'):
508     csv_beeQ.append(file)
509
510 print("Beethoven Quartets (ABC) - Euclidean")
511 f_sc_beeQ_e, p_beeQ_e, r_beeQ_e = compute_hcdf(midi_beeQ,csv_beeQ, 'Euclidean', 10,
        resolution = 28)
512 print(f_sc_beeQ_e, p_beeQ_e, r_beeQ_e)
513 print("Beethoven Quartets (ABC) - Cosine")
514 f_sc_beeQ_c, p_beeQ_c, r_beeQ_c = compute_hcdf(midi_beeQ,csv_beeQ, 'Cosine', 10,
        resolution = 28)
515 print(f_sc_beeQ_c, p_beeQ_c, r_beeQ_c)
516
517 results_euclidean_Beethoven_Quartets, results_cosine_Beethoven_Quartets = results(
        midi_beeQ,csv_beeQ,resolution = 28)
518
519 tune_sigma_plot(results_euclidean_Beethoven_Quartets)
520 tune_sigma_plot(results_cosine_Beethoven_Quartets)
```

References

[1] Gilberto Bernardes, Diogo Cocharro, Marcelo Caetano, Carlos Guedes, and Matthew Davies. A multi-level tonal interval space for modelling pitch relatedness and musical consonance. *Journal of New Music Research*, 45:1–14, 05 2016. doi:10.1080/09298215.2016.1182192.

[2] Pedro Ramoneda and Gilberto Bernardes. Revisiting harmonic change detection. In *Audio Engineering Society Convention 149*. Audio Engineering Society.

[3] Tsung-Ping Chen and Li Su. Attend to chords: Improving harmonic analysis of symbolic music using transformer-based models. *Transactions of the International Society for Music Information Retrieval*, 4(1):1–13, 2021. doi:10.5334/tismir.65.

[4] Leonhard Euler. *Tentamen novae theoriae musicae [microform] : ex certissimus harmoniae principiis dilucide expositae / auctore Leonhardo Eulero*. Ex typographia Academiae scientiarum Petropoli, 1739.

[5] Christopher Harte, Mark Sandler, and Martin Gasser. Detecting harmonic change in musical audio. 10 2006. doi:10.1145/1178723.1178727.

[6] Meinard Müller and Frank Zalkow. libfmp: A python package for fundamentals of music processing. *Journal of Open Source Software*, 6(63), 2021. doi:10.21105/joss.03326.

[7] Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 17(1):168–192, 2020. doi:10.1016/j.aci.2018.08.003.

[8] António Ramires, Gilberto Bernardes, Matthew E. P. Davies, and Xavier Serra. Tiv.lib: an open-source library for the tonal description of musical audio, 2020. arXiv:2008.11529.

[9] Mike Cuthbert and Christopher Ariza. Music21: A toolkit for computer-aided musicology and symbolic music data. In *ISMIR*, 2010.

[10] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. Van Der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vanderplas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul Van Mulbregt, and Sam Tygier. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, February 2020. doi:10.1038/s41592-019-0686-2.

[11] Meinard Müller. *Fundamentals of Music Processing: Using Python and Jupyter Notebooks*. 01 2021. doi:10.1007/978-3-030-69808-9.

[12] Meinard Müller and Frank Zalkow. Fmp notebooks: Educational material for teaching and learning fundamentals of music processing. In *ISMIR*, 2019.

[13] Meinard Müller. An educational guide through the fmp notebooks for teaching and learning fundamentals of music processing. 2021.

[14] Richard Cohn, Brian Hyer, Carl Dahlhaus, Julian Anderson, and Charles Wilson. Harmony, 2001. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000050818, doi:10.1093/gmo/9781561592630.article.50818.

[15] E. Prout. *Analytical Key to the Exercises in Harmony: Its Theory and Practice*. Augener's edition. Augener & Company, 1903. URL: https://books.google.pt/books?id=ROU-AAAAYAAJ.

[16] Chord, 2001. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000005671, doi:10.1093/gmo/9781561592630.article.05671.

[17] Triad, 2001. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000028347, doi:10.1093/gmo/9781561592630.article.28347.

[18] Néstor Nápoles López, Mark R H Gotham, and Ichiro Fujinaga. AugmentedNet: A Roman Numeral Analysis Network with Synthetic Training Examples and Additional Tonal Tasks. In *Proceedings of the 22nd International Society for Music Information Retrieval Conference*, pages 404–411, Online, November 2021. ISMIR. URL: https://doi.org/10.5281/zenodo.5624533, doi:10.5281/zenodo.5624533.

[19] Brian Hyer. *Tonality*, page 726–752. The Cambridge History of Music. Cambridge University Press, 2002. doi:10.1017/CHOL9780521623711.025.

[20] Fred Lerdahl. Tonal pitch space. *Music Perception*, 5:315–350, 01 1988.

[21] Elaine Chew. Towards a mathematical model of tonality. 2000.

[22] Roger N. Shepard. Geometrical approximations to the structure of musical pitch. *Psychological review*, 89 4:305–33, 1982.

[23] Ayush Shah, Manasi Kattel, Araju Nepal, and D. Shrestha. Chroma feature extraction. 01 2019.

[24] Emilia Gómez. *Tonal Description of Music Audio Signals*. PhD thesis, Universitat Pompeu Fabra, 2006. doi:http://hdl.handle.net/10803/7537.

[25] Takuya Fujishima. Realtime chord recognition of musical sound: a system using common lisp music. In *ICMC*, 1999.

[26] João Paulo Caetano Pereira. Musikverb : A harmonically adaptive audio reverberation. 2018.

[27] George Sioros and Gilberto Bernardes. *The Sostenante Pedal: A Novel Pedal for MIDI Keyboards*. Mar 2016.

[28] Dan Ellis. Chroma feature analysis and synthesis. *Resources of Laboratory for the Recognition and Organization of Speech and Audio-LabROSA*, 5, 2007.

[29] Judith Brown and Miller Puckette. "an efficient algorithm for the calculation of a constant q transform". *Journal of the Acoustical Society of America*, 92:2698, 11 1992. doi:10.1121/1.404385.

[30] Matthias Mauch and Simon Dixon. Approximate note transcription for the improved identification of difficult chords. pages 135–140, 01 2010.

[31] J Pauwels, K O'Hanlon, E Gómez, and MB Sandler. 20 years of automatic chord recognition from audio. In *International Society for Music Information Retrieval (ISMIR)*, Delft, Netherlands, 04/11/2019 2019. URL: http://hdl.handle.net/10230/42773.

[32] Jonggwon Park, Kyoyun Choi, Sungwook Jeon, Dokyun Kim, and Jonghun Park. A bi-directional transformer for musical chord recognition. In Arthur Flexer, Geoffroy Peeters, Julián Urbano, and Anja Volk, editors, *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019, Delft, The Netherlands, November 4-8, 2019*, pages 620–627, 2019. URL: http://archives.ismir.net/ismir2019/paper/000075.pdf.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[34] Fred Lerdahl and Ray Jackendoff. *A generative theory of tonal music*. The MIT Press, Cambridge. MA, 1983.

[35] Eita Nakamura, Masatoshi Hamanaka, Keiji Hirata, and Kazuyoshi Yoshii. Tree-structured probabilistic model of monophonic written music based on the generative theory of tonal music. *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 276–280, 2016.

[36] Federico Simonetta, Filippo Carnovalini, Nicola Orio, and Antonio Rodà. Symbolic music similarity through a graph-based representation. In *Proceedings of the Audio Mostly 2018 on Sound in Immersion and Emotion*, AM'18, New York, NY, USA, 2018. Association for Computing Machinery. URL: https://doi.org/10.1145/3243274.3243301, doi:10.1145/3243274.3243301.

[37] Filippo Carnovalini, Antonio Rodà, Nicholas Harley, Steven T. Homer, and Geraint A. Wiggins. *A New Corpus for Computational Music Research AndA Novel Method for Musical Structure Analysis*, page 264–267. Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3478384.3478402.

[38] Colin Raffel, Brian Mcfee, Eric Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, and Daniel Ellis. mir$_{e}$*val* : *Atransparentimplementationofcommonmirmetrics*. 102014.

[39] Johanna Devaney, Claire Arthur, Nathaniel Condit-Schultz, and Kirsten Nisula. Theme and variation encodings with roman numerals (TaVERn): A new data set for symbolic music analysis. In

Proceedings of the 16th International Society for Music Information Retrieval Conference, ISMIR 2015, pages 728–734, Málaga, Spain, 2015. International Society for Music Information Retrieval. URL: http://ismir2015.uma.es/articles/261_Paper.pdf.

[40] Markus Neuwirth, Daniel Harasim, Fabian Moss, and Martin Rohrmeier. The annotated beethoven corpus (abc): A dataset of harmonic analyses of all beethoven string quartets. *Frontiers in Digital Humanities*, 5, 07 2018. doi:10.3389/fdigh.2018.00016.

[41] Ellis D. P. W Raffel, C. Intuitive analysis, creation and manipulation of midi data with pretty_midi. *Proceedings of the International Society for Music Information Retrieval Conference (Late Breaking and Demo Papers)*, 2014.

[42] Nestor Napoles Lopez. *Automatic harmonic analysis of classical string quartets from symbolic score*. PhD thesis, Universitat Pompeu Fabra, December 2017. URL: https://doi.org/10.5281/zenodo.1095617, doi:10.5281/zenodo.1095617.