Politécnica UFRJ

SYNCHRONIZATION AND SCORE ALIGNMENT OF MUSICAL RECORDINGS

Bernardo Vieira de Miranda

Projeto de Graduação apresentado ao Curso de Engenharia Eletrônica e de Computação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Luiz Wagner Pereira Biscainho

Rio de Janeiro

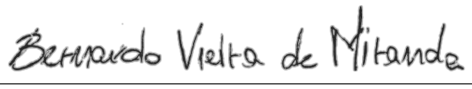
Julho de 2021

SYNCHRONIZATION AND SCORE ALIGNMENT OF MUSICAL RECORDINGS

Bernardo Vieira de Miranda

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRÔNICO E DE COMPUTAÇÃO

Autor:

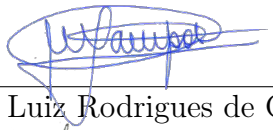


Bernardo Vieira de Miranda

Orientador:

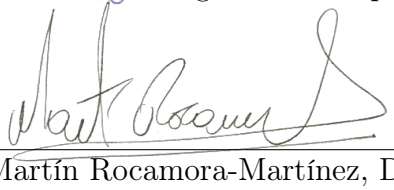
Prof. Luiz Wagner Pereira Biscainho, D.Sc.

Examinador:



Prof. Marcello Luiz Rodrigues de Campos, Ph.D.

Examinador:



Prof. Martín Rocamora-Martínez, D.Sc.

Examinador:

Maurício do Vale Madeira da Costa, D.Sc.

Rio de Janeiro

Novembro de 2021

Declaração de Autoria e de Direitos

Eu, *Bernardo Vieira de Miranda*, CPF *173.196.777-25*, autor da monografia *Synchronization and Score Alignment of Musical Recordings*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.

2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.

3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.

4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.

5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.

6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.

7. Por ser verdade, firmo a presente declaração.

Bernardo Vieira de Miranda

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

DEDICATÓRIA

Aos músicos, que leem nas entrelinhas das partituras.

AGRADECIMENTO

Terminar este trabalho e a minha graduação teria sido impossível se estivesse sozinho. Por isso, gostaria de agradecer à minha mãe Silvia, pela eterna paciência e companheirismo ao longo de todos esses anos, e ao meu pai Paulo André, pelos conselhos profissionais e por ter incentivado minha independência desde sempre. Agradeço também ao meu padrasto, Jorge, pelo gosto pela música e pela matemática sem o qual não estaria aqui, e à minha madrasta, Suzana, por ter me passado o pragmatismo que tantas vezes me foi útil na graduação.

Agradeço aos meus amigos de escola: Bernardo, Enzo, Marcel, Hugo, Gabriel, Pedro, Eduardo, Henrique, Daniel, Paulos Feghali e Castro, Acquarone, Armando, Arthur e Moura, pelos incontáveis churrascos e momentos de descontração, mesmo quando eu estava afundado além da conta em trabalho. Pelos almoços no Grêmio da COPPE e risadas na UFRJ, meu muito obrigado também aos amigos da faculdade: Dudu, Carol, Bruno, Pedro Henrique, Antônio, Matheus, Vinícius, Iuri, Victor, Alexandre e Fabiana. A estes últimos, um agradecimento especial também por terem me mostrado o ferramental que possibilitou que esse trabalho fosse realizado.

Quero agradecer também à minha namorada, Clara Faria, pelo apoio incondicional, por me escutar sempre, e pela paciência com minha personalidade muitas vezes complicada. Esses últimos meses de trabalho teriam sido infinitamente mais difíceis sem ela.

Finalmente, gostaria de agradecer ao meu orientador, Luiz Wagner Biscainho, por ter me guiado nesse trabalho e por ser um exemplo de comprometimento social e profissional. Entregando este trabalho me formo, e passo a ter o enorme privilégio de chamá-lo não só de amigo, mas também de colega.

RESUMO

Este projeto tem como objetivo criar uma versão minimamente funcional de um aplicativo para aprendizado de música através da análise de *performances*. Como cada músico imprime em suas interpretações seu estilo único e pessoal de tocar, frequentemente estudantes de música e musicologia se vêem diante do problema de ter que comparar diferentes versões de uma mesma obra. Devido à necessidade de marcar as minutagens de trechos específicos em cada gravação de interesse, essa tarefa normalmente é trabalhosa, e atualmente não existe interface de livre acesso onde seja possível acompanhar uma partitura e diversas gravações de uma peça musical. Tendo isso em vista, este trabalho desenvolve um *player* de áudio em versão *web* onde é possível, simultaneamente, trocar livremente entre interpretações de uma mesma obra e acompanhar o desenvolvimento de uma obra através de sua partitura. Para isso, representações em *chroma* de cada uma das diferentes gravações fornecidas pelo usuário são usadas como entrada do algoritmo de comparação *Dynamic Time Warping* (DTW), que fornece a equivalência entre quadros das interpretações. No caso do acompanhamento de desenvolvimento, utiliza-se como referência de interpretação uma gravação sintética, criada através do arquivo fornecido pelo usuário, cujas notas ocorrem precisamente nos instantes indicados na partitura. O trabalho menciona algumas heurísticas para a seleção dos parâmetros necessários para os cálculos das representações e da DTW, e ao final são apresentadas as vantagens e limitações do método usado, bem como capturas de tela da interface produto deste projeto.

Palavras-Chave: música, análise de *performance*, representações *chroma*, DTW, interfaces *web*.

ABSTRACT

This project aims to create a minimally functional version of an application for learning music through performance analysis. Because each musician has a unique and personal playing style, often music and musicology students are faced with the problem of comparing different versions of the same musical piece. This is commonly a time consuming task that involves marking the minutes of passages of interest in each of the recordings being studied, and currently there is no easily accessible interface where it is possible to both follow a score and freely switch between different interpretations of a given piece. Having this in mind, this work develops an audio player using web technologies where it is possible to perform both of these tasks. For this, *chroma* representations of each one of the recordings provided by the user are inputted to the Dynamic Time Warping (DTW) similarity algorithm, which is responsible for finding the inter-frame equivalences between the inputs. For score following, a synthetic recording created using the music sheet file given by the user is used as the reference of a mechanical interpretation where all notes happen exactly where the score indicates. This study mentions some heuristics for choosing the right set of parameters to use in the DTW in *chroma* feature extraction, and also analyses the advantages and limitations of the proposed method. In the end, screenshots of the resulting interface are provided.

Key-words: music, performance analysis, *chroma* features, DTW, web interfaces,

ABBREVIATIONS

CSS - Cascading Style Sheets

DFT - Discrete Fourier Transform

DTFS - Discrete-Time Fourier Series

DTFT - Discrete-Time Fourier Transform

DTW - Dynamic Time Warping

FS - Fourier Series

FT - Fourier Transform

HTML - HyperText Markup Language

MIDI - Musical Instrument Digital Interface

OSMD - OpenSheetMusicDisplay

RNN - Recurrent Neural Network

STFT - Short-Time Fourier Transform

Contents

List of Figures

Chapter 1

Introduction

Music, while easy to listen to, is often hard to describe. Despite being simply a creatively ordered collection of sounds, music is able to communicate feelings and sensations that are difficult to define. For every song that exists, each musician has their own interpretation of how it should be played. Professional artists spend years trying to master the best ways of expressing themselves through music, and even today, there are heated debates as to how certain pieces should be played in order to create particular sensations in the audience [1, 2]. The study of the interpretative variations that make two recordings of the same song sound different is called performance analysis [3, 4]. It is a subject of interest not only to professional musicians and musicologists, who make a living out of interpreting musical pieces, but also to amateur players and enthusiasts who enjoy music as an art.

Currently, there is no practical way to study performance differences between artists. For every piece that one may be interested in studying, there is the laborious task of finding the music sheets, hearing each version that is going to be compared, noting down the minutes of the parts of interest, and then going back and forth between recordings while trying to notice the subtleties of each musicians' playing style. Having this in mind, this project aims to create a minimal viable system that allows the user to switch between different recordings with ease and also follow music scores, in order to be able to conveniently compare versions of the same piece. We will examine in detail the signal processing techniques that are needed to find equivalent instants between different recordings, and also create a user friendly interface that can be easily used for performance analysis.

1.1 Musical scores and what is not written in them

Historically, the most traditional form of representing music is through musical scores, or music sheets. Scores contain the instructions for playing a certain piece of music, and make use of specific notation to graphically represent them in paper. However, sheet music is nothing more than a guideline to play a song. Even though tempo and dynamics are almost always written in the score, musicians will often interpret pieces differently by varying them within a certain range, and also by articulating musical sentences in distinct manners.

Furthermore, some musical notation symbols are quite vague in their meaning. A *fermata*, for example, is defined by the Merriam-Webster dictionary as "a prolongation at the discretion of the performer of a musical note, chord, or rest beyond its given time value" [5]. In the score for Beethoven's Fifth Symphony the instructions say that the piece should be played in *allegro con brio*, which means with "a brisk and lively tempo".

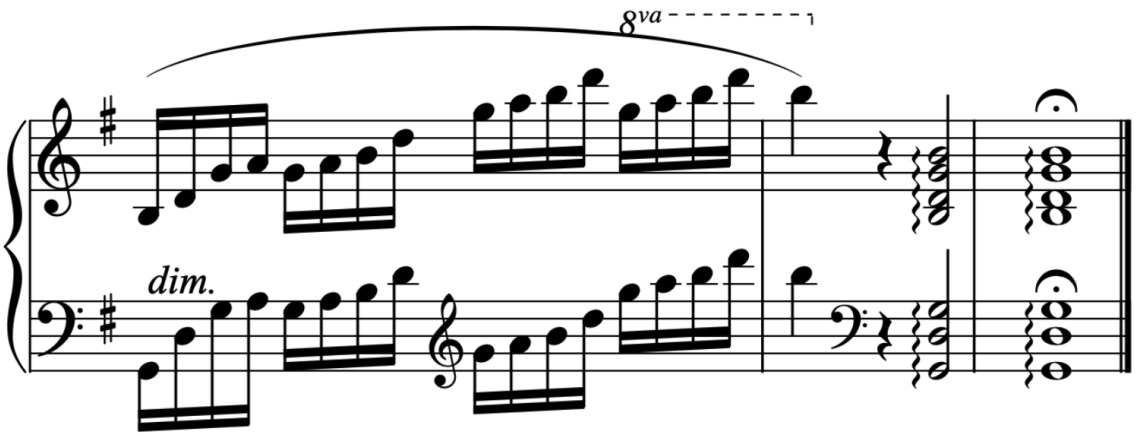


Figure 1.1: Example of musical score showing the many different symbols used in music notation. The semicircle over the last group of notes indicates a *fermata*.

This goes to show that musical scores do not have all the information to precisely determine a music performance. Different musicians will always read sheet music distinctly according to their own views of the piece being played, and also to their own playing styles and habits.

There are other, more recent, ways of representing a musical work. The first and most obvious one is through audio recordings, which capture performances in a record, compact disk, tape, or digital audio file. While recordings perfectly register

what makes a performer's interpretation unique — tempo and dynamic variations, style changes, and articulation —, they do not explicitly show which notes are being played and when.

The alternative to score and audio representations are the so-called symbolic formats [3], which stand somewhere in between the other two. Symbolic audio representations, such as MIDI [6], explicitly show the notes being played and contain instructions that can be converted into sound, but do not reproduce the interpretative nuances of an audio recording. Similarly, symbolic score representations, e.g. MusicXML [7], detail the notes that should be present in the score, and could be converted to sheet music through automatic engraving techniques, but do not inform the final layout of the music symbols on the page.

Performance analysis is challenging because, regardless of the representation that is being used for a song, there will always be gaps that need to be filled by looking at the piece from different perspectives. In order to understand the technique of a performer, and compare it to others, it is necessary to hear different audio recordings of the piece, see the parts of interest in the music sheet, and, ideally, also be capable of having an idea of what would the piece sound like if it was played mechanically, with no interpretative variations.

The main goal of this work is helping the people interested in performance analysis by creating an interface that could simultaneously display these different perspectives of a musical piece in an easy way to use. To reach this objective, two features must absolutely be present in the final product given to the user: interpretation switching, and score following.

1.2 Synchronization of music recordings

Interpretation switching, is the capability of changing between two or more interpretations of the same piece during playback. This means that the user, at any moment during playback of a musician's recording, should be able to press a button and resume audio reproduction from the equivalent instant in another performer's version of the same piece.

This feature is important in an interface for performance analysis because it allows the user to hear an extract by a performer and immediately compare it to the same part of the piece executed by another artist. In this way, the playing style of the first musician is still fresh in the user's head, and parallels between the two interpretations can be easily made.

A little less subjectively, the definition of equivalent instant is that of a musically analogous moment in the second performer's version. This implies finding the temporal correspondence between times in two different versions of the same piece, a task that is also called synchronization of music recordings. In this study, we will use the approach detailed in [3] for finding these equivalences. It starts by transforming audio representations of different recordings into sequences of feature frames that numerically represent the musical content present in chunks of the performances. These numerical representations are called *chroma* features, because they show the amount of energy in each of the twelve musical "colors", or musical notes, regardless of the octave[1]. The features are then used as input to an alignment algorithm called dynamic time warping (DTW), which is responsible for comparing them and providing correspondences between the chunks from the various recordings, as they evolve over time.

This method seems to be the standard in the context of audio synchronization, and there are several systems for interpretation switching that have successfully implemented it [8, 9, 10, 11] or some variation [12, 13]. In [10], this framework was validated in a survey conducted with music students that were supposed to compare nine different recordings of the same extract of Beethoven's Pathétique Sonata Op. 13 using a system for audio synchronization.

There are alternatives both to the *chroma* features [14, 15], and to the DTW [12, 13], but having in mind the wide adoption of this framework for synchronization tasks, and the fact that this project aims more at the creation of an accessible interface than at the investigation of novel audio alignment techniques, a decision was made to adopt the method without further questioning.

[1] An octave is the interval between a musical note and the next note of the same name. For example, the interval between $\boldsymbol{E_3}$ and $\boldsymbol{E_4}$ is an octave.

1.3 Score following

Score following is the task of automatically marking down in a music sheet the current position of a song during playback time. As of today, there are many software solutions for editing music sheets [16, 17, 18] that are capable of synthesizing music based on a symbolic representation, and follow a score along its playback. However, following the audio representation of a human performance is a less straightforward task.

Being able to visually identify the notes being played in a digitally rendered score helps the user to locate him or herself in the music sheet, and also creates the possibility of visually identifying the differences between performances. Moreover, it opens up the possibility of having an interactive score, where the user might be able to navigate by clicking on notes or measures, which would make the interface even easier to use.

Currently, there are solutions for following scores in live performances [19], and even accompanying sheet music for single instruments with digital recordings [20]. Also, the author of [3] was involved in the creation of a web interface for choir rehearsals [21] that is able to follow the score of the different tracks of a choir recording; and recently, the authors of [14] suggested trying to automatically transcribe audio recordings in order to compare them to the music sheet for following.

Yet, there seems to be no readily available interface where it is possible to both follow a score and switch between different interpretations of it. In [9, 11], the authors mention a framework for multi-modal music listening that includes interpretation switching, but the link to the demonstration redirects to a web page that does not exist. Their method consisted in using optical music recognition [3] as a first step towards obtaining a representation for the score in the same domain of audio recordings, and then using the DTW to synchronize the performances to the music sheet.

Here, to maintain the alignment techniques shown in [3], we will adapt this idea and try to align the different recordings available of a given piece with a synthesized version of it. This artificial performance is extracted from a score provided in MusicXML format, and represents a version of the piece without any interpre-

tative traits. From the synthetic recording, the same *chroma* features used in the interpretation switcher can be extracted, and once again the DTW can be used to find the optimal time equivalences between chunks of the performances.

1.4 Goals

The main goal of this project is creating a viable interface for learning music and analyzing different performances of the same recording. To reach this objective, the interface must meet the following criteria:

- being able to play and switch between different interpretations of the same piece;
- allowing the user to accompany playback on the music sheet using the score follower;
- being user friendly and easy to use and navigate.

Meeting these intermediate objectives will make sure that this first version of the interface is usable, and will be a first step in the direction of making it available to the general public online.

1.5 Development tools

In order to create the different building blocks of the interface, a number of tools are used to implement the required features. Most of the project is coded using Python [22], a flexible, interpreted language that is used for many purposes ranging from web development to scientific computing.

Because Python is a popular, all-purpose programming language, there are many readily available packages that implement some of the algorithms that will be seen here. In particular, the librosa [23] library contains implementations for most of the feature calculation and alignment procedures that will be detailed in Chapter 2, including the extraction of *chroma* features and an implementation of the DTW.

Python is also very useful to parse the MusicXML files that are the input for the score follower. The parsers in the music21 package are able to translate these files

into MIDI format, and other Python libraries such as midi2audio can communicate with synthesizers responsible for transforming them into audio. For this last step a synthesizer is obviously required, and FluidSynth was chosen for being open-source and communicating with midi2audio.

As we will see in Chapter 4, in the end it was not possible to create the user interface using solely Python. Engraving digital sheet music is, by itself, a big challenge, and so web technologies are used to render the MusicXML files and create the interface. Even though the pages are created and styled using HTML and CSS, rendering the sheets and controlling playback is done using Javascript, thanks to the language's natural support for event oriented development, and to OpenSheetMusicDisplay, an excellent library for score engraving using MusicXML input.

The integration between the Python code for audio processing and this web based front end happens using the Flask framework, a Python package for web development. With it, it is possible to serve the user interface with the calculations performed by the Python audio processing libraries mentioned above.

1.6 Organization of this project

The following chapters describe the theoretical foundations required for understanding the project, the algorithms, libraries and techniques that are used, and also present a number of experiments that detail the advantages and inconveniences of using the framework that we briefly saw in this introduction.

Chapter 2 is dedicated to the theoretical basis needed to understand how to digitally process audio recordings, extract meaningful features, and compare different audio representations. It covers everything ranging from the mathematical representation of audio signals, to the different variants of dynamic time warping, passing through important details and trade-offs of time-frequency representations. This chapter is complemented by Appendix A; the reader who finds that the concepts present in Chapter 2 are not detailed enough is strongly encouraged to take a look at this appendix.

In Chapter 3 the subject of interest is the interpretation switcher and its performance aligning different types of recordings. It starts explaining how the switching feature is implemented in general terms, introduces the librosa package with all of its important features, and closes with a series of experiments that present: the dataset that was mainly used to test the system, some heuristics for choosing the best set of parameters when aligning audio recordings, and the challenges of the typical use case of the interpretation switcher.

Chapter 4 describes the implementation of the score follower and the operation of the library that is used to render the score on the screen, OpenSheetMusicDisplay. A significant part of it is dedicated to explaining how the onset times of the musical notes in the synthetic version can be calculated, and how this is used for score following. This chapter is fundamental for understanding why an event oriented programming language such as Javascript makes the task of creating an interactive score much easier, and also includes many examples which help understand the limitations of the chosen framework.

The integration of the score follower and the interpretation switcher is described in Chapter 5, where a short explanation on the basic architecture used in modern web applications is presented. This chapter contains screenshots of the interface containing switcher and follower and showcases the main features of the final product.

Finally, Chapter 6 concludes this project with an overview of all topics covered, including remarks about the general performance of the system, and closes the text mentioning possible future work.

Chapter 2

Theoretical Foundations

Having introduced the subject of this study, we present in this chapter the theoretical basis used to represent and analyze music for synchronization. We will start with a brief introduction to signal representations in general, but with a strong emphasis on the short-time Fourier transform and the spectrogram, since they are the starting point for the extraction of audio features explained in Section 2.2 and are used in the audio synchronization algorithm.

We will then proceed by explaining the advantages of using features based on musical notes rather than frequencies, highlighting the robustness of the *chromagram* in Section 2.2.2, before finally closing with an explanation of the dynamic time warping (DTW) algorithm, and how some of its variants can improve audio synchronization performance to the standards required for interpretation switching and score following.

2.1 Usual signal representations

In the field of signal processing, music, speech, and audio in general are manipulated as signals. As defined in [24], signals are functions of one or more variables that carry information on the nature of a physical phenomenon.

The most intuitive way of analyzing a signal is trough the evolution of the represented phenomenon in time, but as we will show along this section, it is also possible to examine a signal as a function of frequency or with a mixed representation using both domains. As an example, consider a signal containing a note being played

on a piano. Through a representation in time domain, it would be possible to identify, for instance, the moment this note was played; a description in frequency domain would reveal the harmonic content present in the signal, from which one could infer the note that was played; finally, a time-frequency representation would convey both information simultaneously.

2.1.1 Signals in time

The time representation of audio signals is called a waveform, and it shows the deviation of air pressure at a specific position in space, with respect to a reference value [3]. In the case of a sound coming from a vibrating string, for instance, the waveform would show how the molecules of air oscillate around a point in space while the sound wave from the string propagates.

Prior to being available in a digital format, the sound of the string in our previous example would be transformed into an electrical signal via a microphone, which is a transducer that converts variations in pressure into variations in electrical voltage, then sampled and quantized[1] using an analog-to-digital converter. In this last step, the signals go from being analog and time-continuous — and thus being able to assume any value at any given time — to being digital and time-discrete — meaning they only assume a finite set of values at samples spaced in time —, and so become computer readable.

In traditional uniform sampling, signals are converted from continuous to discrete by measuring the signal values at times equally spaced by a sampling period T_s; ideally, sampled signals can be perfectly reconstructed, as long as their sampling frequency $F_\mathrm{s} = 1/T_\mathrm{s}$ is greater than twice the highest frequency present in the signal. More details on the sampling process can be found in [24, 3, 26], and in Appendix A.

The interested reader that decides to record the single piano note of the example in an audio editing software like Goldwave [27] or Audacity [28] might get something similar to what is shown in Figure 2.1. In the image, it is possible to see

[1] Quantization is beyond the scope of this study, however, it is enough to know that it maps an interval of continuous values into a set of discrete binary numbers. Here, unless stated otherwise, audio are quantized using 16-bit signed PCM, which has a maximum absolute value of 32768 after conversion to decimal. More on quantization can be seen in [25].

the evolution of the sound along time, but it is not possible to determine which note was played.

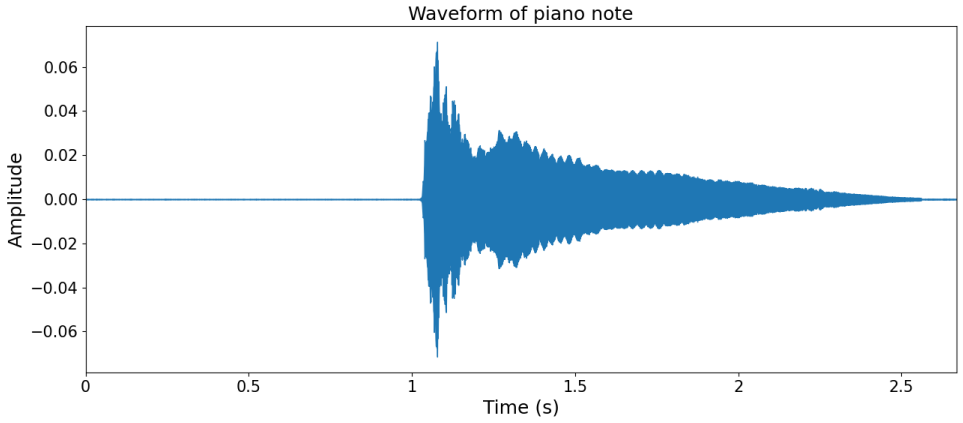


Figure 2.1: Recorded waveform of a single note played on the piano. For convenience, the x axis is converted from samples to seconds, and the y axis displays the amplitude normalized by the maximum possible value after quantization.

2.1.2 Signals in frequency

To make note identification possible, the first step is to define what makes one note sound different from another. Besides differences in duration, loudness and timbre[2], musical notes sound different because of their pitch. Roughly defined as "the perceptual attribute which allows the ordering of sounds on a frequency-related scale" [29], pitch is what makes us capable of distinguishing higher from lower notes.

This becomes much clearer in the case of pure tones, or in other words, for sounds which have a sine function as waveform. For them, frequency is a clearly defined property of the waveform and by listening to them it is possible to match a target sound to a frequency. One could imagine a guitar string playing the same note as the tone of a tuning fork. Even though the waveform of the sound coming from the guitar is not sinusoidal, we can match it to the almost pure tone of the tuning fork because they have the same pitch. This allows us to relate our perception of musical notes to the frequency of their waveforms.

[2]These are all auditory characteristics of a musical tone. Their definitions are beyond the scope of this study, but can be found in [29].

With all that in mind, it is suitable to think of a signal representation that can provide frequency information so that different notes can be clearly represented. The mathematical tool that allows us to go from time to frequency domain is called the Fourier transform, and it enables this transition because it performs the decomposition of a signal into sinusoids of different frequencies.

To understand the advantage of this, consider a pitched sound coming from an instrument. Depending on the type and construction of the instrument, there exist certain oscillations corresponding to frequencies different from the one belonging to the played pitch, which makes the resulting waveform a sum of many pure tone sinusoids. For example, when someone plays the lower $\boldsymbol{E}$ string on a guitar, the resulting tone actually has the pure sinusoid frequencies corresponding to pitches $\boldsymbol{E_2}$ (the written note pitch), $\boldsymbol{E_3}$, $\boldsymbol{B_3}$, and to other multiples of the lowest frequency, which is also called the fundamental frequency (f_0) of the note.

Mathematically speaking, the way the Fourier transform enables this decomposition is through rewriting our previously time-based function as a sum of sinusoidal functions of different frequencies. Similarly to the case when a vector in $\mathbb{R}^2$ is rewritten in another basis, the transformation from time to frequency can be seen as writing the original time function in another function space basis. For digital signals, in this case implying functions both discrete and finite, it is convenient to work with the discrete Fourier transform (DFT), a version of the Fourier transform that is discrete in both domains, meaning that it works with functions consisting of samples and decomposes them into equally spaced frequencies:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}, \tag{2.1}$$

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}kn}. \tag{2.2}$$

Equations (2.1) and (2.2) respectively define the operation to go from time to frequency, and from frequency to time, and the attentive reader will note their similarity to the basis change equations of traditional linear algebra [26, 30]. In them, $x[n]$ is a sample from our discrete time signal x, N is the total number of

samples in x, and the complex exponentials $e^{j\frac{2\pi}{N}kn}$ represent the sinusoidal basis function mentioned earlier.

The resulting discrete function X also has size N, and each one of its k indices is called a frequency bin. It is very important to observe that each associated frequency $\frac{2\pi}{N}k$ is given in radians per sample and not hertz or radians per second as pitch frequencies are usually measured. This is a consequence of sampling continuous-time signals, and in order to go back from a discrete frequency bin to a continuous frequency, one should use the formula

$$f = \frac{kF_\mathrm{s}}{N}, \tag{2.3}$$

where k and N are the same as before, f is the desired continuous frequency, and F_s is the sampling rate used in the analog-to-digital conversion.

The sampling rate, or sampling frequency, is the number of samples of the original signal obtained in one second. Therefore, since our discrete frequency $\frac{2\pi}{N}k$ is given in radians per sample, to make the conversion we can simply multiply by F_s to obtain the value in radians per second, and after that divide by 2π to get a frequency in hertz. We invite the reader to check Appendix A and the sections on sampling in [24, 3, 26] for a more rigorous derivation of this formula.

Applying the DFT to the audio signal example of the previous section yields the results seen in Figure 2.2. In it we can clearly see the frequency peak corresponding to the note played before (a $\boldsymbol{C_5}$ at 523Hz); however, information concerning the onset of the note is completely lost. In order to represent at the same time the information in both domains, we need a new tool capable of assigning a notion of time to the Fourier transform, as will be seen in Section 2.1.3.

2.1.3 The Short-Time Fourier Transform

Created in 1946 by Dennis Gabor, the short-time Fourier transform is a compromise between time and frequency representations [3]. It consists in dividing the original signal in chunks called frames and applying the Fourier transform separately to each one of them. In this way, the frequency content of each frame can be analysed independently, and both time and frequency information are easily displayed.

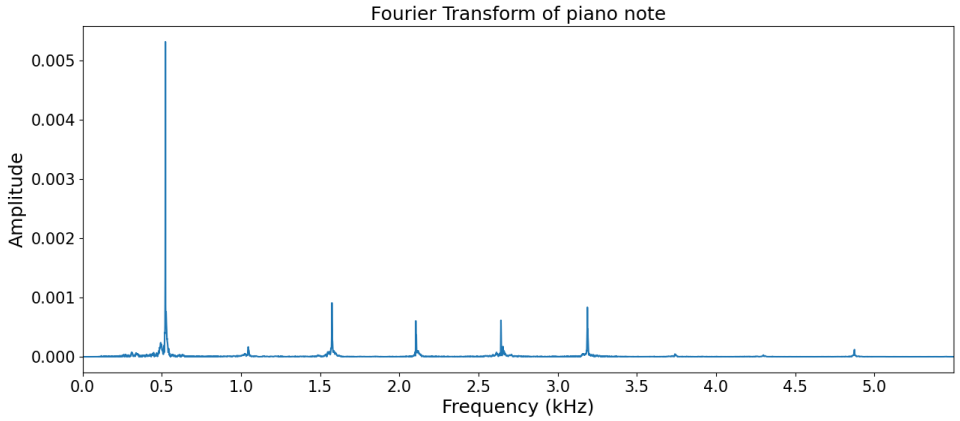


Figure 2.2: Fourier transform of the piano note of the previous example. It can be identified as a $\boldsymbol{C_5}$ due to its main peak at 523Hz, but other peaks corresponding to higher harmonics are also visible. For better visualization, the x axis is converted from bins to hertz and the y axis is scaled and normalized to show values coherent with the previous waveplot.

The visual representation of the STFT is called a spectrogram, and the one corresponding to the example used in the last two sections can be seen in Figure 2.3. With this image, we are finally able to identify with a single representation both the note's pitch and its onset, i.e. the time instant in which it starts.

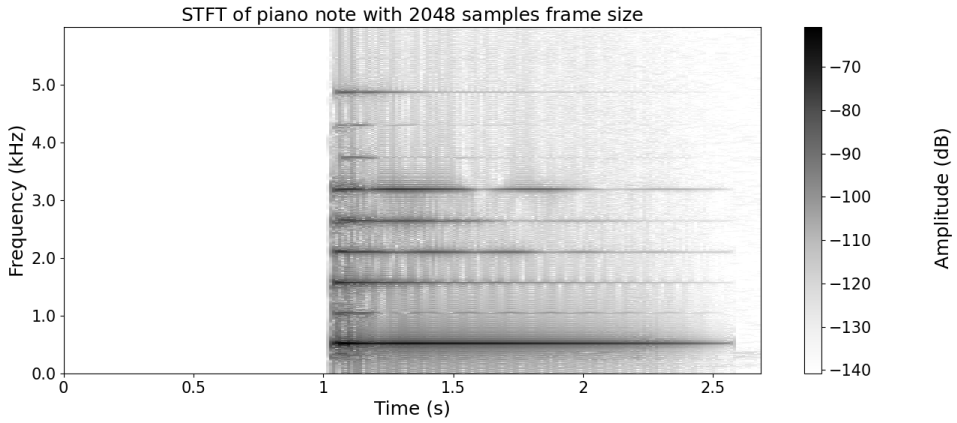


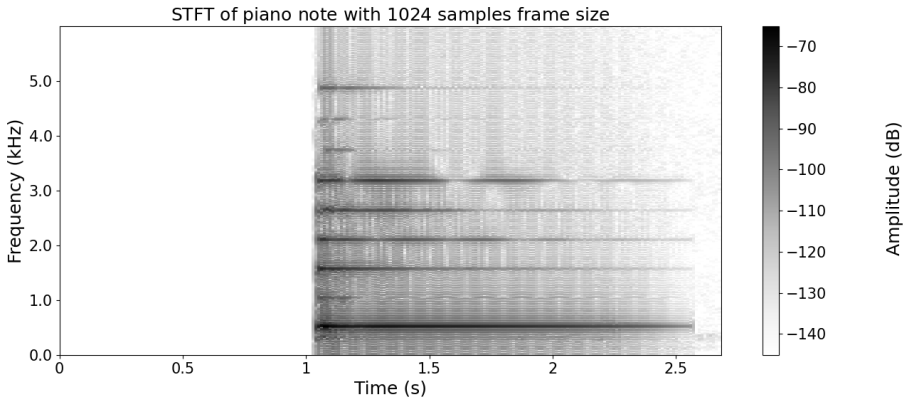
Figure 2.3: Short-time Fourier transform of single note played in a piano. This mixed representation allows to identify both the note frequency and onset. In this and in all other spectrograms unless stated otherwise, the y and x axes were transformed from bins and frames to respectively frequencies and time, and the color scale is displayed in decibels with the maximum possible frequency magnitude value after quantization used as reference.

However, displaying signals in a mixed domain representation such as the STFT comes with a few drawbacks. For the short-time Fourier transform, the most notable inconvenience is the time-frequency trade-off in terms of resolution.

In order to get to the STFT, all chunks of samples need to go through the DFT so that frequency content can be extracted, which causes temporal information inside a frame to be lost. If a sound wave contains two notes in a 1 second interval, but the frames used to build the spectrogram have a size equivalent to 2 seconds, then it is likely that the image will not clearly show the start of both notes separately.

Yet, by reducing the frames we are forced to perform the DFT on smaller signals, spreading frequency content across bins. One of the properties of the Fourier representation of signals is that short signals in time are wide in frequency, and vice-versa [24, 31]. As a consequence, whenever the STFT time frames are reduced, their frequency content becomes wider, thus occupying many bins.

(a)



(b)

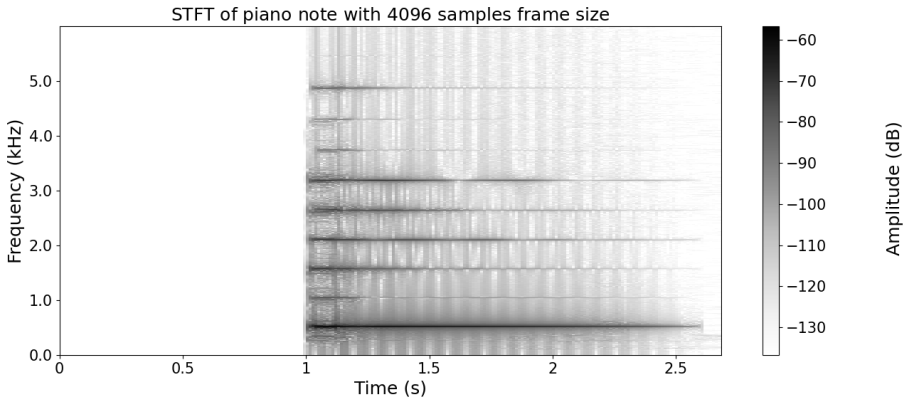
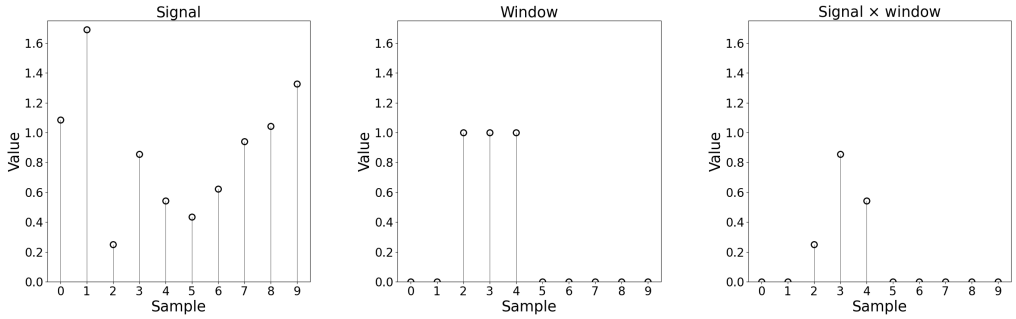


Figure 2.4: Spectrograms with different frame sizes for the same audio as before. (a) STFT with smaller windowing, increasing time resolution and frequency leakage. (b) STFT with larger frames, reducing frequency spreading but also time resolution.

This phenomenon can be seen on Figure 2.4, where we display the spectrograms for the same audio as before, but increasing (Figure 2.4b) and decreasing (Figure 2.4a) the frame size. Figure 2.4a has visibly better time resolution when compared to Figure 2.3, but frequency content is blurred across bins where it was not before. Conversely, Figure 2.4b is less blurry in frequency when compared to Figure 2.3, at the expense of having worse resolution in time.

The process of extracting the parts of the signal belonging to each time frame is called windowing, and it also causes distortions regardless of the time frame (window) duration. As shown in Figure 2.5, extracting the samples of a window can also be interpreted as multiplying the original signal by a rectangular function consisting of ones inside and zeros outside the time frame, which can add frequency content into the original signal's frequency spectrum. As a matter of fact, rectangular windows like the one described before are very rich in high frequency content due to the presence of the abrupt change between 0 and 1 and vice-versa, and so are also responsible for blurring the frequency content of the STFT.

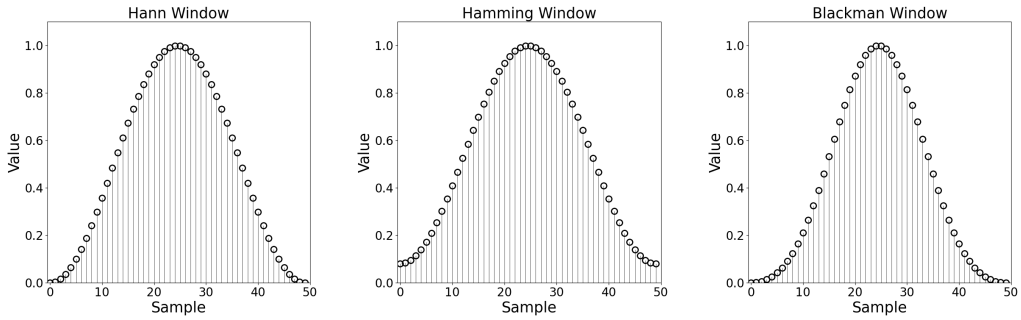


(a) (b) (c)

Figure 2.5: Windowing process for STFT construction. (a) Signal. (b) Window. (c) Windowed signal.

A common solution for this problem is changing the shape of the window — or in other words, weighting differently each sample within the time frame — so that these high frequencies have a smaller effect on the STFT. No window is capable of avoiding frequency distortion in the spectrogram, but smoother windows exhibit reduced leakage, at the cost of worsening frequency resolution. A few examples of window are shown in Figure 2.6.

In order to compensate for the fact that the samples are reduced in magnitude by their weighting, it is commonplace to overlap time frames so that the



(a) (b) (c)

Figure 2.6: A few examples of window types, all sized 50 samples. **(a)** Hann. **(b)** Hamming. **(c)** Blackman.

magnitude in all samples is restored. Overlapping also has the effect of allowing certain frequencies to appear in many time frames, which can partially prevent the loss of time information caused by large window sizes.

The STFT is the basic Fourier representation that is used to extract the information necessary for aligning different audio recordings, and parameters such as window and overlap sizes will be very important later on in Chapter 3 when we start discussing the alignment results. In the next section, we will explain how to go from frequency bins to notes along time, and lay the foundations on which we will build the audio alignment algorithms.

2.2 Audio features

While the short-time Fourier transform enables accessing music information simultaneously in time and frequency, it is very crude in the sense that, if used alone without further processing, it hardly allows the algorithmic use of music data.

Spectrograms contain information about all frequency content in a recording, including noise, instrument timbre, and tuning imperfections, all of which can make the alignment of recordings less robust if used directly as input to our algorithm. This is why it is important to extract features representative of the information we want to analyze, which in our case are the notes being played at each instant of time.

With the STFT as starting point, in this section we will show how to get pitch and note data to create a robust representation of music playing along time.

Assuming two different recordings of the same piece of music are available, our target here is demonstrating how they can be described in such a way that makes them comparable by an appropriate algorithm, despite the excess information in the spectrogram.

2.2.1 Log-frequency spectrogram

When computing the short-time Fourier transform as seen in Section 2.1.3, the result is always a matrix. Since we get K bins when applying the DFT to each one of the M time frames, it follows that the spectrogram χ of a discrete signal x must be a matrix where each value $\chi(m, k)$ corresponds to the frequency content of bin k during frame m.

Formally, remembering that windowing is multiplying a signal by a window function, and also using Equation (2.1), we can write χ as [3]

$$\chi(m, k) = \sum_{n=0}^{N-1} x[n + mH]w[n]e^{-j\frac{2\pi}{N}kn}, \tag{2.4}$$

where H, also called hop length, is the number of samples between subsequent frame starts, and is associated to the overlap by the relation

$$H = \lfloor (1 - O)L \rfloor, \tag{2.5}$$

with L equal to the number of samples in the window and O the percentage of overlap between windows.

In order to get rid of timbre and noise differences between representations of different recordings, the first thing to do is adapt the spectrogram to show pitches instead of bins. As we will show, even though each bin is associated to a continuous frequency, depending on the total number of bins, more than one of those frequencies may be associated to the same pitch, causing these small spectrogram differences.

Frequencies are related to bins through Equation (2.3), but finding the relation between pitches and frequency can be a bit more tricky. As we stated in Section 2.1.2, pitch is the perceptual attribute of sound that allows us to match it to a pure tone of determined frequency, but once again, unfortunately computers

are not able to do this matching by themselves. To solve this issue, the MIDI [6] standard was created as a way to associate note pitches and integers.

Since the second half of the XVIII$^{\text{th}}$ century, the most commonly used scale in Western music is the so-called equal tempered scale, in which notes are arranged in a geometrical series, with an octave being equivalent to doubling the frequency of a note. Counting sharps (or flats), an octave interval spans twelve notes, and therefore any two pitch frequencies can be related through

$$f_2 = 2^{s/12} f_1, \tag{2.6}$$

where f_2 and f_1 are the two pitch frequencies, and where s is the number of notes between them, also called the number of semitones in the interval.

Based on this, the MIDI standard serves as a linear scale connecting frequencies and pitches. Each note pitch is represented with a 7-bit unsigned integer — which means an integer belonging to $[0, 127]$ —, and central $\boldsymbol{A_4}$ is attributed to number 69. Considering that $\boldsymbol{A_4}$ is the concert pitch with an established frequency equal to 440Hz, by using Equation (2.6) we can get the frequency corresponding to a pitch assigned to a specific MIDI number by using

$$F_{\text{pitch}}(p) = 440 \times 2^{\frac{(p-69)}{12}}. \tag{2.7}$$

Using that, it is possible to group frequencies corresponding to the same central pitch to partially remove undesired information from the spectrogram. By defining for each pitch p a set P of bins associated to p as

$$P(p) = \{k : F_{\text{pitch}}(p - 0.5) \leq \frac{kF_{\text{s}}}{N} < F_{\text{pitch}}(p + 0.5)\}, \tag{2.8}$$

we can assign each spectrogram coefficient $\chi(m, k)$ to the pitch whose center frequency is closest to it by summing the squared magnitudes[3] of the bins associated

[3]The square magnitude of a spectrogram element can be seen as the energy contained in bin k at frame m.

to the same p, obtaining:

$$\mathcal{Y}(m,p) = \sum_{k \in P(p)} |\chi(m,k)|^2. \tag{2.9}$$

The resulting spectrogram $\mathcal{Y}$ is called a log-frequency spectrogram [3], because it represents frequencies grouped in pitches geometrically spaced, which means that the frequency axis is converted from a linear to a logarithmic scale during the pooling process.

Pooling is exemplified in Figure 2.7 for MIDI numbers $p = 68$ and $p = 69$, considering a spectrogram computed from DFTs with size $N = 4096$ for a signal sampled at $F_s = 44100$Hz. The curly brackets span the bins being mapped to the same pitch frequency, and it is interesting to see how the interval for k decreases with the MIDI number.

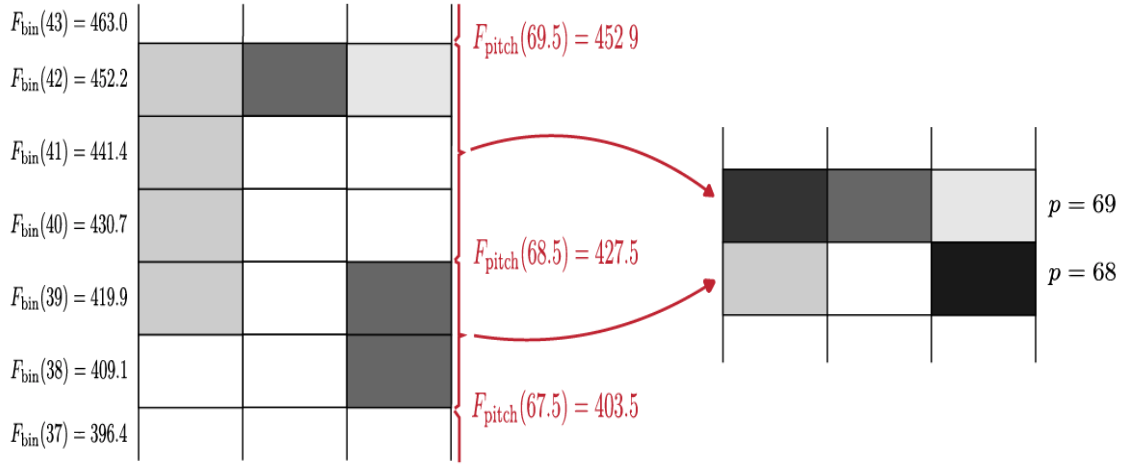


Figure 2.7: Pooling process for MIDI pitches $p = 69$ and $p = 68$, in a spectrogram created using DFTs with 4096 samples from a signal sampled at $F_s = 44100$Hz. F_{bin} corresponds to the continuous frequency associated with each bin and is given by Equation (2.3), whereas F_{pitch} is calculated using Equation (2.7). It is possible to see that bins with frequency between $F_{pitch}(p - 0.5)$ and $F_{pitch}(p + 0.5)$ are grouped together. Created based on [3].

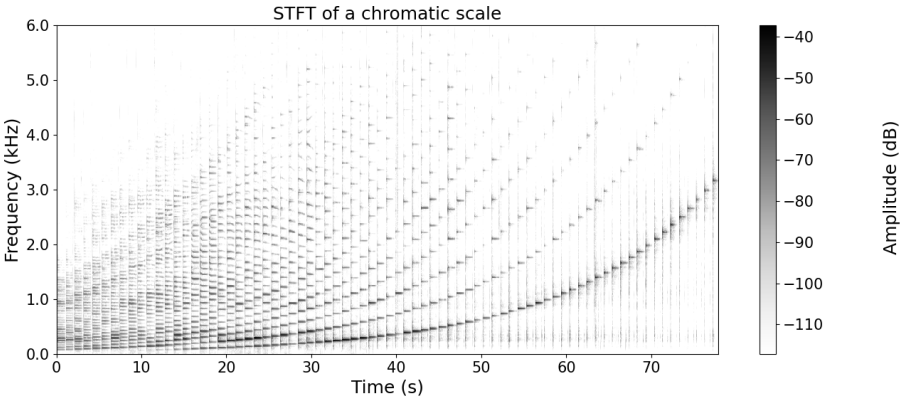
Since pitches are geometrically spaced in frequency, the lower we go on the MIDI scale, the closer the pitch frequencies become. It follows that the interval for acceptable k becomes smaller for lower p, which leads to sets with less elements.

Moreover, time-frequency resolution trade-off plays a large role in the pooling process due to the fact that very small windows can cause the frequency content

to spread beyond the interval between two pitches for low p. A more detailed discussion on the resolution issues related to the spectrogram and the log-frequency spectrogram can be found in [3].

Figure 2.8 shows both the spectrogram (Figure 2.8a) and the log-frequency spectrogram (Figure 2.8b) corresponding to a chromatic scale played from $\boldsymbol{E_1}$ to $\boldsymbol{G_7}$ on an electric piano. The first noticeable difference between the two representations is the frequency scale, with the exponentially separated notes of the chromatic scale appearing linearly distanced in the second image, but it is also possible to observe the resolution issues of pooling in the second image, where lower pitches exhibit thicker lines.

(a)



(b)

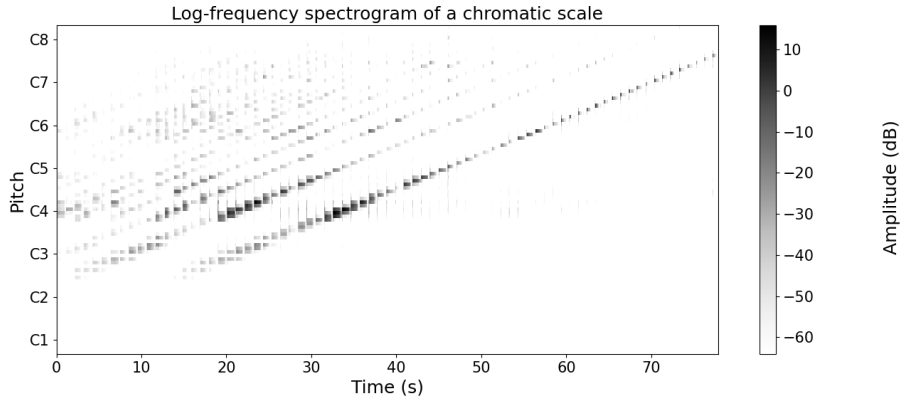


Figure 2.8: Comparison between spectrogram and log-frequency spectrogram. (a) Traditional spectrogram. (b) Log-frequency spectrogram created from the previous one through the pooling process described by Figure 2.7. For lower piano notes, the overtones normally contain most of the energy, which explains why fundamentals are not visible until the second octave.

2.2.2 *Chroma* features

The pitch based representation of the log-frequency spectrogram is able to reduce the unnecessary information in the STFT up to a certain point. Since bins are grouped by MIDI number, information is synthesized in pitches, and noise and timbre influences on the result are reduced. Still, some lack of robustness to these factors persists, as well explained in [3].

As seen in Section 2.1.2, notes played by instruments are generally composed of a fundamental tone associated with some harmonics, which are nothing more than multiples of the fundamental frequency. Since timbre is determined by the relative magnitudes of these harmonics (characteristic of each instrument), even if frequency information is pooled into conventional note pitches, there can still be a noticeable difference between two recordings of the same piece played on very different instruments.

An idea to add robustness to the representation is grouping certain harmonics according to some criterion, so that differences in their magnitudes related to the instrument can be smoothed out in a pooling process similar to the one used before. Pitches can be separated in two distinct perceptual components [3]: tone height, which determines the octave of a note, and *chroma*[4], which distinguishes notes in the same octave. It is natural then to conceive a pooling method that accumulates the magnitudes of pitches that share the same note name, i.e. *chroma*, regardless of their height [3]. This is why the resulting representation is also called a chromagram.

Formally, if we assign to each of the twelve musical notes an integer $c \in [0, 11]$, the chromagram can be calculated as

$$\mathcal{C}(m, c) = \sum_{p \in Q(c)} \mathcal{Y}(m, p), \tag{2.10}$$

with p representing a pitch MIDI number, $\mathcal{Y}$ being the log-frequency spectrogram calculated earlier, and Q being defined as the set of pitches sharing the same *chroma*:

$$Q(c) = \{p \in [0, 127] : p \mod 12 = c\}. \tag{2.11}$$

[4]The name *chroma* comes from the Greek word for color. This analogy relates to the fact that humans are able to perceive equal notes on different octaves as having the same "color", meaning that they feel the same despite having different tone height.

Representations based on *chroma* introduce a high level of robustness against timbre because they ignore the harmonic characteristics of an instrument. Since many harmonic pitches are grouped together, *chroma* representations for the same piece will look similar independently of how the instrument resonates multiples of the fundamental frequency.

The price, however, is irrecoverably losing tone height distinction. For example, if someone plays in a piano a song where the right hand plays a melody containing a $\boldsymbol{C_5}$, while the left hand accompanies with a C chord on the second octave, both $\boldsymbol{C_2}$ and $\boldsymbol{C_5}$ will be grouped into the same *chroma* bin, making melody and harmony indistinguishable.

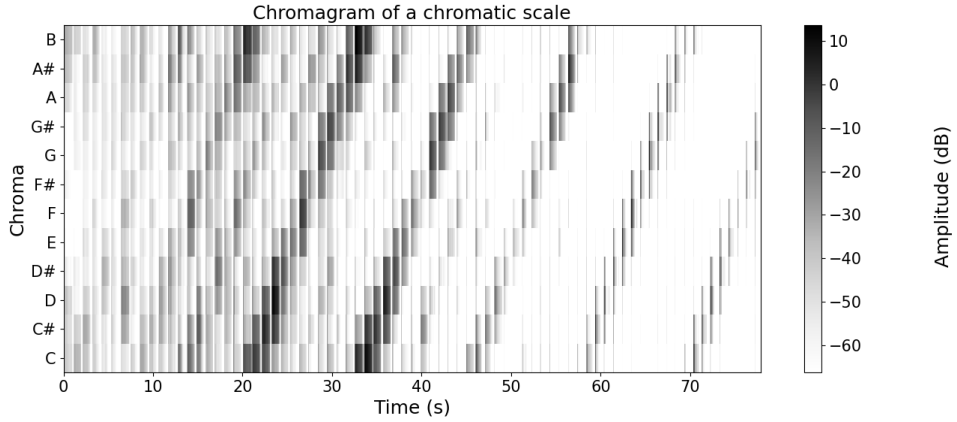


Figure 2.9: Chromagram of the same chromatic scale as before. Because pitches are considered the same regardless of tone height, the representation becomes cyclical along time.

Figure 2.9 shows the chromagram for the same chromatic scale of Figure 2.8. In it, the magnitudes at each time frame are more concentrated than in the previous representations, which shows how *chroma* features can highlight the notes being played, at the cost of not distinguishing between octaves.

Some harmonics are still visible, especially in the lower octaves due to acoustical characteristics of the piano[5], but overall this representation is much more robust than the previous two. By pooling bins into pitches, and pitches into *chromas*, noise and timbre influences are greatly reduced, which, as we will see later, can

[5]In the piano, the harmonics of bass notes typically have higher magnitudes than the fundamental frequency, making it hard to distinguish f_0 in this range.

even enable the comparison of two pieces played in two kinds of instruments or in multi-instrumental recordings.

2.2.2.1 Logarithmic compression

Up until now, all color scales used in the spectrogram and its variations in this study were in decibels, with the maximum magnitude value after quantization being used as the reference value. This decision was made for better visualization of the images, considering the large dynamic range of music signals, and the fact that logarithmic scaling reduces the distance between the highest and lowest elements of the STFT, thus avoiding dominance of the latter by the former.

Using log scales not only helps human visualization of representations, but also highlights certain important elements of the spectrogram. When using time-frequency representations as an input for alignment algorithms, just as with human vision, it may be that some relevant, but small, elements of the representation are obscured by those of greater magnitude.

However, without changing the reference value, the decibel scale is not flexible with relation to the degree of compression. This is why in [3], the author suggests using an alternative log scaling procedure, where a compressed chromagram Γ_γ is calculated using

$$\Gamma_\gamma(m, c) = \log(1 + \gamma \mathcal{C}(m, c)), \tag{2.12}$$

with γ being a positive constant.

In this way, values are displayed on a positive scale where the ratio between the largest and the smallest elements can be regulated by γ. The larger the constant, the closer the smaller values are to the larger ones, that is, the stronger the compression.

Figure 2.10 shows the advantages of using log scales for chromagram values, and also compares the decibel scale and logarithmic compression using different constant values. The dominance of certain elements in unscaled representations can be very clearly seen in Figure 2.10a, where the fundamental pitch of the note $\boldsymbol{C_2}$ hides all the other harmonics, thus underemphasizing potentially useful information. It is possible to see that there is not much difference between decibel scaling and logarithmic compression, but the contrast between Figure 2.10c and Figure 2.10d

indicates the flexibility of the latter, which enables adjusting the scale according to the data.

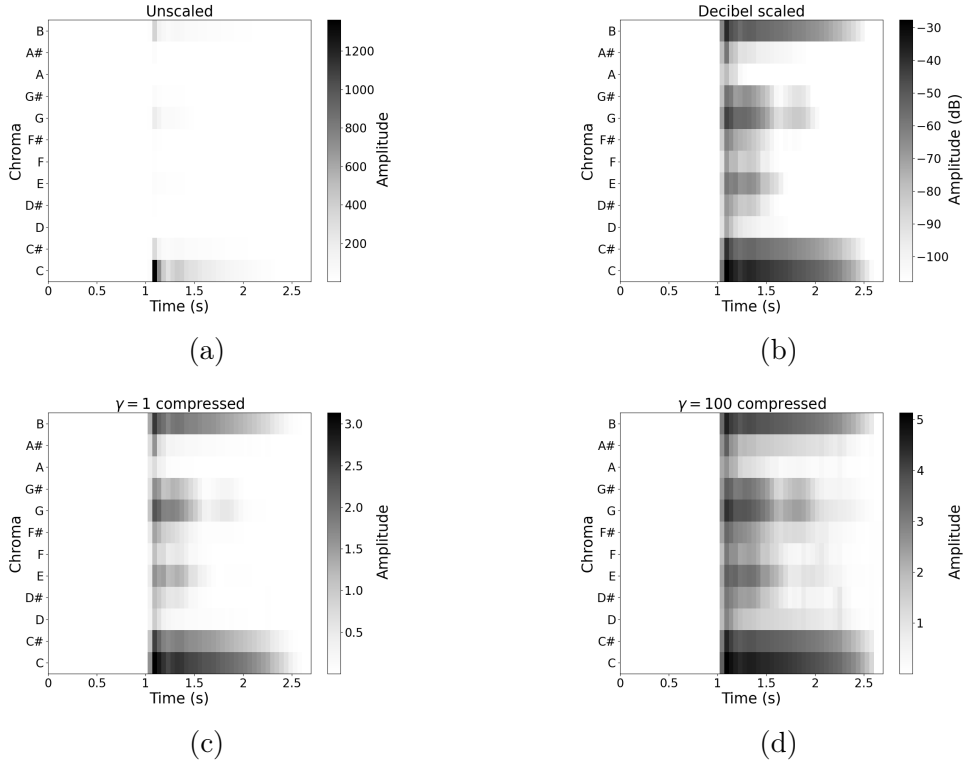


Figure 2.10: Chromagram of the single note recording used as example in Section 2.1 with different types of scaling. (a) Unscaled chromagram. (b) Decibel scaled chromagram with same reference as before. (c) Chromagram scaled with logarithmic compression using $\gamma = 1$. (d) Chromagram scaled with logarithmic compression using $\gamma = 100$.

2.2.2.2 Normalization

A final additional step that can be applied to the chromagram to add robustness to the representation is normalizing all frames after compression. While log scaling can help better handling audios with a wide range of magnitude values, it is unfortunately unable to account for changes in dynamics in music recordings.

When comparing two non normalized audio recordings of the same piece, volume differences between the performances can create undesired dissimilarities between their chromagrams. Dynamics can induce the opposite effect of logarithmic compression, which is overemphasizing irrelevant information. Therefore, normalization can be quite convenient depending on the recordings being compared.

Normalization consists of making all the chromagram frames have a unit norm by replacing each column vector m by $m/\|m\|$, where $\|m\|$ is the norm of m. This adds robustness to interpretation dynamics when comparing two audio recordings, because the chromagrams for both of them will include only elements with values between zero and one. It is common in this type of processing and in others to use the Euclidean norm, which is defined as the square root of the sum of squares of the vector elements, but other norms such as the Manhattan norm [32, 3] or the infinity norm [33, 3] can also be used.

Zero norm frames cannot be normalized, but a caveat suggested in [3] for *chroma* representations is that columns with very small norms should not be normalized either. When there is silence in an audio recording, *chroma* features are likely to be randomly distributed across all twelve possible values with small magnitudes. Hence, normalization in these cases would only emphasize irrelevant background noise.

The solution proposed in [3] is to define a positive threshold below which all frames are replaced by a unit norm vector with equal values for all *chromas*, instead of being normalized. Mathematically speaking, if this rule is used on the compressed chromagram of Equation (2.12), the result is

$$\Gamma_{\gamma,\varepsilon}(m,c) = \begin{cases} \Gamma_\gamma(m,c)/\|\Gamma_\gamma(m,c)\|, & \text{if} \quad \|\Gamma_\gamma(m,c)\| > \varepsilon \\ \overrightarrow{1}/\|\overrightarrow{1}\|, & \text{if} \quad \|\Gamma_\gamma(m,c)\| \leq \varepsilon, \end{cases} \tag{2.13}$$

where $\Gamma_{\gamma,\varepsilon}$ is the normalized and compressed chromagram, ε is the threshold mentioned before, and $\overrightarrow{1}$ is a vector of ones.

Figure 2.11 contains a normalized and compressed version of the chromagram of Figure 2.9 using $\gamma = 1$ and $\varepsilon = 10^{-3}$. Visualization — especially for higher octaves, which had smaller magnitudes — is greatly enhanced and octaves are better defined than in the previous image. Even tough notes before $\boldsymbol{C_2}$ are still problematic, possibly due to the harmonic characteristics of low piano notes, whose energy is mainly in overtones rather than in the fundamental, the image clearly shows how normalization and logarithmic compression can highlight key aspects of an audio recording.

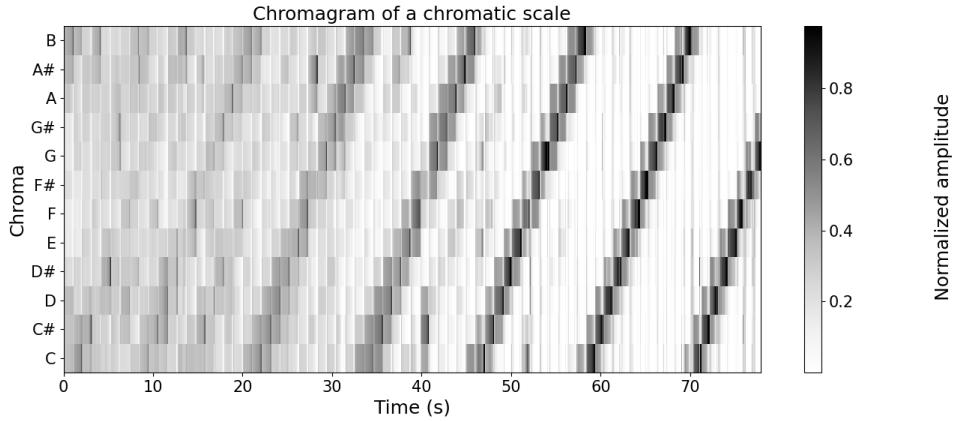


Figure 2.11: Normalized and compressed chromagram for the same recording used in Figure 2.9. In this version of the chromagram, it is possible to see how key aspects of the audio were highlighted thanks to the procedures described before.

The normalized and compressed chromagram is the final representation that is used as input to the alignment algorithms for the interpretation switcher. It is superior to the raw STFT in the sense that it highlights the key information we are interested in, which are the notes being played at each instant in time, while being robust to changes in the aspects that are not useful for alignment, such as noise, timbre, and dynamics.

The next section will show the fundamentals behind dynamic time warping (DTW), the chosen algorithm for finding equivalences between the frames of two different audio recordings. Starting from the concepts of distance and cost matrices, we will build the understanding of this algorithm and show the performance gains of one of its variants for our task.

2.3 Audio synchronization

Synchronization between two different music recordings is the core of the project. As we will see in Chapters 3 and 4, it is audio synchronization that enables switching between interpretations during audio reproduction, and also score following.

To synchronize recordings, the audio features extracted from the STFT are used as input to an alignment algorithm, which in other words is simply a technique for finding the equivalence between frames of two chromagrams. Being given two

interpretations of the same song, divided in N and M frames respectively, the final goal is finding out which of the M frames of the second is most similar to one of the N frames of the first, according to some criterion.

2.3.1 Dynamic Time Warping

The chosen algorithm for audio synchronization, dynamic time warping, is essentially a method for comparing matrices. Given two sequences of vectors, DTW outputs a number that corresponds to how distant they are from each other.

Dynamic time warping works by calculating distances between column vectors — in our case, the frames of the chromagram — and finding correspondences between them using a greedy algorithm that minimizes the total accumulated distance between the two sequences. DTW outputs this total accumulated distance, and it stands out in comparison with traditional matrix distances [34] in the sense that it is time dependent, meaning that columns vectors are considered sequential in time, with an influence of this progression over the output value.

For us, its most important aspect is the capability of finding equivalences between columns of two matrices while possibly reusing some of them along the (always forward) path. This enables us to take into account tempo difference between recordings, which is one of the features that make DTW well suited for audio synchronization.

2.3.1.1 Distances and cost matrices

As mentioned above, DTW is based on calculating distances between the columns of two matrices. Mathematically speaking, a distance, or metric, is a mapping which compares two elements of a set, and that also follows some rules, such as being positive definite, symmetrical, and respecting the triangle inequality[6].

For example, a common metric is the Euclidean distance, which is simply the Euclidean norm of the difference between the corresponding elements of two vectors. Another example is the metric used as standard for the audio alignment system: the

[6]The formal definition of distance is beyond the scope of this study, but more on that and on other mathematical analysis subjects can be found in [33]

cosine distance, defined as

$$c(x, y) = 1 - \frac{\langle x|y\rangle}{\|x\|\|y\|} \tag{2.14}$$

for two vectors x and y, with $\langle x|y\rangle = x^T y$ being their inner product [3, 35, 30].

It is a well known result from linear algebra [35] that if the angle between two vectors x and y is α, then $\cos\alpha = \frac{\langle x|y\rangle}{\|x\|\|y\|}$. So, an intuitive interpretation of cosine distance is that vectors are very similar if they are "pointing" in the same direction, since it is zero if they are parallel. This does not take into account the length of the vectors, which is an advantage for audio synchronization in the sense that it disregards dynamics, i.e., volume differences between the recordings will not affect this similarity measure.

Considering the distance between two chromagram columns as the cost of attributing them to each other, we can define a cost matrix $\mathbf{C}$ where each element represents a measure of the distance between two frames. Formally, supposing a distance d and two chromagrams X and Y with frames $(x_1, x_2, ..., x_N)$ and $(y_1, y_2, ..., y_M)$, we have that

$$\mathbf{C}(n, m) = d(x_n, y_m). \tag{2.15}$$

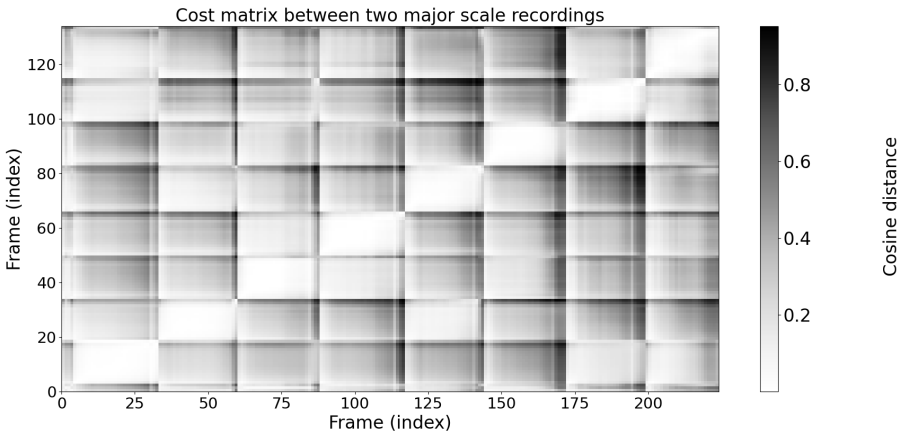


Figure 2.12: Cosine distance cost matrix between two C major scale recordings starting at $\boldsymbol{C_3}$. For convenience, time here is shown in frames instead of seconds since the matrix compares vector frames of each recording.

Figure 2.12 shows the cost matrix of the comparison between two recordings of a C major scale played in a piano. The first note of both recordings is $\boldsymbol{C_3}$, and the only notable difference between them is tempo. Since they both show the same

note progression over time, the lowest values are more or less around the diagonal of the image, with small vertical patches appearing when two frames of the slower recording are musically equivalent to a single frame of the faster one.

2.3.1.2 DTW distance

The goal of DTW is using the cost matrix to determine the smallest total accumulated distance between two chromagrams. Putting it more simply, this is equivalent to finding the sequence of cells $\mathbf{C}(n, m)$ in the cost matrix which leads to the final frame of both inputs — $\mathbf{C}(N, M)$, the top-right cell — and that also minimizes the sum of the cells' values.

Graphically, this means finding the sequence of elements in the cost matrix with the lowest values in the color scale. In Figure 2.12, for example, this would be the path of white cells spread around the diagonal of the image that we pointed out before.

There is a dynamic programming[7] algorithm for determining the smallest accumulated distance between any two frames n and m. If we store the accumulated distances between all frames in a matrix $\mathbf{D}$, whose values $\mathbf{D}(n, m)$ represent the minimal sum of costs needed to get from $\mathbf{C}(1, 1)$ to $\mathbf{C}(n, m)$, then the accumulated distance to any frame can be calculated using [3]

$$\mathbf{D}(n, 1) = \sum_{i=1}^{n} \mathbf{C}(n, 1) \text{ for } n \in [1, N], \tag{2.16}$$

$$\mathbf{D}(1, m) = \sum_{i=1}^{m} \mathbf{C}(1, m) \text{ for } m \in [1, M], \tag{2.17}$$

$$\mathbf{D}(n, m) = \mathbf{C}(n, m) + \min \begin{cases} \mathbf{D}(n-1, m) \\ \mathbf{D}(n, m-1) \\ \mathbf{D}(n-1, m-1) \end{cases} \quad \text{for } (n, m) \in [2, N] \times [2, M], \tag{2.18}$$

with N and M being the number of frames in the two sequences compared.

[7] Dynamic programming [36, 37] is a coding technique that consists of dividing a problem into smaller subproblems to solve it. It differs from the standard divide-and-conquer coding strategy because it caches intermediate results in a well suited data structure, making it more efficient.

The proof that this algorithm does return the optimal accumulated distance between any two frames can be found in [3], but Figure 2.13 shows an intuition as to why it is a reasonable approach to the problem. By taking only the smallest accumulated value in neighboring cells and adding it to the cost $\mathbf{C}(n, m)$, the DTW algorithm uses recursion to work greedily towards minimizing the value $\mathbf{D}(n, m)$.

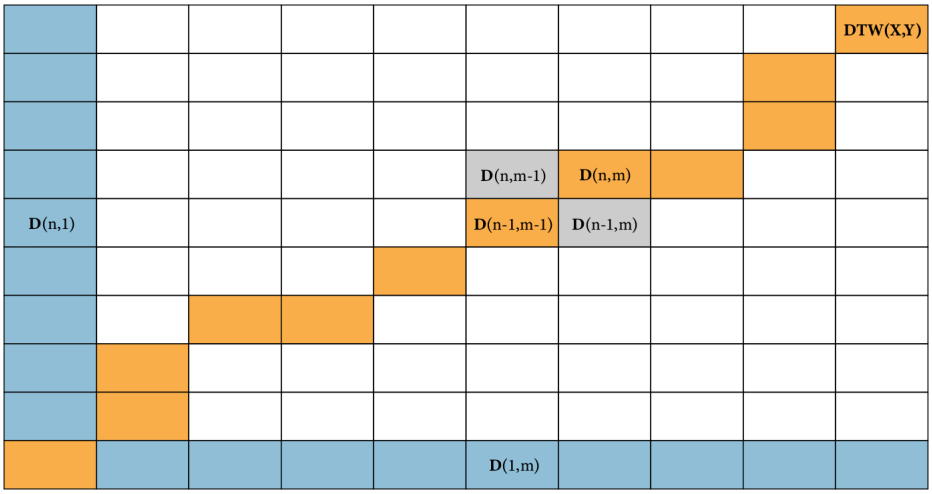


Figure 2.13: Dynamic time warping uses a greedy approach to achieve the minimum accumulated distance. The smallest neighboring accumulated value to the left is always added to $\mathbf{C}(n, m)$ to obtain the lowest possible cost until cell (n, m). The final cell $\mathbf{D}(N, M)$ is also called the DTW distance between two matrices X, and Y. In orange the sequence of cells with smallest accumulated cost. Based on [3].

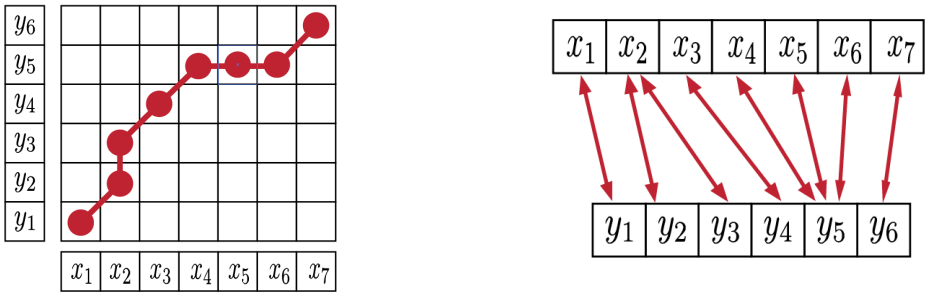
Furthermore, choosing only cells to the left of $\mathbf{D}(n, m)$ to accumulate imposes the very desirable constraint of not being able to "see the future". When calculating the accumulated cost to get to frames n and m, we should not be allowed to look at the accumulated costs of reaching $n + 1$ and $m + 1$, as that would violate the time dependency of the DTW. If this were the case, then the sequential aspect of the DTW would be gone, because a frame considered to be further located in time could affect previous choices.

Despite not respecting the conditions necessary for being a metric [3], the total accumulated distance between two sequences $\mathbf{D}(N, M)$ is often called the DTW distance, since it is a measure of similarity between two matrices. For audio alignment purposes, however, it is more important to find the sequence of cells used to reach $\mathbf{D}(N, M)$ than its value.

2.3.1.3 Warping path

The rules used to create the accumulated cost matrix $\mathbf{D}$ can also be used to backtrack the sequence of cells with minimal total accumulated cost. This sequence, which is also sometimes called the warping path, contains the equivalence between frames mentioned in the beginning of this section.

The interpretation leading to this can be seen in Figure 2.14. Figure 2.14a contains an illustration of a matrix $\mathbf{D}$ with a warping path marked on it. Because the marked sequence shows the path with minimal total cost, it follows that, if the frames with indexes on the path were paired, then the sum of the distances between pairs would be minimal. From a musical standpoint, this should be the equivalent of saying that if frames were attributed to each other as shown in Figure 2.14b, their audible difference would be the lowest possible, since the pairs are maximally similar.



(a) (b)

Figure 2.14: (a) Illustration of a warping path between two musical recordings. (b) Musical equivalence between frames based on warping path. Figures based on [3]

Backtracking to find the warping path is incremental, and it works as explained in [3]: starting from $q_L = (N, M)$, add the index of the smallest available neighbor between the ones allowed in Equation 2.18 to the beginning of a list meant to store the warping path, and stop when the element $q_1 = (1, 1)$ is reached.

Formally, this means that for all cells q_ℓ of indices $\ell = L, L-1, ..., 1$, the previous cell in the warping path can always be found using

$$q_{\ell-1} = \text{argmin}\{\mathbf{D}(n-1, m), \mathbf{D}(n, m-1), \mathbf{D}(n-1, m-1)\}, \tag{2.19}$$

unless $n = 1$, or $m = 1$. In these cases we are on the first frame of one of the chromagrams and the only way to reach $q_1 = (1, 1)$ is through:

$$q_{\ell-1} = (1, m-1), \quad \text{for} \quad n = 1, \tag{2.20}$$

$$q_{\ell-1} = (n-1, 1), \quad \text{for} \quad m = 1. \tag{2.21}$$

The lowest cost cell sequence is then given by $Q = (q_1, q_2, ..., q_L)$. It is important to guarantee both the start at $q_1 = (1, 1)$ and the end at $q_L = (N, M)$, which ensure that the path will connect the beginning and end of both chromagrams, and also follow strictly the neighborhood restrictions stated before, thus preventing musical equivalence with past frames.

Figure 2.15 shows the same cost matrix of Figure 2.12 with the warping path between the two recordings drawn in red, and we can verify that the graphical intuition for the DTW holds. The warping path follows the lowest values in the cost matrix, which gives the more or less diagonal sequence to which we called attention earlier.

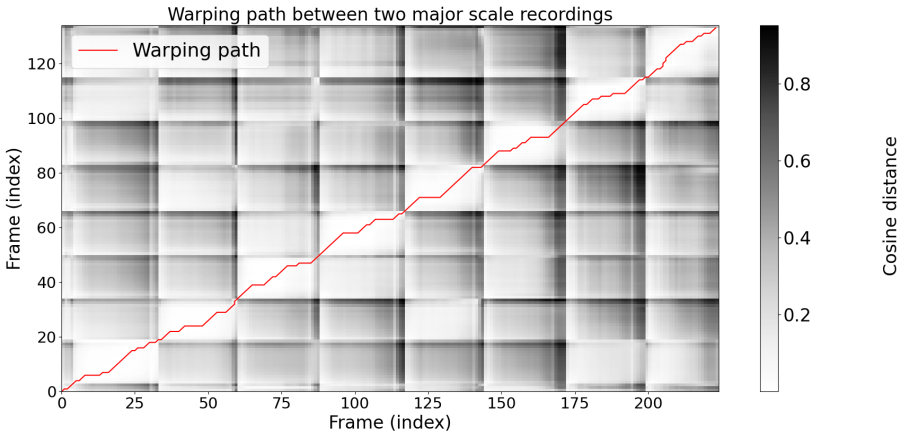


Figure 2.15: Warping path between the same C major scale recordings used in Figure 2.12. It is possible to verify that the optimal equivalence between frames goes through the diagonal path mentioned earlier.

2.3.2 DTW variants

In the literature, there are variants to the original dynamic time warping algorithm that are capable of improving audio alignment performance in some cases.

They provide a modified warping path that corresponds to the smallest accumulated cost between two chromagrams considering modified constraints. In the following sections, the variants implemented in this study are presented, and in Chapters 3 and 4, we will demonstrate why they bring significant enhancements to the system.

2.3.2.1 DTW weighting

The first modification, and possibly the most important for this project, is weighted DTW, a variant that slightly changes the creation of the $\mathbf{D}$ matrix by using

$$\mathbf{D}(n,1) = \sum_{i=1}^{n} w_{\mathrm{h}}\mathbf{C}(n,1), \quad \text{for} \quad n \in [1,N], \tag{2.22}$$

$$\mathbf{D}(1,m) = \sum_{i=1}^{m} w_{\mathrm{v}}\mathbf{C}(1,m), \quad \text{for} \quad m \in [1,M], \tag{2.23}$$

$$\mathbf{D}(n,m) = \min \begin{cases} \mathbf{D}(n-1,m) + w_{\mathrm{h}}\mathbf{C}(n,m) \\ \mathbf{D}(n,m-1) + w_{\mathrm{v}}\mathbf{C}(n,m) \\ \mathbf{D}(n-1,m-1) + w_{\mathrm{d}}\mathbf{C}(n,m) \end{cases}, \tag{2.24}$$

instead of Equations 2.16–2.18.

Here the coefficients w_{h}, w_{v}, and w_{d} are real positive numbers that act as local weights, making the choice of each of the three directions more or less costly. In [3], for instance, the author remarks that using $w_{\mathrm{h}} = 1$, $w_{\mathrm{v}} = 1$, and $w_{\mathrm{d}} = 2$ could neutralize the classic DTW's tendency to choose diagonal paths. The intuition is that a diagonal move is composed of a horizontal and a vertical step, and thus should be weighted to cost twice as much as the other directions.

In Figure 2.16, the warping path calculated using the weighted DTW proposition of [3] can be seen. Here diagonal paths are multiplied by twice the weight of horizontal and vertical steps in the accumulated cost matrix, which results in many more horizontal and vertical patches in the optimal frame equivalence.

In Chapter 3 we will see that this is the opposite of whats is needed for switching between interpretations. Because we are aligning two pieces that should be, to some extent, similar, the diagonal direction represents the shortest (and most

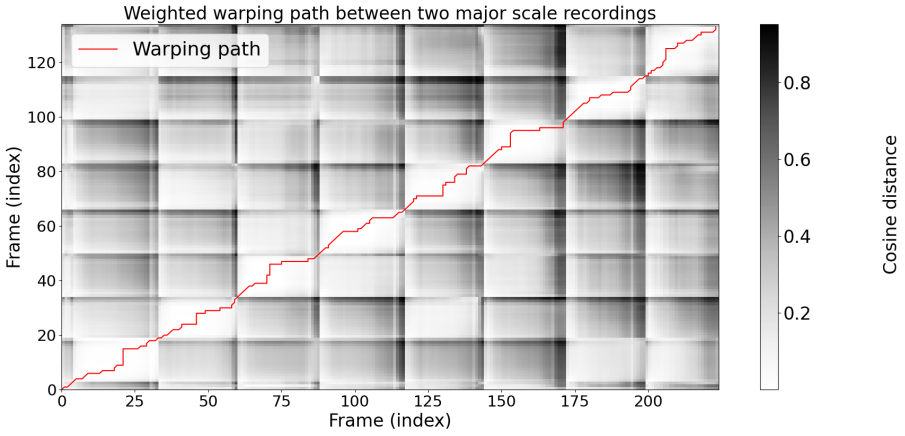


Figure 2.16: Warping path between the C major scale recordings of Figure 2.12, but this time using multiplicative weights to neutralize the diagonal tendency mentioned in [3]. In the accumulated cost matrix, diagonal steps here have twice the cost of horizontal and vertical steps.

natural) path to align them. Instead, changing the weights to boost the diagonals can be a valuable tool in finding the most significant musical alignment.

2.3.2.2 Different step sizes

Another variant introduced in [3] changes the neighboring conditions of Equation 2.18. Rather than only allowing the three closest past cells to be accumulated as seen in Figure 2.13, this version of the DTW changes the algorithm step size, that is, it allows neighbors farther away to be considered in the minimum operation.

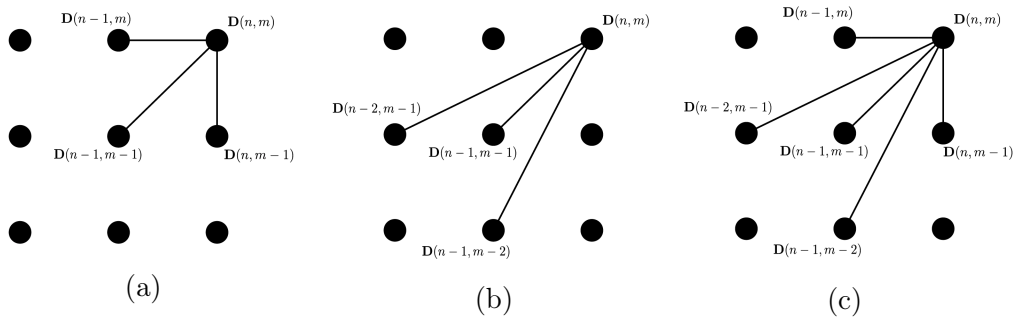


Figure 2.17: Different step sizes proposals. Images based on [3]

For example, if the step size condition of Figure 2.17b is chosen instead of the standard DTW (Figure 2.17a), then Equation 2.18 becomes

$$\mathbf{D}(n,m) = \mathbf{C}(n,m) + \min \begin{cases} \mathbf{D}(n-2,m-1) \\ \mathbf{D}(n-1,m-2) \\ \mathbf{D}(n-1,m-1), \end{cases} \tag{2.25}$$

and if Figure 2.17c is picked, the new condition is

$$\mathbf{D}(n,m) = \mathbf{C}(n,m) + \min \begin{cases} \mathbf{D}(n-1,m) \\ \mathbf{D}(n,m-1) \\ \mathbf{D}(n-2,m-1) \\ \mathbf{D}(n-1,m-2) \\ \mathbf{D}(n-1,m-1). \end{cases} \tag{2.26}$$

With the correct step size choice, this variant has the advantage of avoiding long horizontal or vertical patches in the warping path. If this happens, it means that many frames of one interpretation are being musically attributed to a single frame of the other. While this is not a problem for a small number of frames, since it can occur naturally due to differences in tempo as mentioned earlier, it is very troublesome when the patches are long, because it will generate paths that are optimal in the sense of cost, but that are not well fit for switching and following, as we will see in Chapters 3 and 4.

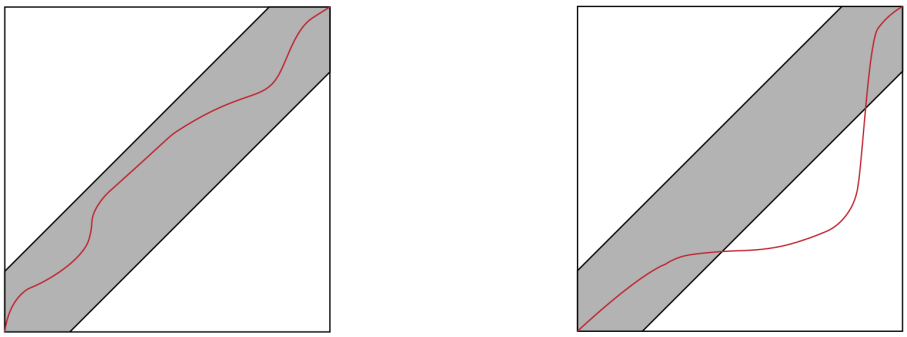
As mentioned earlier, in the rest of this study we will further explore this subject and see that weighting can also be used to solve this issue, with more cost being given to horizontal and vertical neighbors, using an intuition opposite to that proposed in [3].

2.3.2.3 Global constraints

The last variant that is implemented in this project consists in restricting the region that can be used for calculating the warping path. This is called globally

constraining the DTW, because it limits the equivalences to a set of columns in both representations, regardless of neighboring conditions or weighting.

Many different global constraints exist [3], but the only one that is present in this work is the one that is known as the Sakoe-Chiba band. It restricts the set of possible warping paths to the ones located inside a diagonal band of fixed width.



(a) (b)

Figure 2.18: Two warping paths considering the Sakoe-Chiba band global constraint. (a) Allowed path. (b) Prohibited path. The quantity corresponding to half of the band width is sometimes called band radius, and it is and additional parameter to set up when using this version of the DTW.

Figure 2.18 exemplifies the idea of globally constraining with the Sakoe-Chiba band. The two illustrations show an allowed and a prohibited path to give the graphical idea of restraining the area of possible warping paths.

Using global constraints like this one can improve the system performance because the band forces paths to stay reasonably diagonal, avoiding the problem of long horizontal or vertical patches. Furthermore, as can be seen in [3], banding can significantly reduce the number of calculations needed to compute the DTW, as long as the band width is much smaller than the number of frames of the longest recording.

However, adjusting the band width can be difficult, as a constraint that is too restrictive might exclude the desired warping path from the set of allowed paths. Conversely, choosing too big a value for the the width would have no effect whatsoever because the allowed path set would remain unchanged.

These variants add a layer of parameters to the DTW, which have to be adjusted for best results just like the ones for the audio features. This process is

hard, and can be dependent on the music recordings being aligned. For this reason, Chapters 3 and 4 of this study are dedicated to the two main parts of the web app, switching recordings and score following using synchronization, and to how each of the variables used in the last sections can impact these two activities.

Chapter 3

The Interpretation Switcher

The first of the two main blocks of the project is the interpretation switcher. It is the part responsible for ensuring navigation between audio recordings, in a way that feels musically seamless to the user. Supposing we have two recordings of the same piece by different artists, the goal of the interpretation switcher is making it possible to start playing the second one from the same point as the first one, in real time, and without needing to pause and find the musical equivalence between the interpretations by ear.

For this task, it is crucial to make sure that the alignment found by the DTW is optimal not only in the sense of cost, but also musically. This requires testing interpretation switching on many different audio recordings, and tweaking the parameters to find the values that are best suited for each situation.

In this chapter, we will see how the interpretation switcher is implemented in the web app, to understand exactly which parameters are available and how to change them if needed, as well as some experiment results — shown here in the form of DTW alignment paths drawn over cost matrices — so that a few heuristics for choosing parameters according to the recordings can be explained.

3.1 Implementation

Originally, this work began as a scientific initiation fellowship dedicated to trying to reproduce the results shown in [3], and most of the audio feature extraction and alignment procedures explained in Chapter 2 were implemented in MATLAB

code [38] to assess the viability of using the DTW as a tool for audio switching during playback.

When the project became a term paper aiming to create a viable product using these techniques, it became very clear that a tool more flexible than MATLAB would be needed. Python was the chosen programming language, both for its capability to produce all sorts of user interfaces — including web based ones — but also because of the availability of an excellent package for music and audio processing: librosa [23, 39].

3.1.1 The librosa audio processing package

Developed by researchers at LabROSA, the Laboratory for the Recognition and Organization of Speech and Audio of the University of Columbia in New York, librosa is a collection of useful tools for music information retrieval. It conveniently includes functions for loading audio files, an implementation of the DTW exactly as explained in [3], and also an alternative method for *chroma* feature extraction using a filter bank [39].

In their version, the chromagram is constructed straight from the spectrogram by multiplying it by a matrix of dimensions $12 \times M$, with M being the number of STFT bins, and where the value of any given element (i, j) is positive only if bin j corresponds to a frequency associated with *chroma* i. For example, if a DFT of 4096 samples is used to build the spectrogram for an audio recording using Equation 2.3, we find that bin 41 will correspond to a frequency of 441.4Hz, which has $\boldsymbol{A_4}$ as its closest pitch. This means that in the filter bank matrix, the element $(9, 41)$[1] will be nonzero, because bin 41 is associated with an A note.

The general idea can be seen in Figure 3.1, where it is possible once again to observe both that pitch classes are cyclical in frequency, and that pitch frequencies are geometrically spaced from one another. An important detail is that the magnitude of the (i, j) values is inversely proportional to the number of bins associated to each pitch. Lower notes span a smaller number of bins, which requires larger

[1]Considering matrix indices going from 0 (C) to 11 (B), which results in A being associated to number 9.

values to compensate for the natural accumulation caused by matrix multiplication in higher notes occupying many more bins.

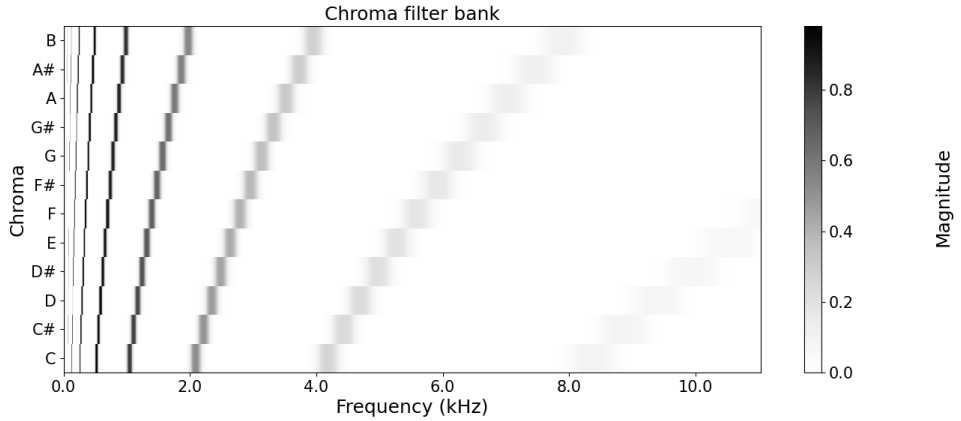


Figure 3.1: Filter bank used for *chroma* feature extraction in librosa considering a sampling frequency $F_s = 44.1$kHz, and a DFT of 4096 samples. Here the x-axis was converted from bins to frequency for convenience.

The package's load tools for reading audio files are compatible with most audio formats found on the web[2], but the interpretation switcher was only tested in *.wav* files. This type of audio file is the pure sound wave after analog-to-digital conversion, meaning that there is no loss due to compression, and no decoding operation to be made in order to read them, as is the case of lossy and lossless compression formats, such as *.mp3* and *.flac* respectively.

Considering this, and that the final goal of the project is developing a minimally functional application for navigating between audio recordings, the project decision to test and use the switcher only with *.wav* files is justified, so that any potential drawbacks related to compression losses or the way librosa handles decoding for lossless compression formats can be avoided. However, the interpretation switcher handles other subtle audio file differences, such as accepting both mono and stereo files[3], and dealing with audios with uncommon sampling frequencies, as long as all interpretations of the piece being analyzed are sampled at the same rate.

[2]At least with all file formats compatible with the SoundFile Python package, which is the basis for those load functions. See [40, 41] for a complete list of accepted formats.

[3]Mono files have a single audio channel, whereas stereo files have left and right channels, which can have different instruments recorded in them. To handle this, we convert all stereo files to mono by adding the two channels and dividing by two.

3.1.2 Audio alignment workflow

As stated in the previous section, the interpretation switcher is heavily based on librosa functions. Ignoring for now the user interface aspect, which will be seen in more detail in Chapter 5, the process for navigating through different tracks of the same piece during playback is as follows:

1. the user gives the audio files containing the recordings that will be reproduced with seamless navigation;

2. a chromagram is extracted for each one of them, using the librosa convenience function *chroma_stft()* with parameters provided by the user;

3. the chromagrams are compressed and normalized, using values for γ and ε provided by the user;

4. the warping path between all pairs of recordings is found using the *dtw()* function from librosa, once again with user defined parameters;

5. the equivalence between frames of each pair of recordings is written in a Python dictionary [22], which is then passed to the user interface that plays the audios, along with the sampling rate used in the files;

6. when switching between recordings is required, the user interface uses the playback time and sampling frequency of the current interpretation to calculate the current frame, and then checks the equivalence dictionary to know exactly where to resume playback in the desired interpretation.

Figure 3.2 shows a diagram containing all the steps mentioned above in the form of blocks that show the information flow for the interpretation switcher. All the convenience functions from librosa are wrapped in custom-made functions which appropriately pass the parameters according to the workflow in the image, and are contained inside the diagram blocks. Both the parameters provided by the user and the structure of the dictionary containing the frame equivalences are explicitly shown next to the corresponding blocks or arrows.

In the final version of the interpretation switcher, the user can choose the following audio feature parameters: window length, number of samples used in DFTs

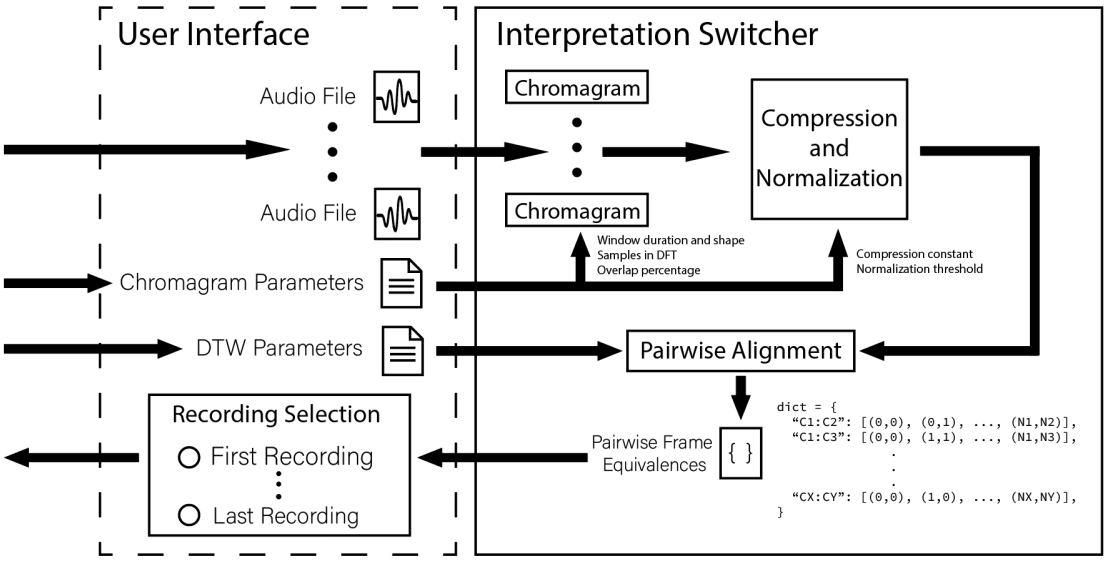


Figure 3.2: Diagram showing how the interpretation switcher works. The inward arrows outside the boxes represent user input and the outward arrow the audio playback. The structure of the dictionary containing frame equivalences is displayed in text next to the corresponding block, where X and Y are respectively the indexes for chromagram X and chromagram Y.

for the time-frequency representation, window function, overlap percentage, normalization norm and threshold, and compression constant. Concerning the DTW, weighting, calculating alignment with diferent distance functions, and using global restrictions are available as possible options for the user, but no changes to the traditional set of neighbor frames of Equation 2.18 are allowed.

3.2 Experiments

To verify that the switcher met all the requirements we have seen earlier, a number of experiments were conducted. They consisted simply in aligning different interpretations of the same song, while trying to hear out any musical errors caused by the DTW, but for some interesting cases, the warping paths were also drawn over the cost matrix of the alignment to try to visualize possible mistakes in frame equivalence. These special examples will be explained here in this subsection, where we will briefly describe the characteristics of each recording, before analyzing the resulting warping path.

It is important to observe that the DTW has some limitations that restrain the scope of the experiments we can do. For example, since the DTW is a method

heavily dependent on the sequence of time frames in the two chromagrams, it follows that there can be no difference in the music structure of the recordings. If, for example, one of the interpretations of a piece contains an additional bridge, or coda, then this part will have no correspondence on the second recording, leading to a musically meaningless warping path. This is also true for solos, or improvised sections, which are very common in certain styles, like jazz and rock.

Also, transposing a musical piece causes the chromagram to shift upwards or downwards of a few half-tones [3], which can make frames very dissimilar depending on the distance metric used. This affects the total cost, possibly leading once again to warping paths that are not musically useful, and therefore transposition must be avoided while using the interpretation switcher[4].

Because of this, the experiments shown here are restricted to classical music, and to pieces where the key and structure are the same, regardless of interpretation differences. Classical music has the advantage of being very well documented, with music sheets widely available on the internet, and also of being more rigid in form, which often ensures the constraints mentioned above are respected.

3.2.1 Musical recording database

Having this in mind, most of the testing done on the interpretation switcher, and in this project in general, was made using the set of twenty-four Preludes for piano belonging to Frédéric Chopin's *opus* 28 [42]. They have the advantage of being short pieces very different from each other, thus covering a wide range of tempos and dynamics. Moreover, they have been recorded by many pianists since the early twentieth century, allowing us to experiment with varied playing styles and recording conditions.

The scores for the preludes can be found for free in the Musescore catalog [43], and music notation software like Finale, Sibelius, or Musescore [18, 17, 16] can be used to convert it to MIDI, so that a version without interpretation traits can also be analyzed. Other recordings by famous pianists of the twenty-four preludes were

[4]This is not an unavoidable issue, but a feature to correct this was not implemented here. If the musical work's keys are provided, it is possible to shift the chromagram to transpose one of them to match the other. Similarly, for recordings that are not in concert pitch, shifting can be done to correct this, as long as a reference is given, or estimated [3]

kindly taken from the personal collection of professor Luiz Wagner P. Biscainho, who is a classical music enthusiast besides being a signal processing teacher and researcher.

3.2.2 Heuristics for parameter selection

As we mentioned earlier, the quality of the alignment resulting from the DTW is heavily dependent on the values of the parameters used in audio feature extraction, and also in the algorithm itself. Here we will see a couple of examples that illustrate this statement, and also describe a few heuristics that helped finding musically significant alignment paths for different types of recordings and songs.

3.2.2.1 Unbalanced weighting

The first of these experiments is the comparison through alignment of two recordings of the seventh Prelude of Chopin's *opus* 28, one by Brazilian pianist Nelson Freire, and the other by his Italian colleague Maurizio Pollini.

The latter is known for his intellectual and relatively formal approach to music, with very conservative interpretations relying heavily on what is written in the music sheets, allowing little room to dynamic and tempo variations not explicitly indicated by the composer originally. Freire, on the other hand, while certainly not a melodramatic musician, is much more liberal in the sense of using unwritten variations in tempo and dynamics as a way of expressing himself.

The pair, along with Prelude No. 7's characteristic pauses and pedal notes, makes for an interesting example of the musical equivalence found by the DTW. The repetition of long notes associated to the interpretation difference between the two pianists creates tricky similarities among parts of the piece, which sometimes leads to misalignment.

Figure 3.3 shows the warping path over the cost matrix of the comparison between these two versions of Chopin's piece. Being small recordings lasting no longer than one minute, a standard Hann window of 4096 samples with 50% overlap was used to calculate the chromagram, which was then compressed with a log-constant of 10 and normalized using $\varepsilon = 0.0001$. In this first experiment, the

standard DTW was used; neighboring constraints were limited to the closest frames, and no weighting was applied.

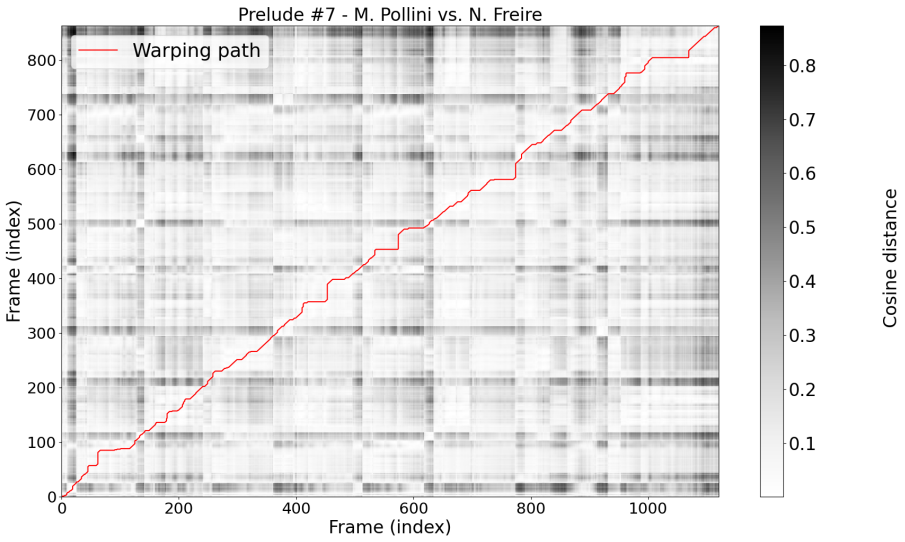


Figure 3.3: Warping path over cost matrix for the two recordings mentioned above. In it we can see the large quantity of horizontal and vertical stretches caused by similar note sequences ocurring along the piece.

The most notable aspect of the warping path shown is the quantity of horizontal and vertical patches, which indicate that a large segment of one of the recordings was attributed to a small portion of the other. Freire's version is shown along the horizontal axis, and Pollini's is represented on the vertical one, and we can notably observe the large horizontal stretch happening around the 100^{th} frame of the recording by the Italian pianist (roughly at 00:04 considering the sampling rate of 44.1kHz), which indicates that several frames of Freire's recording were musically attributed to a single moment in Pollini's version.

Coincidentally, this part of the piece consists of three sustained chords played in sequence, and Freire lets them sound for much longer than Pollini. Both the similarity between the frames corresponding to the chord series and the pause added between them by Freire create a situation where the switching algorithm is not able to identify which chord it is in, introducing errors that might propagate in the warping path.

To solve this, we can use an intuition opposed to the one presented in [3], where the author suggests the use of weights to compensate for diagonal tendencies

in the DTW. By making horizontal and vertical steps more costly, we can force the algorithm to stay on track with the song progression, avoiding the long equivalences present in Figure 3.3.

Using a ratio of 1/3 between the diagonal and horizontal/vertical steps of the DTW without changing neighboring conditions provided results more musically significant than using the standard values or the ones proposed in [3], all without excessively favoring diagonal paths. Figure 3.4 shows the results of aligning the same interpretations as before with the same parameters, only changing the weights used to $w_{\mathrm{d}} = 1$, $w_{\mathrm{v}} = 3$, and $w_{\mathrm{h}} = 3$.

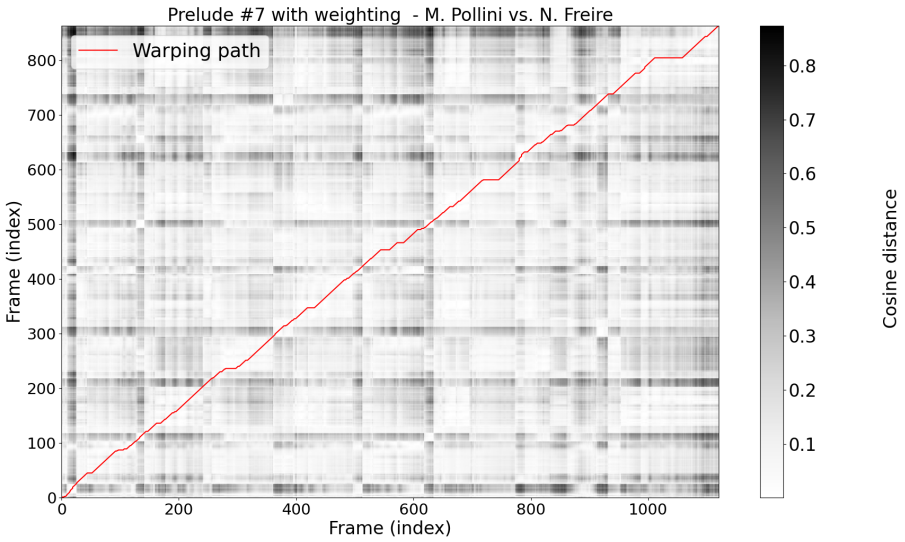


Figure 3.4: New warping path calculated using the modified DTW, displayed over the same cost matrix as before. The use of weights that increase the cost of horizontal and vertical steps relatively to diagonal ones reduces the number of cases of extended attributions to one frame.

It is possible to see that the modified weighting sharply reduced the number of horizontal and vertical patches in the warping path, and this is reflected during switching, with greater musical accuracy when changing between interpretations. The result is better than before, but still, as it is possible to see in the small horizontal stretches in the warping path, there is some confusion in frame equivalence caused by the sustained, repeated chords of Prelude No. 7.

3.2.2.2 Enhanced compression

To further enhance switching to reach the objective of seamless change between recordings, another heuristic that proved useful was increasing logarithmic compression. As seen in Section 2.2.2, the interest of using a flexible log scale in our audio representations is being able to adjust the dynamic range of chromagram values in order to allow significant low magnitude bins to help distinguish between frames, avoiding the predominance of just a few high magnitude bins.

If the compression constant γ is not properly chosen, the DTW algorithm may not be able to detect differences between frames, since their vectors will be relatively similar. Very large values will compress the chromagram to the point where frames are indistinguishable because of the reduced dynamic range, while values too small will cause subtle, but relevant, content to be eclipsed by the *chromas* having the highest magnitudes. This can lead to less meaningful features that cause sub-optimal alignment in the musical sense, because the similarity may cause the algorithm to choose, for example, to align the sustain and the decay phases of the notes in one of the recordings to just the sustain part of the notes in the other. The resulting error could propagate along the warping path, leading to the same horizontal and vertical paths as before, and creating an alignment that will not sound seamless during switching.

In Figure 3.5 we can observe the warping path over the cost matrix for the same experiment as before, once again using a three-to-one weighting ratio, but this time increasing the compression constant γ from 10 to 100. It is possible to see that the dynamic range of frame values was decreased in comparison to Figure 3.4, but, more notably, the warping path was smoothed out with reduction in the size and number of horizontal stretches. A long horizontal patch remains around frame 1000 of Freire's recording (minute 00:46), but this is due to the long ending notes played by the Brazilian artist in his version of the piece.

3.2.2.3 Number of frames and processing time

Another important aspect to be considered when choosing parameters for the interpretation switcher is the window size to be used. As we have seen in Chapter 2,

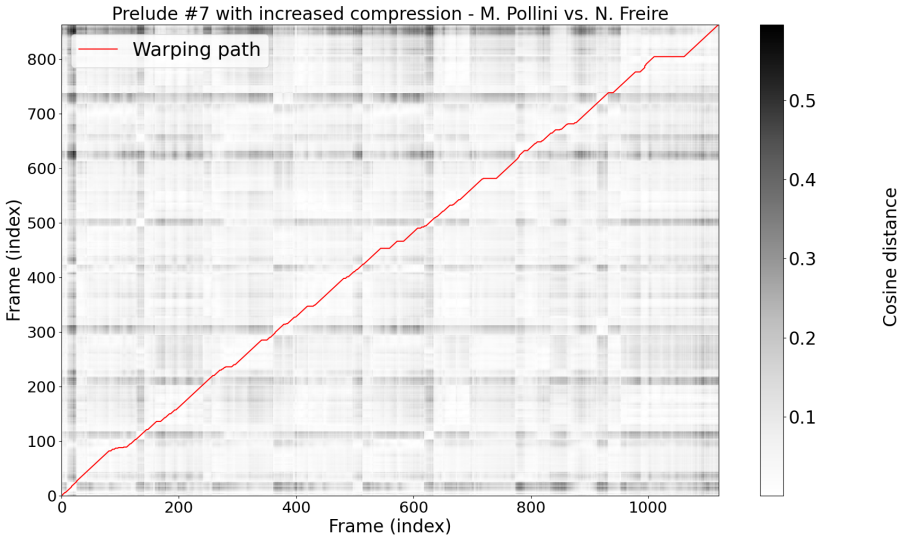


Figure 3.5: Alignment calculated for the same experiment as before, once again using unbalanced weighting, but this time increasing the constant for logarithmic compression. The dynamic range of the original chromagrams decreases, leading to better distinction between frames.

using shorter or longer windows in the STFT influences time-frequency resolution, possibly causing some events to be imprecisely represented in one of the two domains.

Therefore it is fundamental to choose a window large enough to accurately represent frequency content without making it excessively small, causing time information loss. For most test pieces, a default 50% overlapped window of 4096 samples, corresponding to roughly 92ms using the standard sampling rate of 44.1kHz, performed well enough.

However, a careful consideration of the processing time of the DTW must also be made before choosing the window length, otherwise the user may have to wait for some time before the switcher is ready to start. To illustrate this, we will observe a second experiment, consisting of the alignment of two recordings of the first movement of Beethoven's famous Symphony No. 5: one in the Liszt's piano version played by the Russian musician Konstantin Scherbakov (in a 2006 recording published by NAXOS), and other in a traditional orchestral version conducted by the Austrian conductor Herbert Von Karajan (in a 1963 recording with the Berlin Philharmonic Orchestra, issued by Deutsche Grammophon).

Figures 3.6 and 3.7 show the warping path over the cost matrix of the exposition part of the movement, which covers more or less the first two minutes of

both recordings, and essentially evolves around the piece's famous four note rhythmic motif. The difference between the two images is that the first was constructed using the proposed standard window of 4096 samples, while the second was made using windows four times larger, but keeping the same choices of the remaining parameters.

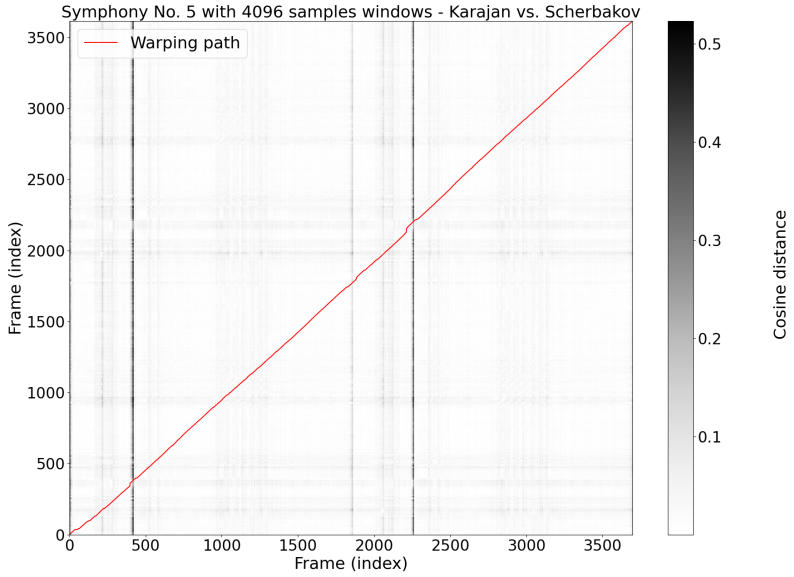


Figure 3.6: Alignment of Beethoven's Symphony No. 5 using the standard window size found to be useful for the switcher: 4096 samples. The piano version by Scherbakov is displayed on the horizontal axis, and the orchestral version of Karanjan conducting the Berlin Philharmonic on the vertical axis. This example shows how *chroma* features are capable of handling recordings made on different instruments or groups of instruments.

The first noteworthy outcome of this experiment is that it demonstrates the robustness of *chroma* features, even for comparing performances with different instrumentation. The alignment was performed on parts of the recordings which were about the same size of the preludes of the previous case, in order to avoid making horizontal or vertical patches look smaller because of the image scale. Yet, it is hardly possible to see any large stretch that could indicate abrupt switching between the recordings.

Another interesting fact is that there is little difference between the two warping paths. Even though the second image uses a much larger window size, there is practically no loss in alignment quality thanks to the fact that the windows

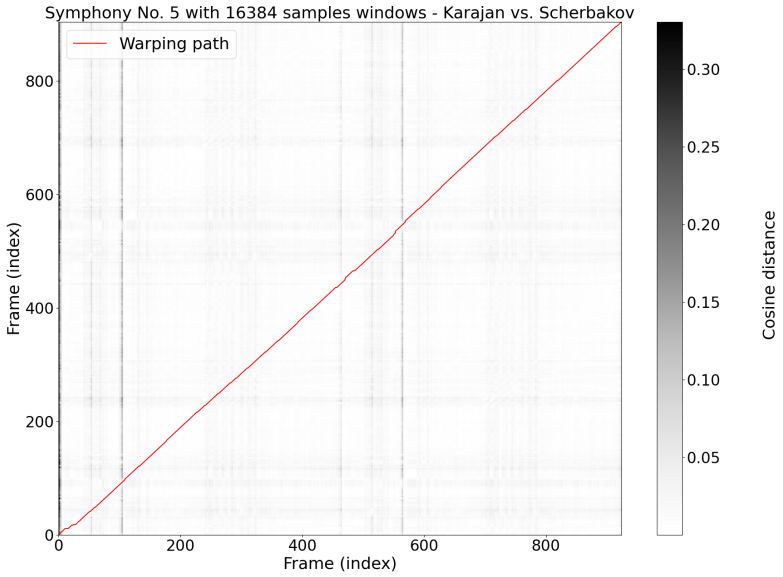


Figure 3.7: Alignment of Beethoven's Symphony No. 5 using a window size of 16384 samples. In this case, enlarging the windows did not cause any loss in quality for the alignment, but it reduced the processing time of the algorithm.

used were still small enough to adequately portray time domain events like note onsets and decays in our audio feature representation for the piece.

This can be used to the user's advantage when dealing with very large pieces such as this one. If the recordings do not contain fast paced parts whose information could be lost when using a large window, an alternative to reduce processing time in the switcher is increasing window size without changing overlap so that the number of frames to be analyzed by the DTW can be reduced.

Aligning the whole first movement, which is 07:17 minutes long in the pianist's version, and 07:14 in Karajan's, the total processing time for 4096-sample windows was 12.6s, against 5.5s for 16384-sample windows. When extending this to a full piece, which could easily exceed 30 minutes — as is the case with the complete recordings of Symphony No. 5 — the result could be too long a wait time to make interpretation switching a feasible feature in an audio player, especially considering that more than two recordings of the same piece could be added, which should be pairwise aligned before the switcher could run.

3.2.3 Results

Aside from weighting, compression, and window size combined with overlap, the parameters used both in the DTW and in audio feature extraction do not seem to provide significantly better warping paths when changed. Because of this, the following parameters are set as default for the switcher in all other experiments presented here: 4096 samples Hann windows with 50% overlap, 4096 points for the DFT, compression constant $\gamma = 100$, normalization constant $\varepsilon = 0.0001$ with Euclidean norm, and cosine distance with standard neighboring conditions and three-to-one weighting for the DTW.

3.2.3.1 Effect of silences in the warping path

An interesting result from the interpretation switcher, and also one that must be taken into consideration when aligning different recordings, is the effect of silences in the warping path. Most commercial recordings of any genre contain a few seconds of silence in the beginning and end of each track, so that they can be clearly separated when played in sequence in a CD or other format.

Up until now the recordings used in the examples presented were stripped off of these silences, since we were more interested in seeing how parameter choices could modify the frame equivalences found by the switcher. However, it is important to notice that they will be taken into account by the DTW when the warping path is calculated, which could possibly cause problems in alignment.

To illustrate this, we will see two images corresponding to the warping path over the cost matrix of M. Pollini's version of Prelude No. 4 when aligned with a computer generated version of the same piece without any additional silence. The technique used to create this synthetic version of Chopin's work was the same that will be further explained in Chapter 4, but for now it is enough to say that it is equivalent to a recording of this song without any interpretation nuance whatsoever.

This prelude has a very rhythmic left hand, which marks the pace of the piece, and a simple melody with few notes throughout most of its duration. Close to the end, however, it has a sequence of fast notes followed by a return of the previous simple melody. The expectation when aligning these two *.wav* files is an

almost perfect alignment, given that Pollini is very respectful to what is written in the music sheet, and does not add many dynamic and tempo variations to the piece.

However, as it is possible to observe in Figure 3.8, with Pollini's version on the x axis and the synthetic version on the y axis, when the piece approaches the end, more or less at frame 1750 of the Italian pianist's version (roughly at 01:20 considering $F_s = 44.1$kHz), the warping path strays from the diagonal route, which could mean an inaccurate frame equivalence, considering how strict Pollini is.

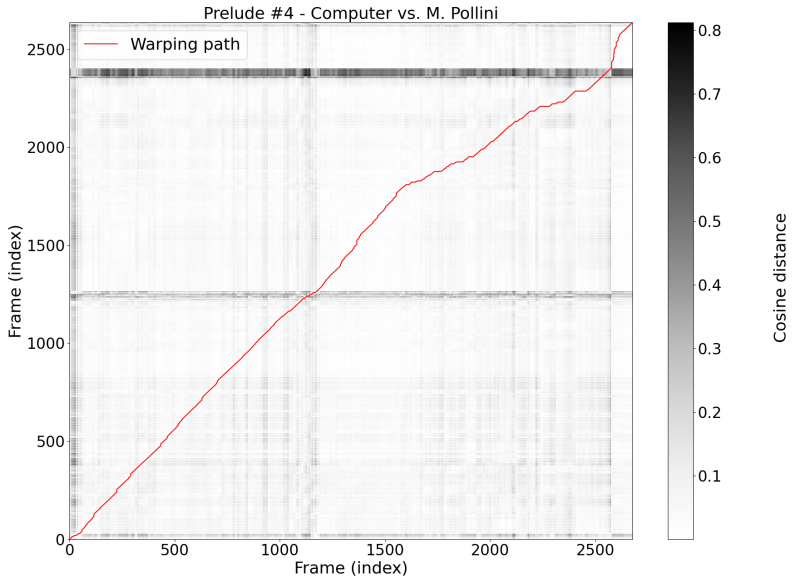


Figure 3.8: Warping path over cost matrix for Pollini's recording of Prelude No. 4 when compared to a computer generated version of the same piece. The fact that Pollini's version contains silences in its end for commercial reasons causes the DTW algorithm to get lost near the end of the piece.

By listening to the recordings using the interpretation switcher it becomes clear that the warping path becomes messy just after the change of pace in the melody. The impression is that the DTW is not able to assign frames between the recordings having in mind the change in rhythm near the end. This is not caused by any poorly adjusted parameter, or by intrinsic characteristics of the piece or the recordings that make alignment hard in this case, since there aren't any.

The problem is simply that, near the end, the DTW has to take silent frames into account when calculating the optimal warping path, which cascades in the frame equivalence near the end of the recordings. When these frames are removed from Pollini's recording using an audio editing software like Audacity [28], the warping

path becomes much more diagonal, and switching does not sound inaccurate any longer.

This can be verified in Figure 3.9, which contains the same alignment as before, but this time with the silences removed. It may sound too specific to analyze this case considering that most commercial recordings have silences near their ends that could be considered more or less equivalent, but besides the fact that these silences may vary in length, we will see in Chapter 4 that aligning human recordings with these computer generated versions can be very useful for score following, with silences having a significant impact on performance.

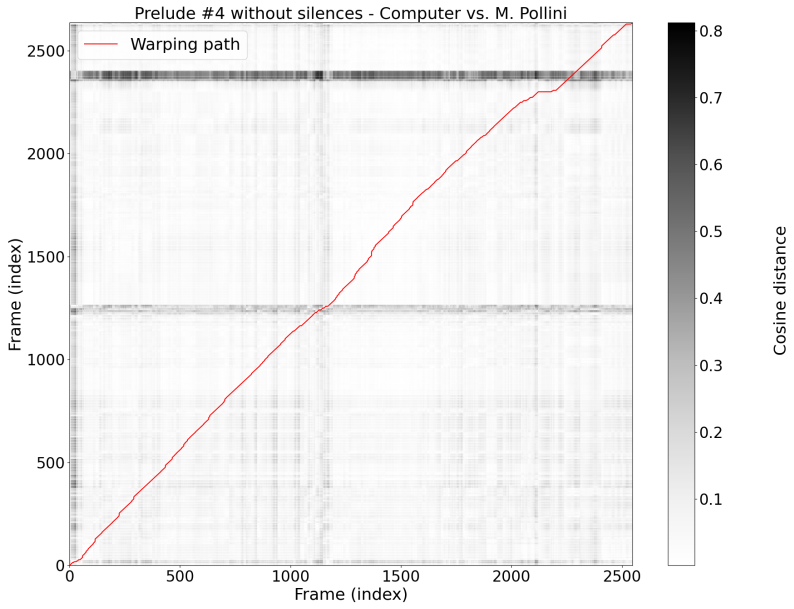


Figure 3.9: Same example as in Figure 3.8, but this time with the silences removed. It can be seen that the warping path becomes much more diagonal, indicating that taking silent frames into account can decrease the quality of the alignment found.

The automatic removal of such silences is possible, but is not implemented in the interpretations switcher. To add this feature, a possible solution could be simply removing all samples below a certain threshold at the beginning and end of the audio files before processing them. In all examples presented from now on, recording silences are manually removed, unless stated otherwise.

3.2.3.2 Effect of noise on the warping path

Given that it might be desirable for the user to analyze interpretations recorded by artists not alive anymore, or even to compare recordings made by the same person, but on different dates, it is also important to observe how the switcher performs on noisy recordings.

The combination of the DTW with *chroma* features proved itself to be fairly robust to noise, and to verify this, we can observe the alignment of two versions of Chopin's Prelude No. 3: one by N. Freire, recorded in the same year as the ones that were analyzed here before, and another by French pianist Alfred Cortot, from 1925.

If all of Chopin's interpreters presented so far were ordered by their respect to dynamics written on the music sheet, from the less rigid to the most formal, Cortot would certainly be first on the list, with more nuances than the other two, then followed by Freire and his intuitive playing style, before finally reaching Pollini's deep respect for the music score.

Nevertheless, despite the fact that there is a difference in the level of dynamic and tempo variations introduced by Freire and Cortot, this example can be seen independently from the differences between the two musicians because Prelude No. 3 contains a very fast and rhythmic left hand that limits the amount of nuances that can be added by the pianists.

Being so, this becomes a good test case for the performance of the algorithm on noisy recordings, since Cortot's version is filled with background noise inherent to the techniques available for music recording in 1925. The warping path over cost matrix for the comparison between both pieces is available on Figure 3.10, with Cortot on the x axis and Freire on the y axis.

It is remarkable how the alignment path remains diagonal and how the overall distance values remain low despite the fact that the French artist's interpretation is heavily degraded by background noise. This can be explained thanks to the preprocessing steps seen in Chapter 2.2.

The background noise present in the oldest version is spread more or less homogeneously across the frequency spectrum. When pooling is performed in order

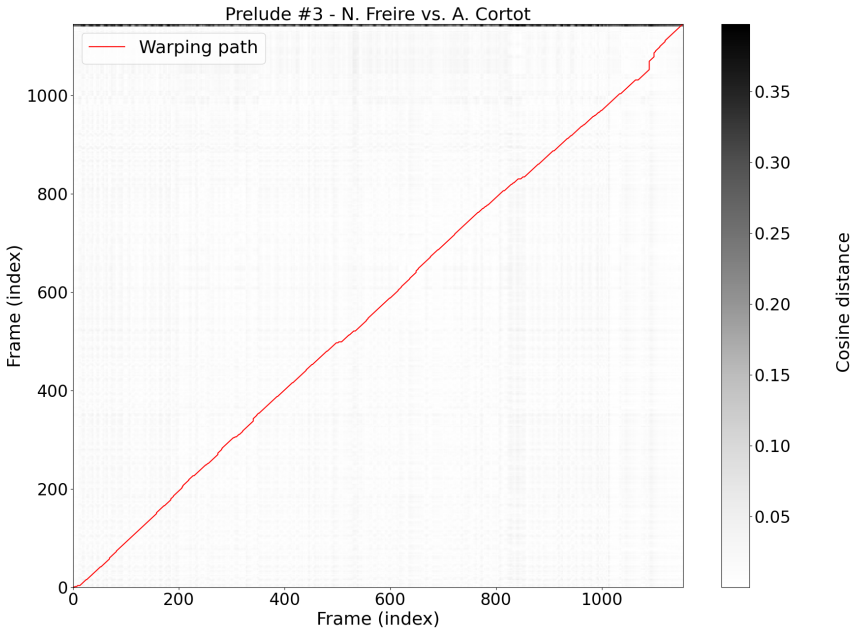


Figure 3.10: Alignment between two recordings of Prelude No. 3, by N. Freire, and A. Cortot, with the latter being an old recording from 1925. It is possible to observe that the warping path's diagonal trajectory is mostly maintained despite the presence of noise. This robustness in given by the preprocessing steps seen in earlier chapters.

to construct the chromagram, all *chromas* are similarly affected, thus keeping bin profiles approximately the same.

After normalization, the effect of noise virtually cancels out, since it is applied to all bins; and as the rows corresponding to the notes being played have larger magnitudes, we obtain representative features even in the presence of noise. The problem happens when the noise in a given recording either is present only within a specific frequency range or is so intense relative to the signal that the magnitude added by the notes is insignificant, to the point of producing almost random features.

The latter can be seen in Figure 3.10, where in the end of Cortot's recording, as the last note played gradually fades into silence, the ratio between signal and noise slowly decreases, creating random audio features. The result is that the distance between the last frames of the two recordings is much larger than throughout the rest of the alignment, as the dark line on the top of the image indicates.

This result shows that the way the audio features used in the alignment are built adds some robustness to the method. Cortot's 1925 recording can be considered

a fairly extreme example of noisy recording, and is definitely not the typical use case of the interpretation switcher. However, the fact that the algorithm performs well on his version shows that broadband noise should not be an issue in most alignment tasks, at least not as long as there are not other factors that can influence the quality of the warping path.

3.2.3.3 Effect of large interpretative variations in the warping path

There is a limit to how much the DTW is capable of stretching one interpretation so that it can find the best possible warping path to another one. When two versions of a piece differ too much in tempo and dynamics, some distortion surely will appear on the warping path.

The most common situations in which these distortions tend to occur are sequences of identical chords or notes, especially if their duration or if the pauses between them can be shortened or enlarged depending on the effect the artist wants to pass to the audience. This occurs naturally because the DTW will be forced to make attributions of multiple frames to one to compensate the duration differences between the interpretations.

To highlight this effect, we will see the alignment of just the first repetition of the characteristic motif of Beethoven's Symphony No. 5 in two orchestral recordings, one by Italian maestro Arturo Toscanini (with the NBC orchestra, in 1952), and another by his German contemporary Wilhelm Furtwangler (with the BPO, in 1954). While Toscanini conducts the symphony in a very dry way, shortening the silences and the duration of the final note of the motif, Furtwangler makes them as long as possible to create a feeling of tension to the listener.

When listening to the alignment, if switching is made exactly on the second note of the second repetition of motif, the impression when going from Furtwangler's version to Toscanini's is that a note was missed from playback, as if the second orchestra simply had forgotten to play it. In fact, what happens is that for this note there is an incorrect frame attribution that makes most of the second and third notes in the German maestro's version equivalent to the third note in the other recording, causing this abrupt change.

In Figure 3.11, this corresponds to the second long horizontal frame attribution happening around frame 125 (roughly 00:06) of Furtwangler's version, which is displayed in the horizontal axis. The other two horizontal stretches happen exactly during the silences between the motifs, and occur because Toscanini is much more concise in their use than Furtwangler.

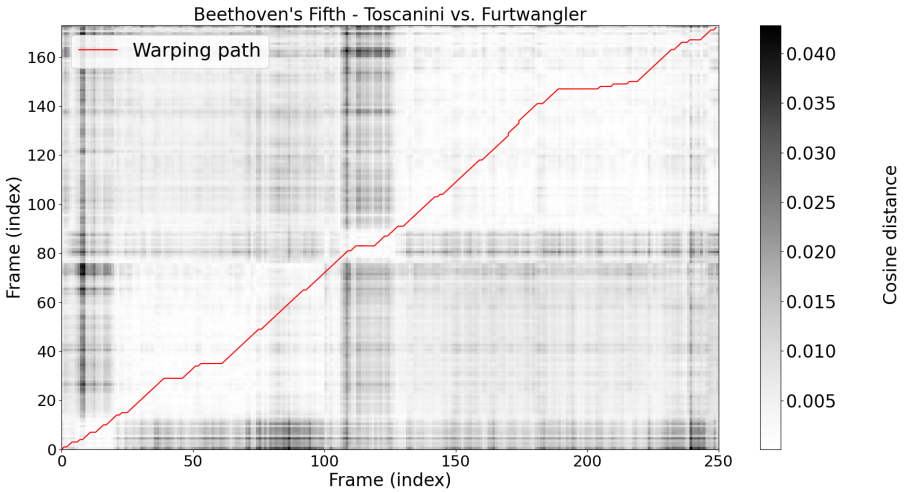


Figure 3.11: Alignment between two orchestral versions of the characteristic motif of Beethoven's Symphony No. 5, one conducted by Arturo Toscanini (vertical axis), and the other by Wilhelm Furtwangler (horizontal axis). The three long horizontal stretches in the warping path are a consequence of the startling differences between the interpretations of the two maestros.

This may seem like a bad example since the horizontal patches in this warping path look larger than before because the excerpt of the piece that was analyzed is very short. However, it not only shows the effect that many to one attributions can have on listening while using the interpretation switcher, but it also demonstrates that the DTW has clear limitations related to how much it can distort one interpretation to fit into the other without generating musically incorrect frame equivalences.

In Chapter 4 we will analyze in further depth examples like the one in Figure 3.9, consisting of comparing real recordings with computer synthesized versions of the pieces, and we will see that these horizontal and vertical patches happen frequently in these situations because of the acoustic differences between synthetic and real recordings, and also because of the lack of interpretation in the artificial

versions. Then, it will be important to be aware of this limitation of the method to understand the results presented for score following.

3.2.4 Typical use case

To synthesize the results of interest for the interpretation switcher, we will see a final example representing a typical use case of the system. It consists of the alignment of two recordings of Prelude No. 18 of *opus* 28 by Chopin.

This is a very hard piece in terms of performance, mainly because of rhythmic structures that make it difficult for the artist to find the beat and maintain tempo. Because of this, there is almost always a significant difference in its execution by different interpreters, which is precisely what we want to be able to identify thanks to the switcher.

The two artists we will compare are Alfred Cortot, once again, but this time in a recording of 1955 — less noisy than the one from 1925, but still 'dirtier' than a modern recording — , and François-René Duchâble, a French pianist who can be considered to be even more strict and formal than M. Pollini. The latter's playing style is so cold, that even for this piece it presents little variation in tempo, despite the fact that it is very rhythmically challenging. Cortot, on the other hand, was already old at the time of this recording, and not only takes immense liberties with tempo and dynamics as usual, but also plays several wrong notes and truncates some difficult passages.

Figure 3.12 shows the warping path over the cost matrix of the comparison between the two recordings, with the version by Cortot in the horizontal axis, and the one by Duchâble in the vertical axis. Like in most cases for the interpretation switcher, the warping path is not fully diagonal, but presents few and short many to one attributions.

The short horizontal stretch that occurs around frame 600 of Cortot's version (about 00:27 minutes into the recording), is caused by a pause in his interpretation that is much longer than indicated on the music sheet, as opposed to Duchâble, who strictly respects the score in this moment of the piece. Other two short horizontal patches appear just before this, and are also consequence of Cortot's dramatic pauses after certain support chords.

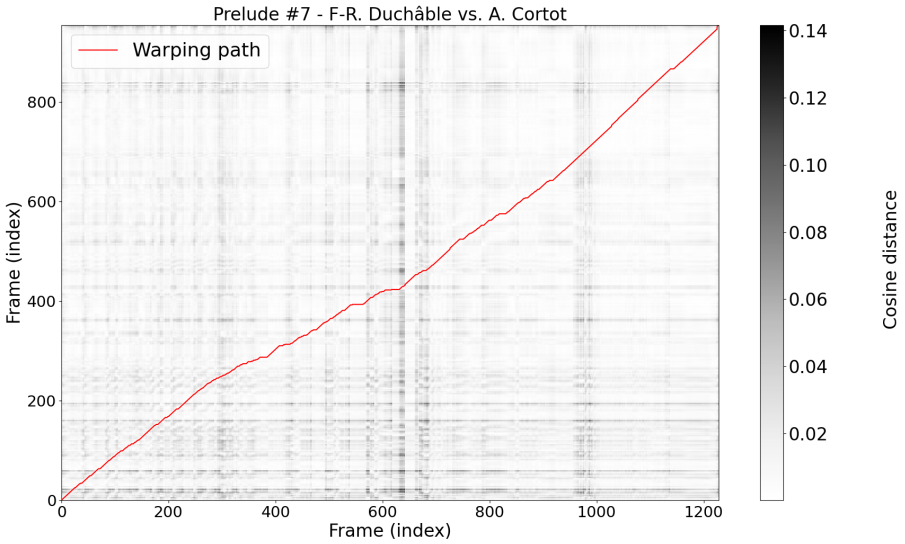


Figure 3.12: Alignment between two versions of Chopin's Prelude No. 18, one by Alfred Cortot (recorded in 1955) and another by François-René Duchâble. There are considerable interpretation differences between the recordings, but the warping path found remains more or less diagonal. Cases like this one, where there are interpretation differences, noise, and possible small mistakes by the musicians can be considered to be a typical use case situation for the switcher.

All in all, it is interesting to observe a real use case for the interpretation switcher. In this example we can see its performance when there are differences in recording quality and noise, variations in playing style, and even occasional mistakes by the musicians.

It shows that the method is able to respond with considerable robustness to some of the challenges of finding equivalences between instants of different recordings of the same piece. Despite still being vulnerable to extremely large interpretative differences and to the pitfall of end-of-recording silences, the method still performs well in the presence of noise and was able to find musically correct equivalences for most parts of the songs analyzed.

The concepts seen in this chapter will be useful to understand the working and performance of the other block of the project: the score follower. Its functioning relies heavily on using the same method presented here to align computer generated versions of the pieces being analyzed with human recordings of them. In Chapter 4 we will see how these synthetic music recordings can be used to identify where a

musician is in the music sheet, and how to use this to display the score and follow it in real time.

Chapter 4

The Score Follower

Score following composes the second important block of this project. Given that with the interpretation switcher it is possible to hear the differences between two different versions of the same musical piece, it is very important for educational purposes to also be able to know which part of the score is being played at any moment where there could be switching. To study two pianists and their style differences in a certain piece, for example, it is not enough to hear the way they both play the piece, it is also necessary to contrast both recordings with the music sheet, so that there is a base reference for comparison.

Being so, the core idea of the score follower is displaying the score for the piece being analyzed in the interpretation switcher, so that this reference is available for the user without needing to open the music sheet in a specialized viewer. But more than that, it is also to be able to mark the notes being played in any given recording in execution time, to make the system accessible for beginners who might not have mastered reading music yet.

Score following is still an active research topic and here we will see how it is implemented in the web app using the alignment techniques that were seen earlier. In particular, we will verify the limitations of the libraries for reading and displaying scores, and also observe the performance of their combined use with the DTW for following a score.

4.1 Implementation

For the score follower, two different problems need to be solved: rendering the score on a screen for the user, and mapping an instant of a recording to the corresponding note in the displayed score. For the latter, the intuition is to create a computer synthesized audio version of the score, and then obtain the onset instant of each note by counting the number of beats[1] coming before it and dividing the result by the tempo given in beats per minute. The onsets of the notes in any other version of the same song can then be found using a DTW frame equivalence between the synthesized version and the recording.

For the former, however, no intuition was enough to make score rendering in Python a simple task. A first demo was made using Kivy [44], a Python library for cross-platform graphical user interface development, but coding the drawing of all possible music notation symbols on the interface soon became excessively complicated. Because of this, the score follower is redeveloped here as part of the web app, in order to make use of some Javascript libraries that make score rendering both simpler and friendlier to the most common digital music sheet format, MusicXML.

4.1.1 Rendering digital music sheets

Nowadays, sheet music is available in many different formats. From old paper scans in *.pdf* files, to markup languages that describe each symbol in a score, there is a plethora of ways to digitally display music scores. Software with proprietary music sheet formats, like Musescore [16], Finale [18], and Sibelius [17], are in general also score editing computer programs, and not only do they read and display music notation, but they frequently allow the users to hear and edit the scores they wish. As an example, a screenshot of Musescore's interface is shown in Figure 4.1.

In the past, using professional software like these posed a very simple inconvenience: not being able to translate scores between different proprietary formats.

[1] Beat here is understood as the duration of the note as a reference for a tempo instruction given in beats per minute. If the beat is, for example, a quarter note, then this means that in a 180bpm performance the first minute will contain the equivalent of 180 quarter notes. It is important to observe that this could mean 360 eighth notes, 90 half notes, 45 whole notes, or any other suitable combination.

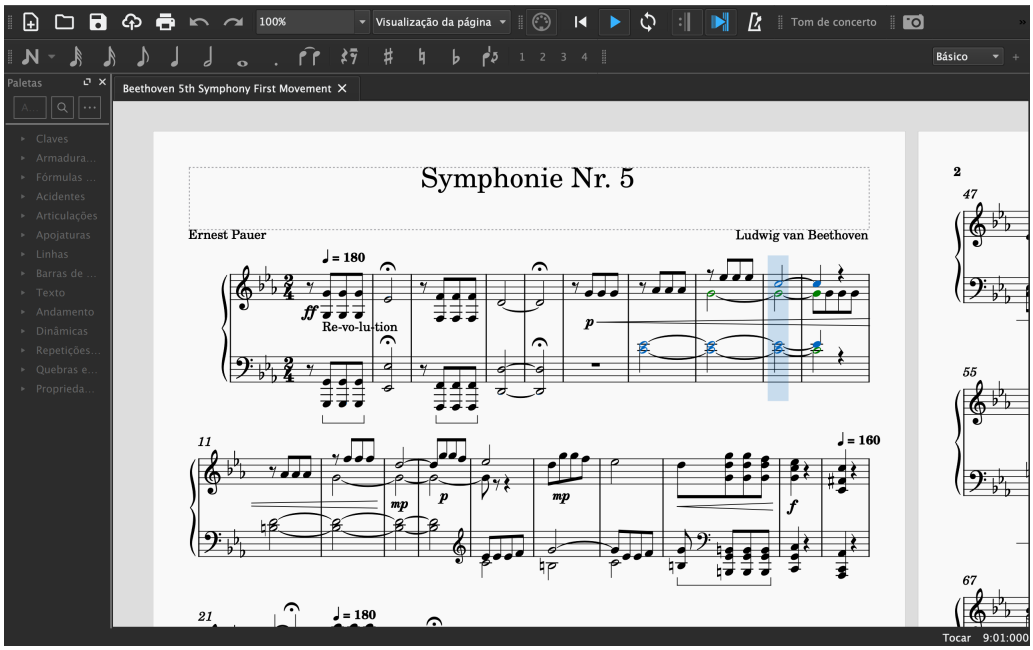


Figure 4.1: Musescore screenshot for a music sheet for the first movement of Beethoven's Symphony No. 5. The screenshot was made during playback to illustrate Musescore's functionality of being able to synthesize the music sheets it reads.

To solve this, Michael Good created in 2001 the MusicXML format [7], with the goal of creating a standard file extension for interactive music sheets.

MusicXML is a tag markup language, very similar to the standard XML in which it is based, and to other markup formats like HTML. Essentially, every music element, including notes, measures, staffs and clefs is represented with a tag that can have attributes, like the number of a measure or the identification number of an instrument part, and child elements, such as note tags inside a bar element.

Today, *.musicxml* files and their compressed versions, *.mxl*, are the *de facto* standard for sharing music sheets via the internet, with more than 250 applications currently supporting it, according to the format's website [45].

MusicXML became mainstream in the web development environment when libraries capable of parsing *.musicxml* files, and drawing music notation on the browser using HTML's *canvas* and *svg* tags started appearing. An example of the former is OpenSheetMusicDisplay (OSMD) [46], the library that is used to render music sheets for the score follower, which is powered by VexFlow [47], a music engraving package that falls into the second category mentioned above.

Basically, OSMD works as a bridge, parsing all tags in the *.musicmxl*, and delivering the necessary information to the correct VexFlow methods responsible for effectively drawing music notation. The full list of features and limitations of OpenSheetMusicDisplay can be found in the project's GitHub page [48], but it suffices to say that the library covers the most common symbols in music notation, like beat signatures, notes, rests, *crescendos*, and much more.

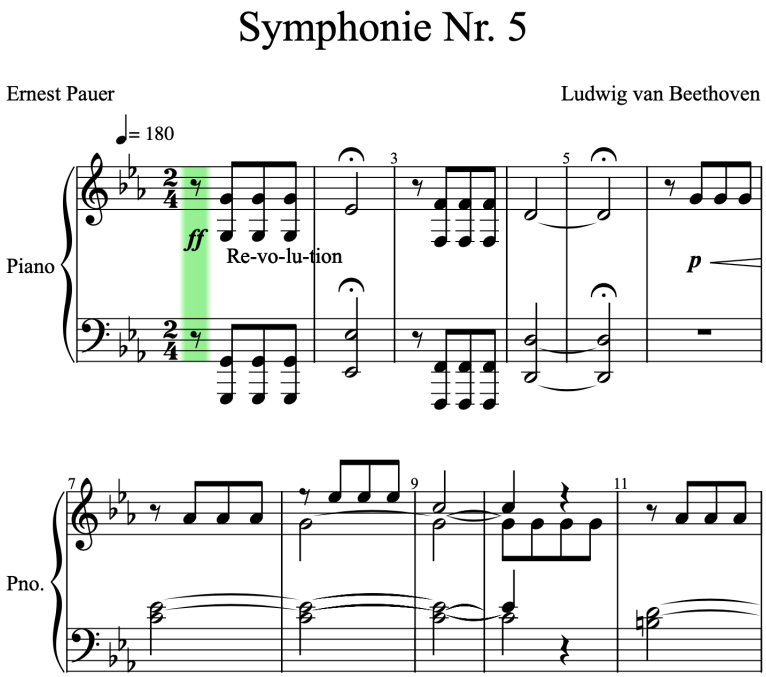


Figure 4.2: Screenshot of a music sheet rendered in a blank web page using OpenSheetMusicDisplay. OSMD covers the most common music notation symbols, and also provides options for controlling drawing margins, changing the score's font family, and more.

Figure 4.2 shows the result of rendering the same score of Figure 4.1 on a blank web page using OSMD. The package allows to control several aspects of the final display, such as: changing the margins to the borders of the HTML container holding the score, using different font families for lyrics, and choosing not to showcase the title and author of the piece in a header.

The reader might observe in Figure 4.2 that there is a bright green rectangle hovering over the first bar of the music sheet. That is OpenSheetMusicDisplay's cursor, the element that is used to mark the notes currently being played in the score follower, and we will see later how it can be moved around for this purpose along audio playback.

4.1.2 From score to second

OSMD's cursor is not simply a graphical element displayed in the music sheet. Like all visual components of the score, it is represented in code through a Javascript object, but differently from the other objects in OSMD, it comes equipped with methods and accessors that enable iterating through all rendered MusicXML tags, extracting information from them.

For better understanding, in the following few paragraphs, all OSMD objects will be referenced to as capitalized camel case words in italic bold font, while properties and methods will be written in plain italic, also using camel case. Methods will be distinguished from properties by being followed by empty parentheses.

The ***Cursor*** has a child object called the ***Iterator***, which can be moved from symbol to symbol using its method *moveToNext()*. At every stop, the ***Iterator*** can access two important properties, that are: the *currentMeasureIndex*, corresponding to the current bar's number, and the *currentTimeStamp*, representing the elapsed time in the score, measured as the sum of the numeric values of the lengths of all notes passed.

For example, if the ***Iterator*** is on a quarter note preceded by two half notes and one eighth note, the meaning of the last sentence is that the *currentTimeStamp* at this moment will be equal to $1.375 = 0.5 + 0.5 + 0.125 + 0.25$.

Based on their index numbers, all ***Measure*** objects can be retrieved from the ***Sheet***, and from them both the instantaneous tempo in bpm and beat reference note length can be found using the *tempoExpressions* and *tempoInBPM* properties.

Combining the timestamp and tempo information, it is possible to create an array $\mathbf{t} = [t_1, t_2, ..., t_N]$ containing the timestamps t_i in seconds of the onsets of all notes rendered in the sheet. Denoting the numeric value of the beat reference length as b_u, and the current ***Iterator*** timestamp as I_i, any note timestamp t_i can be calculated using the equation

$$t_i = t_{i-1} + \frac{(I_i - I_{i-1})}{b_\mathrm{u}} \times \frac{1}{\mathrm{BPM}} \times 60. \tag{4.1}$$

Equation 4.1 recursively calculates the note onsets in seconds by converting the timestamps coming from OSMD to beats by dividing them by b_u, then converting

the result from beats to minutes dividing by the tempo in bpm, and finally passing the resulting onsets in minutes, to seconds. Because every ***Iterator*** timestamp difference is calculated before conversion, the formula takes into account possible changes in tempo during the piece, either because of changes in the reference beat or in the speed in bpm.

An important remark, however, is that this method is not able to handle mid-measure tempo changes. Tempo signatures can be placed anywhere in the sheet, including in the middle of a bar, without causing problems to score rendering. Yet, since tempo expressions are associated only to measures through the *tempoExpressions* attribute, it is impossible to associate a tempo instruction to a single note inside a bar.

4.1.3 Bridging markup and sound

Having the note onsets for an execution of the song of interest without any interpretation dynamics whatsoever, we face the challenge of creating this synthetic recording from the music sheet.

As briefly mentioned in Chapter 2.2, the MIDI standard provides a way of digitally representing music by associating note pitches with 7-bit integers. However, the format supports more than just representing pitches, and MIDI files can also store information related to note duration and volume.

Furthermore, MIDI can be synthesized in audio *.wav* files, opening the possibility of creating artificial recordings without interpretation dynamics, as long as it is possible to find a way to parse MusicXML instructions into a *.mid* file. Happily for this project, researchers at MIT created music21, a Python "open source toolkit for computer-aided musicology" [49].

In expansion since it appeared in 2008 thanks to Michael Cuthbert, music21 is a package offering very diverse tools for studying music datasets in Python [49]. It is capable of parsing multiple sheet music formats, and easily converting between some of them, besides providing an object oriented programming approach to dealing with music notation in Python.

Originally for this project, the idea was even using music21 to parse and render *.musicxml* files using Python only, but in the end this single language approach

did not work because of music21's incapacity to create interactive scores such as the ones generated with OSMD. Even though rendering turned out not to be feasible using solely this package, music21 still provided a simple and efficient way to parse MusicXML in Python, and also a convenient way to convert these files to MIDI.

After converting any given score to *.mid* using music21, synthesizing it as an audio file ready to be followed using the timestamps calculated earlier is also a matter of finding the right computational tool for the job.

The tool on which Musescore [16]'s audio synthesis is based, FluidSynth [50], is an open source, sample based, audio synthesizer capable of creating *.wav* files from MIDI. It essentially works using sound fonts, which are pre-recorded notes from several instruments that can be assembled together during conversion to create audio based on MIDI instructions.

Conveniently enough, Python has a library called midi2audio [51] that is able to pass instructions to FluidSynth from standard Python code. The library calls FluidSynth's command line tool based on certain parameters given in the function call, and is able to return clean *.wav* files from MIDI inputs.

The resulting audio file after conversion using FluidSynth will precisely follow the instructions saved on the MIDI file, which, thanks to music21, will be an exact, dried up copy of what is written in the music sheet. When listening to the audio, no interpretation variations are present, and notes are placed exactly where indicated in the sheet, with no elongated pauses or silences.

As a consequence, all note onsets happen exactly at the timestamps calculated previously using OSMD's objects. Based on this, it is possible to know, in playback time, where to place the cursor so that the score can be followed by marking the note being currently played.

4.1.4 Calling back the cursor

The problem that remains, after having both the note onsets and the recording that perfectly respects their positions, is actually updating the graphical cursor on the screen so that the score can follow the audio being played.

Despite being able to iterate over all symbols in the sheet, the ***Iterator*** is invisible to the user. The actual Javascript object that is displayed on the screen is

its parent, the ***Cursor***, which also comes equipped with methods for moving itself around the score as well as with some convenient tools for hiding its presence and resetting its position.

In order to update the ***Cursor*** position using its built in *next()* method, we need to set up a function callback, which is simply a timer that triggers the execution of that function in regular periods of time[2]. Callbacks are a commonly used tool in web developing, and are useful whenever there are elements in the browser that need to be regularly updated, like the score display.

To make sure that the ***Cursor*** is in the correct position at every moment, an array containing an ordered copy of the onsets is kept. Whenever the update function is called back, it removes the first timestamps from the array as long as the first element remains behind the current playback time. For every onset time removed from the array, the ***Cursor*** is moved ahead by one position, ensuring that it always tracks audio progression.

Take, for instance, the example of Figure 4.3. The timestamps for this music sheet, as calculated by OSMD, would be $\mathbf{t} = [0, 1, 1.5, 2, 3]$, and we could suppose a callback updating both the array and the position of the cursor at every 1.5 seconds after the playback begins.

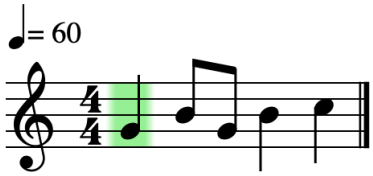


Figure 4.3: Single bar with five notes used to explain the callback for updating the cursor. Because the tempo provided is of 60 quarter notes (beats) per minute, it follows that each quarter lasts one second, and that each eighth lasts 0.5 seconds.

At the time $T = 1.5$s, the callback would be called into action, and would pop elements from the beginning of $\mathbf{t}$, until the first element of the array is larger than 1.5. In other words, it would check that $t_0 = 0 < 1.5$, then remove t_0 and move the cursor one position, then verify that $t_1 = 1 < 1.5$, pop out t_1, and move

[2]Even though this may sound like asynchronous programming, Javascript actually performs callbacks synchronously thanks to a smart queuing system that stacks up the remaining tasks once their timer has expired.

once again the cursor, before arriving at $t_2 \geq 1.5$ and obtaining what is shown in Figure 4.4.

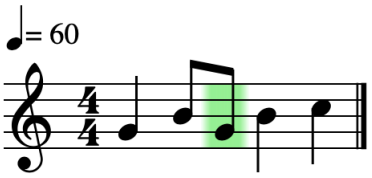


Figure 4.4: Same bar as in Figure 4.3, after one timeout for the callback. The cursor updating system periodically verifies which timestamps have already passed in the audio.

The algorithm would then continue doing this for all playback times T multiples of 1.5, hence updating the ***Cursor*** position along playback. In the example, certain notes are skipped because the callback interval is too large, but by choosing an adequate period between cursor updates, it is possible to give to the user the impression that the update is instantaneous, and that the sheet is followed at every instant of audio playback.

4.1.5 Score following workflow

The technique described above shows how it is possible to follow a score when the corresponding audio is a synthetic version with no interpretation dynamics. We have seen how, from a MusicXML score, we can create a MIDI file, and then use this *.mid* to synthesize sound that can be accompanied by a cursor placed in a music sheet rendered using the same original *.musicxml* from before.

The commercial software that was mentioned earlier is already able to perform these tasks. The difference between the system proposed in this project and the solutions already existing on the internet is the capability to follow a score not only using artificial audio, but also using real recordings.

In order to do this, it is necessary to add the techniques seen in Chapter 3 to the score follower, and think of the DTW as a way to connect human and computer recordings by using the frame equivalences between them. Supposing that a human recorded performance needs to be followed from a MusicXML sheet, the workflow for updating the cursor according to the song playback would be:

1. converting the original *.musicxml* file to MIDI using a parser, like the one provided by music21;

2. synthesizing an artificial recording without interpretation traces from the *.mid* using FluidSynth;

3. calculating the frame equivalences between the synthetic version and the human one using the DTW algorithm, and storing them;

4. rendering the music sheet using the MusicXML file from the first step and OSMD;

5. calculating note timestamps with the ***Iterator*** class, and keeping them in an array for consultation;

6. repeating, in short intervals of playback time, the steps of:

   (a) finding the equivalent frame in the artificial recording,

   (b) calculating the equivalent instant in the synthetic version by converting frames to seconds,

   (c) checking the first element of the timestamps array to see if an update is required,

   (d) moving, if necessary, the cursor until its next calculated timestamp is just after the current playback time.

A block diagram summarizing this routine can be seen in Figure 4.5. In it, the left side represents the user interface where the input is provided, and also where the score is rendered to the user. The blocks represent the packages, objects, and algorithms discussed earlier for producing each one of the partial outputs necessary for the score follower.

It may seem odd that libraries in different languages are used to parse the *.musicxml* in order to create the MIDI file and render the score, just as saying that the DTW is being called to find the frame equivalences, considering that it also uses libraries in Python, as opposed to OSMD's Javascript. This is a subject that will be explored in further details in Chapter 5, where we will also see how the user interface was incorporated in both the score follower and the interpretation switcher.

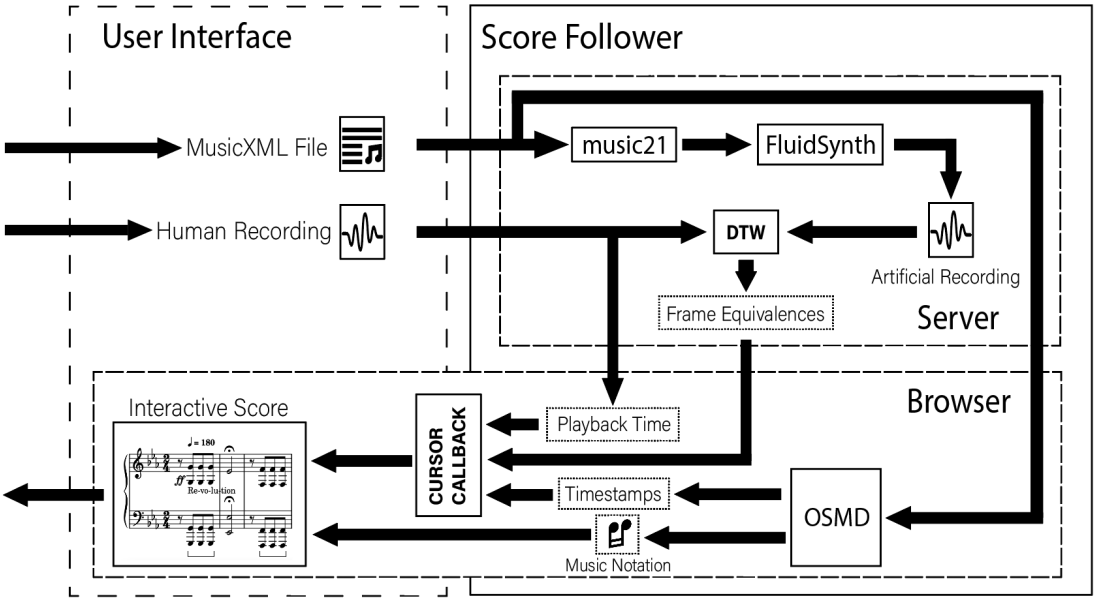


Figure 4.5: Block diagram explaining the relation between the components of the score follower. Once again, the inward arrows indicate user input, and this time the output represented by the outward arrow is the interactive score in which the cursor is updated to follow audio playback. Certain components of the score follower are used in the server side of the application, while others are called in the user's browser, which is why all elements seen before are contained in one of these two blocks in the diagram.

For now, it is enough to know that rendering the score, calculating the timestamps and updating the cursor are operations that are happening on the web page only, whereas the other steps of the process are performed before loading the website in the browser using a Python back end. The files and information needed for Javascript to perform the update callbacks are all stored in the server thanks to these hidden operations, and are fetched back and sent to the user's browser for the following.

4.2 Tests

To test the score follower, free versions of the scores for the pieces mentioned in Section 3.2.1 were found in [43], and were then provided as input to the system alongside the human recordings for these songs. Therefore, Chopin's twenty-four prelude set was once again used to test the performance of the system, but this time

comparing the evolution of the cursor on the screen to the sound coming from the playback of the original versions of the pieces.

The scores were not changed in any manner whatsoever, except for handling factors which were not perfectly managed by OSMD. This includes, in particular, mid bar tempo changes, which are not supported for timestamp estimation. For these cases, the written tempo changes were advanced to the beginning of the bar they were into, possibly replacing a preexisting tempo signature.

As before with the interpretation switcher, it is unfortunately not possible to reproduce this sort of experiment on plain text because of the need to hear and see the performance of the system to evaluate it[3]. To compensate for this, we will see the warping paths over cost matrices of the alignment between the synthetic and human recordings, as was briefly discussed in Chapter 3.

The quality of the alignment between the synthesized MIDI and the original recordings says a lot about the audiovisual performance of the cursor updates. Callback strategies and the technological tools used may vary in order to make estimating timestamps and updating the cursor more precise, but if it is not possible to find a musically significant frame equivalence between the artificial versions and the real ones, it is certain that score following performance will drop using the proposed method.

In the following tests, we once again look for elements in the warping path that could indicate that the score follower is unable to pinpoint the exact equivalence between versions of the same piece. This time, however, because the times in the synthesized recording represent a position in the score, distortions in the warping path express themselves as updates coming too late or too early for the cursor, instead of delays in sound.

4.2.1 Test One – Prelude #4 by M. Pollini

In Chapter 3 we quickly discussed the effects of silence in alignment, and exemplified these results by comparing a human recording of Chopin's Prelude No. 4 and a synthetic version of the same piece. In this first test, we will see once again

[3]This text was originally published alongside a presentation containing videos showing the performance of the switcher and follower. For more information, please contact contact professor Luiz Wagner P. Biscainho or myself.

the effects of end-of-recording silences with the same example of Section 3.2.3.1, but this time applied to score following.

Back in Section 3.2.3, we briefly mentioned that the case of aligning artificial and ordinary recordings was important because of its impact on the performance of the score follower. Indeed, in view of the previous sections, it becomes clear that problematic frame equivalences between the synthesized scores and the real ones will create issues on score following because of the callback used to update the cursor.

When moving the cursor, first the equivalent time to the current playback instant is obtained in the artificial recording. Then, this value is passed for comparison with the timestamps vector calculated from the score, and the cursor is moved forward as long as the next stamp remains behind this equivalent instant in the synthetic version. The consequence is that, if there is an error in the equivalent time due to a musically incorrect warping path, then the cursor will be updated at the wrong moment, creating a time difference between the audio and the position of the follower in the score.

Figure 4.6 shows the position of the cursor at playback minute 01:22 of Pollini's version, when following the recording with trailing silences in its beginning and end. When listening to the recording, it is possible to notice that, at this moment, the piece is in fact a few notes ahead (see position manually marked in red) of the position marked by the cursor.

Reexamining Figure 3.8, it is possible to see that the warping path strays from the diagonal path around frame 1750 of Pollini's version, which is roughly equivalent to minute 01:20 after conversion using $F_s = 44100$Hz. This gives a feel of the visual impact of incorrect warping paths on score following, and highlights the importance of silence removal for correct alignment between human and synthetic recordings.

After removing trailing silences, which do not affect the time instant where the same note of the previous example occurs, that is, playback minute 01:22, the cursor position appears right next to where it should be, with this small one note mistake probably being caused by the grace note[4] placed just before. Since grace

[4]Grace notes are musical notes which are non essential to the melody being played. They are used as ornaments to other notes, and as such do not have a length associated to them.

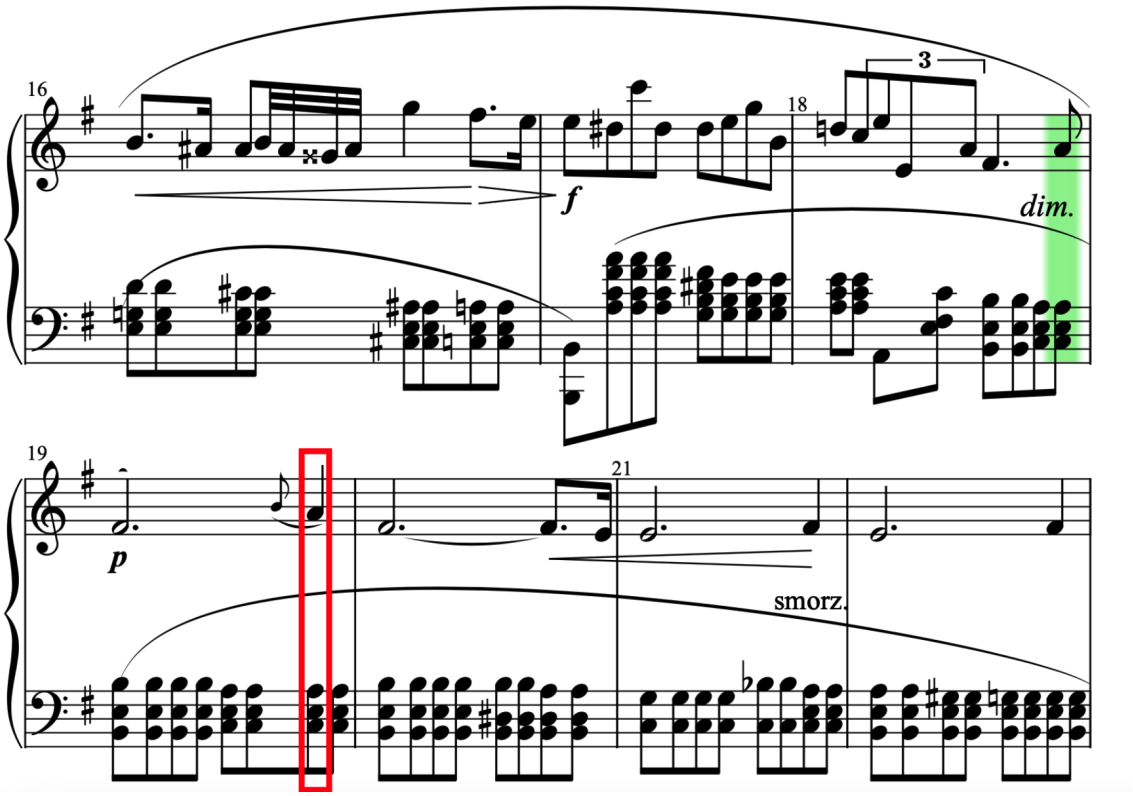


Figure 4.6: Screenshot with the cursor position at minute 01:22 of M. Pollini's recording of Prelude No. 4 while score following. The cursor is a few notes behind of where it should be (indicated in red), an error caused by the imprecise warping path calculated taking account of end-of-recording silence.

notes technically do not have a specified duration, OSMD reads them as having length zero, which explains the small shift from the expected position.

This example once again highlights the importance of removing trailing silences when aligning different versions of a given piece, but, more importantly, gives a visual feel of the impact of incorrect frame equivalences in score following.

If there are long horizontal or vertical patches in the warping path between a recording and the synthesis of the score, or if the alignment calculated strays too far away from the diagonal direction, this means one of the audios is lagging behind the other, and therefore there are several many-to-one frame attributions in the alignment. Visually, the consequence when score following is that either the cursor will remain in the same place while the audio is long past the note it currently hovers over, or it will move faster than playback, marking the wrong notes in both cases.

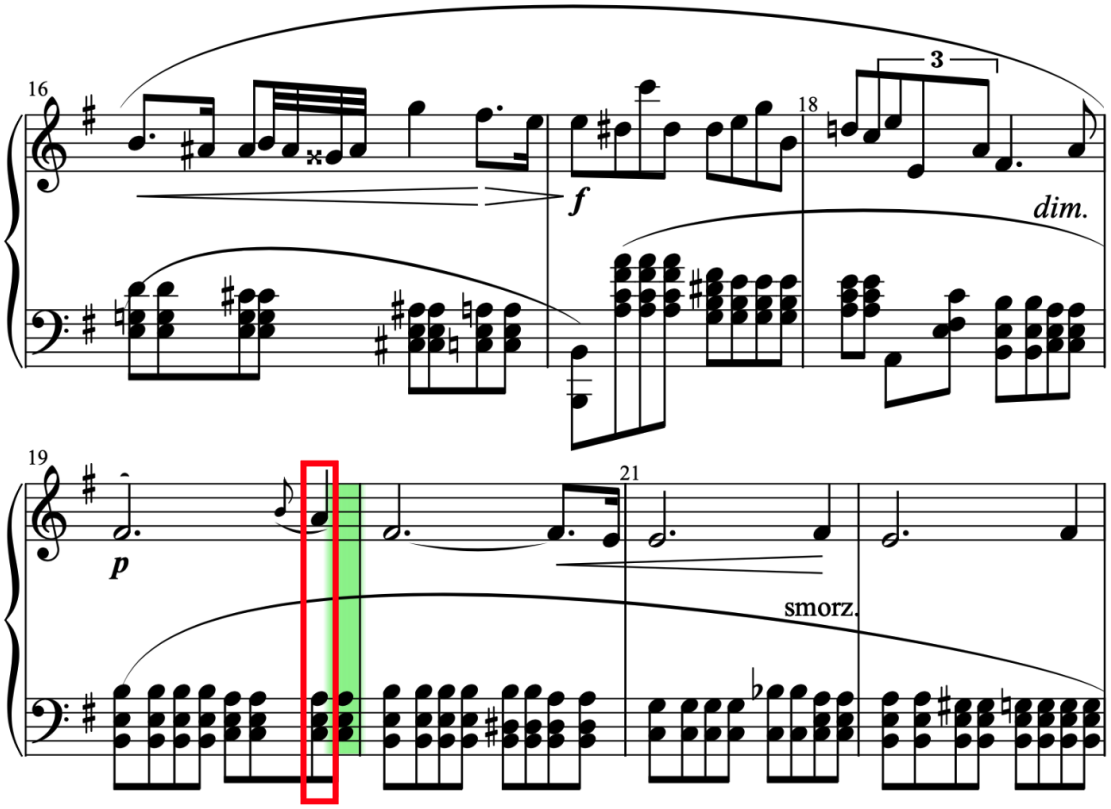


Figure 4.7: Screenshot with the cursor position for the same recording as before, but this time without trailing silences at the end of the recording. The cursor position is placed much closer to the correct one, with the small error probably being caused by the grace note placed just before the note being examined.

4.2.2 Test Two – Prelude #7 by N. Freire

As we have briefly seen in Section 3.2.2, another factor that might influence the resulting warping path from the DTW is the presence of pedal notes. Back then, when we compared M. Pollini's and N. Freire's versions of Prelude No. 7, it was possible to observe that the similarity between frames containing long sustained notes could cause errors in the alignment.

Even though the heuristics we have seen previously help mitigate this effect, some imprecision remains in the resulting frame equivalences. This becomes particularly noticeable when following the score because of the visual effect of the lagging cursor when compared to the audio.

Figure 4.8 shows the warping path over the cost matrix of the comparison between N. Freire's recording of Prelude No. 7 and the synthesized score of the same piece, with the Brazilian pianist's version represented in the x axis. There are several regularly spaced horizontal patches corresponding to interpretative emphases

not executed by the machine, with the most noticeable three occurring around frames 380, 700, and 1000 of Freire's version.

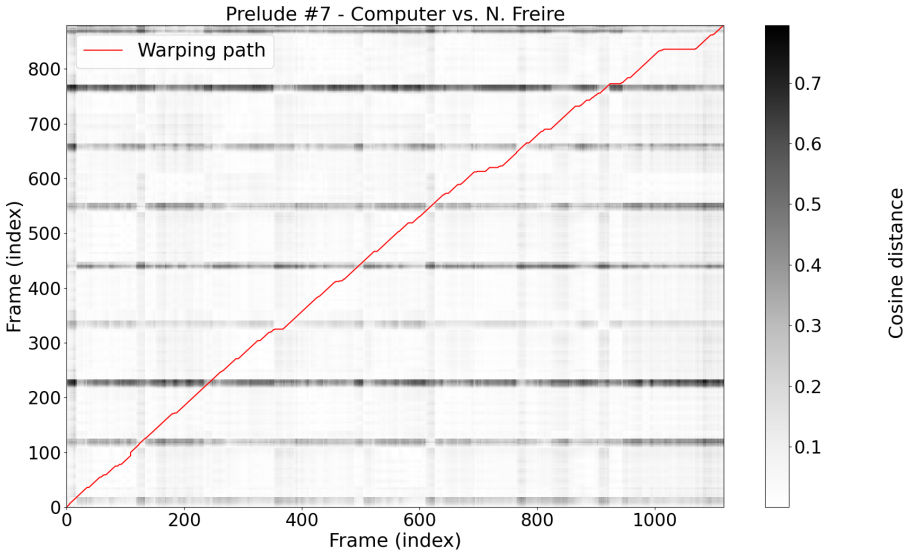


Figure 4.8: Warping path over cost matrix for the alignment of Freire's version of Prelude No. 7 and a synthesized version of the score. It is possible to observe regularly spaced horizontal patches in the warping path corresponding to the sequences of long identical chords in this piece.

In these portions of the alignment path, many frames of Freire's recording — possibly corresponding to notes that have been sustained for interpretative reasons — were attributed to a single frame of the synthesized score. Since Prelude No. 7 contains many long identical chords, frames belonging to each of these are very similar, which blurs the distinction between the chords for the DTW, and causes excessively long horizontal patches in the warping path.

Figure 4.9 shows the position of the cursor at minute 00:18 of Freire's recording (corresponding to frame 387 using $F_s = 44100$Hz), along with, once again, the correct cursor position drawn in red. The cursor is one note behind the audio because there is a sequence of two identical sustained chords played with very soft attacks.

It is important to note that the attacks also play a role in separating the identical chords. Sequences of identical notes that are played very softly will present the same energy in the same *chroma* bands, since the variations in magnitude on the note onsets will be less noticeable.

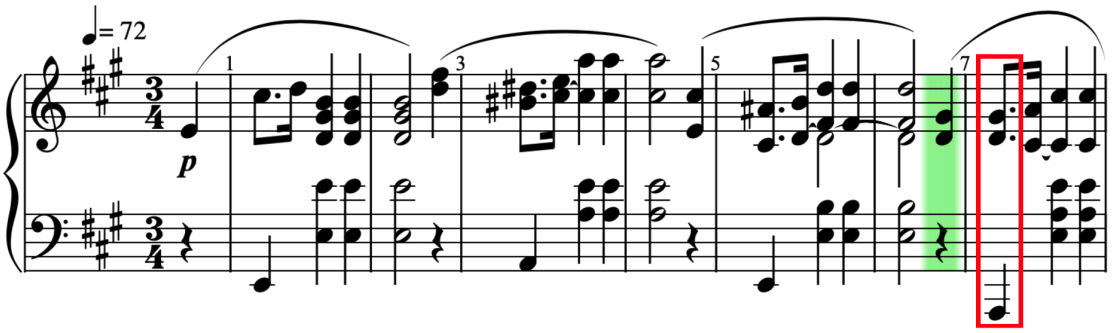


Figure 4.9: Screenshot with the cursor position for the follower at minute 00:18 of N. Freire's recording of Prelude No. 7. Since this piece has many sequences of identical sustained chords, the DTW is not able to perfectly distinguish between them and there might be small errors in the cursor position.

The key takeaway from this test is the impact of pedal notes in score following. Whenever there are sequences of softly played identical notes, the DTW makes small mistakes in the alignment with the synthesized recording because of the similarity between frames. Visually, these mistakes manifest as lags in the cursor position, which are particularly noticeable in slow pieces, like Prelude No. 7.

4.2.3 Test Three – Prelude #18 by A. Cortot

In Section 3.2.4, we looked at an example comparing François-René Duchâble's version of Prelude No. 18 to a 1955 recording of the same piece performed by an aging Alfred Cortot. It was an example of a typical use case of the interpretation switcher, where noise, interpretation differences, and performance errors could influence in the result of the DTW.

These same effects are also present when following the score, since the method relies on finding frame equivalences between two audio inputs, with the additional problem of dealing with the possible limitations of score synthesis.

Figure 4.10 shows the warping path over the cost matrix of the comparison between Cortot's 1955 recording of Prelude No. 18 and a version synthesized from the score for this piece. Just like in the alignment with Duchâble, it is possible to see that the optimal path found strays a bit from the diagonal direction, certainly because of Cortot's occasional mistakes and unorthodox tempo keeping.

Additionally, two horizontal stretches can be seen between 1000 and 1200. The reason they appear could be related not only to Cortot's execution of the end

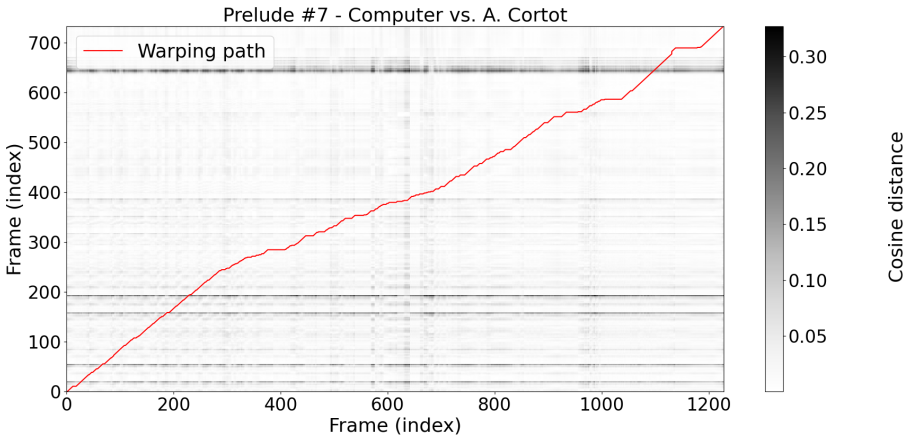


Figure 4.10: Alignment path over cost matrix of the comparison between A. Cortot's 1955 recording of Prelude No. 18 and a synthesis of the score for the piece. The frame equivalence strays from the diagonal path because Cortot makes occasional mistakes in execution, and has trouble executing the piece's rhythmically challenging structures.

of the piece, but also because of the limitations of the libraries that were used to translate the score from MusicXML to MIDI and to synthesize the *.mid* file.

Listening to the artificial recording and looking at Figure 4.11, it is possible to notice that, in the last measures of the piece, both the arpeggiated chord of bar 17 and the trill of bar 18 are not synthesized despite being marked on the score. This creates slight differences in the *chroma* features that can influence the resulting warping path.

Moreover, Cortot holds the last chord of the prelude, as indicated by the *fermata* in the last bar. Because a *fermata* is only an indication that a note should be held beyond its duration at the discretion of the performer, music21 translates the notes of the chord as common whole notes, which makes them sound for less time than in Cortot's recording.

Figure 4.11 shows the cursor's obtained and expected position at minute 00:50 of Cortot's recording, and it is possible to see that score following does not work perfectly in the last seconds of the piece. This is probably caused by the differences between Cortot's recording and the synthetic version that were mentioned earlier, showing the possible limitations of using synthesis for score following.

Nevertheless, the cursor is able to accompany the totality of the recording without being farther than a bar away from the correct position, which is impres-

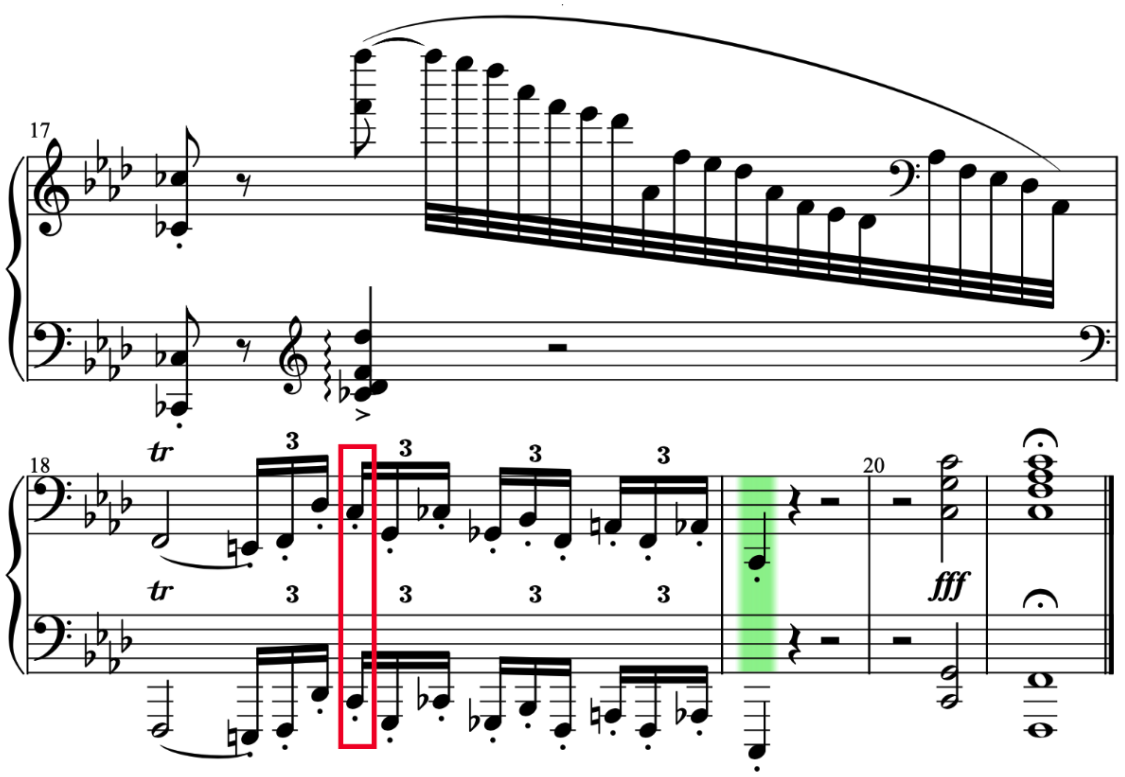


Figure 4.11: Score with the last two bars of Prelude No. 18. The arpeggiated chord in the beginning of measure 17 (represented by the vertical squiggly line), the trill in the start of the 18$^{\text{th}}$ bar (noted by the letters *tr*), and the *fermata* in the last chord (indicated by the dot and and arc over the notes) are not correctly synthesized in the artificial recording.

sive considering Cortot's flexible interpretation of Prelude No. 18. Hearing the synthesized score, this example also shows that the method used to calculate the timestamps works well, as there are not any errors when following the synthesized version, despite the presence of 17- and 20-note tuplets.

This test is representative of the challenges when using the score follower, as it highlights possible problems when synthesizing the scores using the method proposed earlier. However, it also shows that the score follower is able to handle rhythmically challenging structures, even when the performer of the recording being analyzed cannot.

In the next chapter, we will see the architecture that integrates interpretation switching and score following in an easy to use web interface, so that both of them can be used by people studying music and analyzing the playing styles of different performers.

Chapter 5

System Architecture

To reach the goal of creating a minimally viable product for performance analysis, it is not enough to know how the interpretation switcher and score follower work. It is still necessary to find a way to integrate these two blocks and provide a friendly user interface to their usage.

Even though both blocks are fundamentally based on a Python implementation of the DTW, since the score follower relies on Javascript running in a web browser to render the score and update the cursor, a mean of communication between these two tools is needed.

In this chapter we will learn how this was implemented, as well as visualize the full user interface used to control audio playback and navigate between recordings. We will also see the final outlook of the application, and discuss how it can be improved.

5.1 Framework

As mentioned briefly in Section 4.1, in order to use OSMD's tools to render the scores, the follower has to be included in a web application. Traditionally, web applications are composed of two parts: the front end, which runs on the browser and focuses on parts of the website that will be explicitly shown to the user, and the back end, that runs only on the server side, and that handles all the background operations necessary for the website to work.

In the case of this project, the front end renders the score, providing an audio reproduction and navigation system, and updates the cursor during playback. Meanwhile, the back end is responsible for parsing the MusicXML file, correctly synthesizing the audio based on the score, and finding the frame equivalences between all recordings.

The back end part of a website also typically handles redirecting HTTP[1] requests to the correct pages, many times executing certain functions whenever a given URL is requested via a GET or POST instruction. With this in mind, web frameworks [54] having functions that simplify routing requests and other common web development problems were created for many programming languages.

Since most of the other back end tasks were already implemented in Python, the natural choice was using a framework in the same language. Even though there were more sophisticated options [55], the chosen framework for the project was Flask [56], a very simple to use, minimalist framework.

Creating an application using Flask is easy. It comes equipped with a ***Flask*** class, which is responsible for listening to packets arriving at a TCP[2] port specified by the user. Once a request arrives in the assigned port, this class automatically triggers the function in the server assigned to the URL being requested by the client. Usually, this function returns an HTML page and the corresponding CSS and Javascript files to the ***Flask*** class, which then forwards them to the client in the form of an HTTP response.

A detailed description of how routing is done in Flask is beyond the scope of this project, but can be found in the Flask documentation [56]. Here, it suffices to know that functions in the server are assigned in code to run whenever certain URLs are accessed by being wrapped in a Python decorator [57, 56].

Because Flask allows serving complete web pages and running code whenever these pages are accessed, it is perfect for creating a user interface capable of receiving

[1]Hypertext Transfer Protocol, or HTTP, is a communication protocol used to request documents on the internet [52, 53]. In HTTP, a client (usually a web browser) sends a GET or POST request to a server in order to, respectively, obtain or send specific information.

[2]Transfer Control Protocol, or TCP, is the communications protocol which is used by HTTP to ensure delivery of requested contents [53]. When an HTTP request is made from a client to a server, a connection between these two machines is established using TCP, which transfers the information requested by HTTP. TCP uses ports to distinguish specific processes in a client/server.

the inputs needed for score following and interpretations switching and calculating *chroma* features and frame equivalences.

The structure of the web site for this project consists of just two pages: a form page, where the score, recordings and parameters for audio analysis are provided by the user, and a playback page, which contains simple audio navigation controls and the score with the cursor being updated in real time.

5.2 Form page

The form page can be seen in Figures 5.1, 5.2, and 5.3. It is a single HTML form styled using CSS to separate user input into three different sections: the score section, where a MusicXML file should be provided, the recordings section, where all the *.wav* files of the recordings should be given, and the parameters section, where the user is able to adjust the parameters used for calculating the chromagrams and the alignment path.

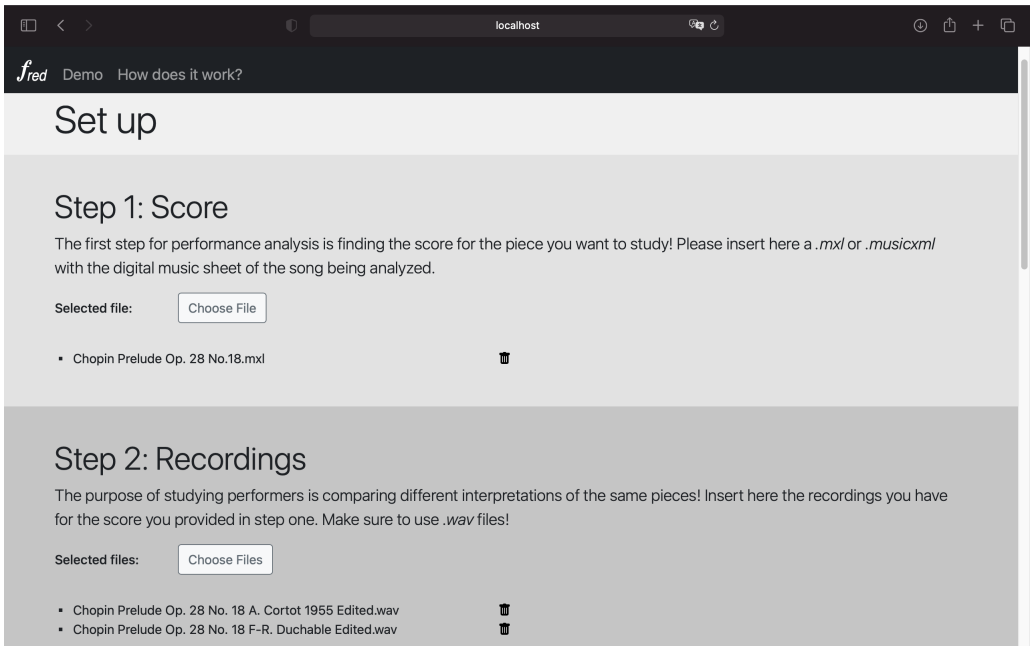


Figure 5.1: Screenshot of the first part of the form page, showing the input fields for the recordings and the score. The whole page consists of a single HTML form styled with CSS.

When the user presses the green button at the bottom of the page (Figure 5.3) after filling the form, a POST request is sent to the server with the form inputs, and the following actions are triggered:

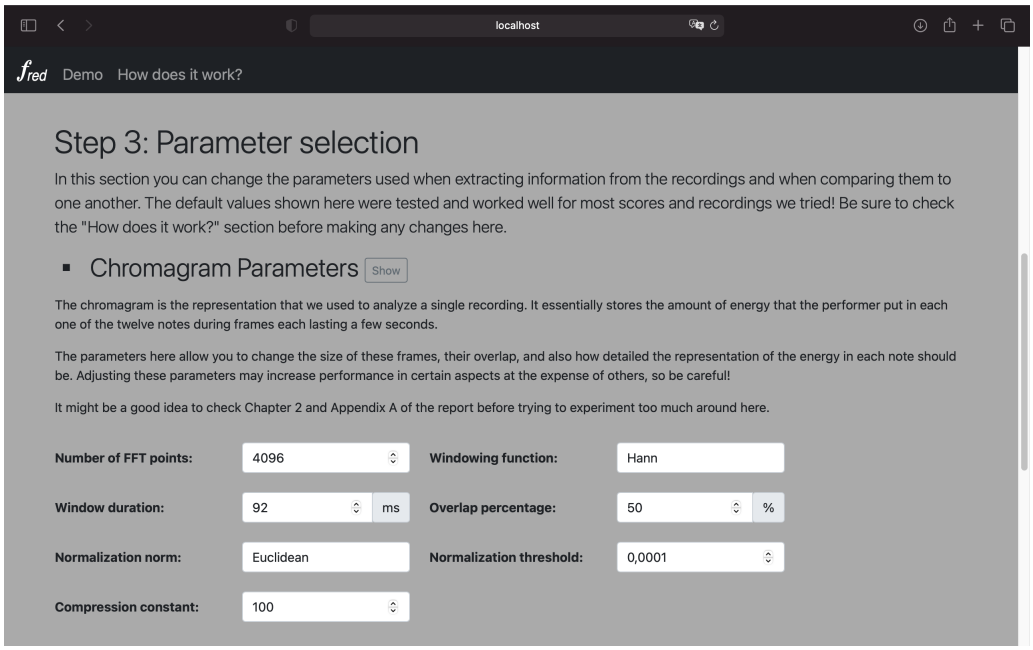


Figure 5.2: Screenshot showing the first part of the parameters section on the form page, where the chromagram parameters are informed by the user. The window size is given in milliseconds so that the number is easier to visualize, but an input verification is made using the recording's sampling frequency to ensure that the window contains less samples than the number used in the Fourier transform.

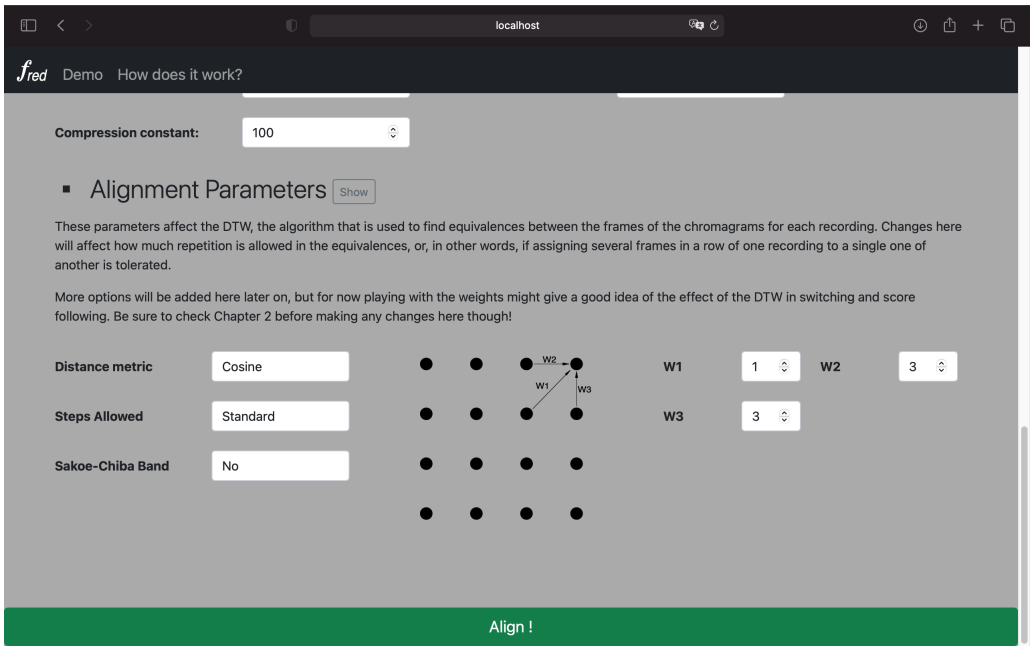


Figure 5.3: Screenshot with the alignment part of the parameter input section. Here are all the parameters related to the DTW, including the weights, the distance used, and the steps allowed. For now, only the standard DTW neighboring scheme, with the three closest neighbors was implemented.

- the score and recording formats are validated, to make sure they are, respectively, MusicXML and WAVE files;

- the score is uploaded and then synthesized to *.wav* format as described in Chapter 4, and the result is stored in the server;
- the recordings are uploaded and stored in the server;
- the sampling frequency for each recording is extracted to be validated and stored with the other chromagram parameters;
- the chromagram and alignment parameters are verified, to ensure they are all valid: all recordings must have the same F_s, the overlap needs to be smaller than 100%, all parameters must have positive values, and the window size in samples needs to be smaller than the number of DFT points;
- all pairwise frame equivalences are calculated as described in Chapter 3;
- the file paths for the score and the recordings (human and synthetic), the validated parameters, and the dictionary containing the frame equivalences between every pair of recordings, including the synthetic one, are saved in the server in JSON[3] format.

In this way, all elements necessary for playing the audio while following the score are put in place and saved in the server. However, the issue that remains is that the audio navigation system runs in Javascript, which is served alongside the HTML page and runs only in the user's browser.

5.3 Interface

The consequence of the need to serve the results of the Python routines to the navigation web page is that an interface is required to send these files back to user's browser. Directly handling them to the user instead of saving them in the server is not possible, as it would require the browser to access local files in the user's computer later.

[3]Javascript Object Notation, or JSON in short, is a lightweight data interchange format [58] used on the web. It conveniently stores information in a structure similar to Javascript objects and Python dictionaries.

Conveniently for us, Javascript is capable of requesting web pages using its Fetch API [59]. With it, Javascript is able to make HTTP requests to access complementary information in any server available on the internet. In simple terms this means that code running in a browser web page can request information from other websites, which is precisely what is necessary to pass the parameters and song files to the user's computer.

Therefore, to serve the JSON with the information required by the navigation system, the solution is simply to create a new route in the back end, responsible for sending this data to the user's browser via an HTTP request. Flask is capable of using Python's built-in json module to read the information stored in the server and pass it on (once again in JSON format) to the requester whenever the correct URL is accessed.

In this project, a single URL is used to pass on the score and audio file paths, the chromagram and DTW parameters, and also the frame equivalences. Because the file paths are later used to fetch the score and audio file in the server — a rather time consuming task — a possible improvement to the interface that is not included in this implementation would be dividing this stage into three separate URLs. This would allow fetching the files and data in parallel [60], which could make the web page load faster.

5.4 Playback page

In the playback page, the *fetch()* method from the Fetch API is responsible for accessing the URL that serves the data and getting the information stored in the response JSON. This information is then used to create the audio navigation system, which can be seen in Figure 5.4

After retrieving the file paths stored inside the information *.json* file, the playback page loads the synthetic version and each one of the human recordings using Howler.js, a Javascript library for playing audio in the browser.

Howler.js is able to load each audio file into a ***Howl*** object, which is then accessible to the browser via Javascript code. All ***Howl*** instances come equipped

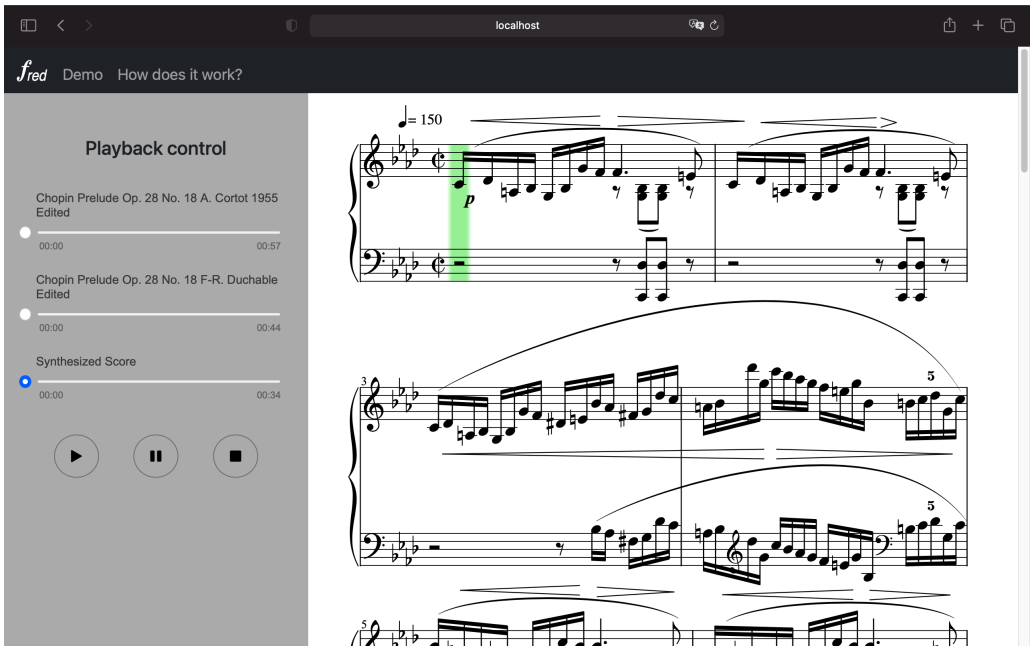


Figure 5.4: Screenshot of the playback page showing the score, the cursor, and the audio navigation controls. Switching can be performed at any time clicking the radio button next to the desired recording, and audio navigation can be made both by clicking in the music sheet and by clicking in the progress bars.

with *play()*, *pause()*, and *stop()* methods[4], which can be used to control the playback of each recording. They also have a *seek()* method, that can be used without any arguments to find out the elapsed playback time in seconds, and with a desired time in seconds argument to go forward or backwards in a playback.

It is important to remember that we are dealing with standard digital audio, meaning that the original sound waves have been converted to a computer readable format by taking uniformly spaced samples of the original signal as described in Section 2.1.1 and in Appendix A. When *seek()* returns a time in seconds, or uses a time in seconds as argument, Howler.js is internally making the conversion between seconds and samples using the audio sampling frequency, which normally comes inside the file's metadata.

In Figure 5.4, the audio navigation controls are displayed in the gray container on the left-hand side of the image. The three round buttons control play, pause, and stop functionalities.

[4]The difference between pause and stop is that pause allows resuming playback from the same instant as before, whereas stop aborts playback sets the audio position to instant zero.

5.4.1 Switching between recordings

The radio buttons below the names of the recordings control which interpretation is being played at the moment. Whenever the user clicks one of these buttons selecting a different recording, the audio switches between the two interpretations with the procedure described below.

First the playback currently running is paused, if it is not already. A flag variable is created to indicate if the the audio needs to be resumed in the new interpretation after the switch.

Then, using *seek()* without any arguments, the current time in seconds is obtained for the audio from where the switch is happening. Because the audio files are all uniformly sampled at $F_s = 1/T_s$, the recordings were created by taking one sample at every T_s seconds from the original sound wave. Hence, for any sample n, its relation to the time it should be played is $t = nT_s$, which implies $n = tF_s$.

Using the sampling frequency stored in the information *.json* file, the current time in seconds is reverted to samples, and then further converted to frames through integer division by the hop length, which is defined as the number of samples within each frame, and is calculated from the overlap using Equation 2.5.

The result from this operation is the current frame of the origin audio in the chromagram. Recall that the DTW provided the frame equivalences between this recording and the target one, and that this information is also stored in the *.json* file.

Using this, the frame in the destination recording that is equivalent to the one in the origin audio can be discovered. Then, this equivalent frame can be converted back to seconds using the inverse of the procedure described above: first the frame is converted to samples by multiplying by the hop length, then the sample is transformed to a time in seconds by dividing by the sampling frequency.

The result is passed to the *seek()* method of the ***Howl*** corresponding to the destination recording, which puts it in the correct position. The final step is using the *play()* method to resume playback in the new recording, if the flag variable indicates that the origin audio was playing before the switch.

5.4.2 Navigating using the progress bars

Next to the radio buttons for selecting the recording, there are white progress bars that are filled as the recordings are played. They show the progress simultaneously in each recording, meaning that they compare the equivalent instants in the audios to their total duration.

In the Javascript code that runs when the playback page is accessed, there is a callback that is responsible for updating the progress bars as the audio plays. The callback function runs at every 10ms, and, for every recording, it finds the time in seconds (after conversion from frames) that is equivalent to the elapsed time in the currently selected recording. With the equivalent instants in hand, this function simply updates the HTML elements of the page using traditional Javascript[5].

The technique used to find the equivalent instants is the same that has been used in interpretation switching, with the difference that there is no need to pause one of the recordings, since the audio that is playing does not change.

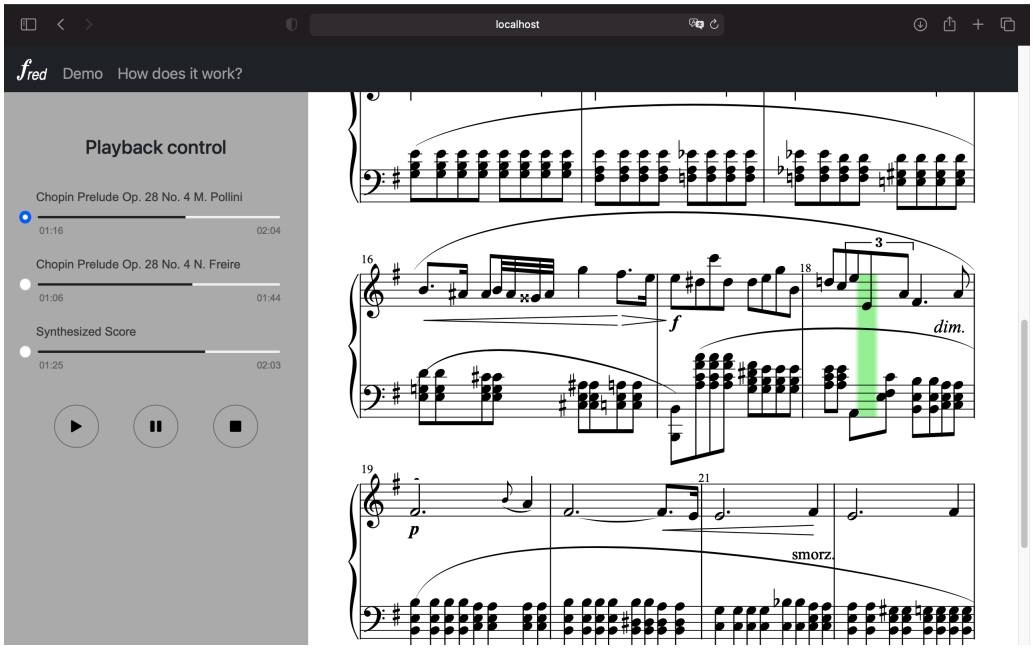


Figure 5.5: Screenshot of the playback page during audio playback. It is possible to see that the progress bars fill simultaneously as the audio plays, thanks to the callback that gets the equivalent times and updates them.

[5]Dynamically changing a page to create new objects, or changing the style of existing elements, as is the case here, is a typical use of Javascript. This is made by manipulating the DOM [61], Javascript's representation of the objects constituting a web page.

It is also possible to control the position of audio playback by clicking in the progress bars, as usual when listening to music on the web. Javascript is capable of handling information from click events happening on the web page, and, using this, discover how far the click in a progress bar was from its left bound.

By dividing this value by the total width of the progress bar, we get the percentage of the audio playback that the user wants to go to. Imagining the width of the progress bars is 10 pixels, if someone clicks in the bar at a position 3 pixels away from its beginning, then this means that the desired position for the audio is at 30% of its total duration.

Having this percentage, it is only a matter of multiplying it by the duration of the recording corresponding to the bar that was clicked to obtain the desired time in seconds for this performance. Then, updating the other progress bars, and even the audio position, if the progress bar that was clicked did not belong to the audio that was playing, can be done using the same method of obtaining the equivalent times as before.

5.4.3 Navigating through the score

The progress bars are not the only dynamic part of the playback page. In this page, the music sheet provided by the user is displayed and followed using the procedures described in Chapter 4.

The page fetches the path for the score MusicXML in the information *.json* file, and OSMD calculates the timestamps using the method seen in Section 4.1.3 as soon as the music sheet is loaded into the page. Using them, score following is done using a callback responsible for periodically checking if the current song's equivalent time in the artificial recording is smaller than the next stored timestamp, and moving the cursor if not the case.

Another layer of interaction with the music sheet was added by enabling navigation through the score. When a user clicks a note, the click coordinates in the page can be transformed in coordinates in the music sheet, as explained in one of the issues of OSMD's GitHub repository [62]. Using this and the convenient *GetNearestNote()* method from OSMD's ***GraphicSheet*** class, the playback page is able to return the object corresponding to the note closest to the click.

With this, in order to scroll the audio to the position of the note, it is only necessary to ensure that the objects used for the graphical representation of the notes are stored along with the timestamps during load time. Then, a list of the notes can be passed on for comparison with this closest note object, and the desired timestamp can be extracted through its association with the graphical note representation.

To ensure that score following can be resumed after using *seek()* to go to the desired time of the correct ***Howl*** object, a copy of the original timestamps calculated before must be kept. In this way, when a user clicks a note that has already passed in the audio, it is possible to restore the elements that were popped out of the array that is used to keep track of the cursor position.

This is how navigation through clicking is implemented in the score, but, as of today, some bugs still seem to occur in the code. For example, sometimes when clicking a note, the cursor will go to the position corresponding to the note coming just before the one that was clicked. This could be a consequence of a difference between the floating point precision of the timestamps and of the time returned by the *seek()* method. Since the algorithm for updating the cursor compares two decimals with different precision, this rounding error could cause small mistakes when navigating via score.

Also, both with the score and the progress bars, there seems to be a problem while navigating forward in the audios, specially if this is done without pausing the audio beforehand. Depending on the instant that is requested, the timestamps array is not correctly updated, causing the score to scroll to the wrong note. This does not happen all the time, and because it is a bug that is not easy to reproduce, it was not possible to find its cause. However, it is probably related to the queuing system in Javascript and how the callbacks for updating the progress bars and cursor are scheduled.

Regardless of these problems, the application still presented a satisfactory performance for its goal of being a minimally viable version of a performance analyzer. The user interface is simple and friendly, and it allows controlling most of the algorithm through its parameters. Navigation through the progress bars and the score is available, even with its problems, and most users should be able to enjoy the application without much trouble.

Chapter 6

Conclusion

In this project, we have seen how a web application for performance analysis in music was developed, and explored in detail the signal processing and information technology subjects necessary to understand how it worked. In the following, we will summarize the points of interest seen in this study, and we will detail some of the possible future perspectives of this work.

6.1 About the developed project

More often than not, music is more about expression and feeling than about the notes that are written in the score. Regardless of the level of experience, every musician has their own approach when playing musical pieces, and it is this individual way of communicating through music that makes a performer's work unique. Because of this, it is common for music students to want to study the styles of different musicians, so that they can learn their expressions and incorporate new tools in their own set of artistic interpretations of phrases and songs.

Normally, analyzing the performance of many artists is a tedious process that involves not only finding a correct music sheet for the desired piece, but also listening to all the versions that must be compared, noting down the parts of interest, and going back and forth between the different recordings to have a feel of the differences between interpretations.

Even though there are nowadays digital music sheet readers that can play scores and navigate through them, there is no solution capable of placing different

recordings next to one another and switching between them as the user wishes. Furthermore, score readers render synthetic versions of the music sheets they receive, and are incapable of score following real recordings or adding interpretative aspects to a song.

Having this in mind, this project proposed to create a minimally viable application that would allow music students to compare different recordings in a convenient and user friendly way, thus filling a feature gap that currently exists with digital score readers. The idea was to create a user interface where it would be simultaneously possible to follow a music score and freely switch between different interpretations of the same work.

Starting from the different representations of a music signal in time, frequency, and in the time-frequency domain, we have seen how to display digital music in a matrix format containing the evolution of the played notes along time. Then, using dynamic time warping, we studied how two matrices corresponding to different recordings could be compared in order to find out the equivalence between instants in both of them, and proposed heuristics to make this equivalence as musically meaningful as possible.

With this information in hand, we were capable of developing a method for freely switching between recordings of the same piece — one of the goals of the project — simply by resuming playback in the equivalent instant of the recording that we were switching into. Despite working in most cases, and being robust to noise and variations in timbre, this method was not foolproof, and in this project we also saw its limitations related to very large interpretative differences and to silences present at the end of commercial recordings.

By synthesizing the score similarly to music sheet readers, we were able to create an artificial recording that could also be aligned with the real performances, creating a ground truth of what would the piece sound like if there were no interpretative variations whatsoever. Combining this with a Javascript library for rendering scores, it was possible to calculate the note onsets in this artificial version, and use this together with DTW to find out the correspondence between the instants of the real recordings and the notes in the score.

Once again, this method performed well for the purpose of creating a minimally viable product, but showed its limitations. The library that was used to render the score, OpenSheetMusicDisplay, does not support mid measure tempo changes when calculating the timestamps, and grace notes could cause problems in alignment since they are interpreted as having zero duration. Moreover, pedal notes seem to be particularly hard to align using the DTW, and it appears that *fermatas* and other notation that may not be correctly synthesized could cause problems in the following procedure.

Both of these two blocks, the interpretation switcher and the score follower, were integrated in a web based user interface that is capable of receiving the inputs necessary for alignment via an HTML form. The final application was developed using the Flask minimalist Python web framework, and the communication between its front and back end was made by sending carefully written JSON files through HTTP requests.

All things considered, the goal of creating a minimal working version of an application for performance analysis was achieved, even though there are limitations to the techniques that were used. Despite its issues, the system behaves as expected most of the time, and it performs correctly as a web application for music education. Both for students who want to learn how to read music, and for experienced musicians or music lovers that wish to compare the playing styles of their favorite artists, the application provides a convenient way to analyze different recordings and deepen their musical knowledge.

6.2 Next steps

The interface that was presented in the previous chapters is available on a GitHub repository [63], and is ready to be deployed in a server to be made available on the internet. This should allow people to begin using the system and generate some feedback on its current state. Because most of the testing was made on Chopin's set of twenty-four preludes, the whole system containing both the interpretation switcher and the score following was baptized as Fred, in a reference to the Polish composer's first name, Frédéric.

Perhaps the most significant weakness of this work is the lack of an objective evaluation of its performance. All tests were evaluated subjectively, and it would be good to obtain some sort of numerical measure of the system's accuracy. To evaluate score following a possible strategy could be annotating the note onsets on a set of recordings of different pieces, and then manually annotating the instants where the cursor hovers over the notes. Because this can be a rather time consuming strategy, the authors of [14] suggest randomly changing the interval between notes of annotated pieces in the MAPS [64] dataset, thus creating artificial examples where the ground truth onset times would be known.

There are some features that could be added to the application to further enhance user experience. The simplest one is the addition of automatic removal of trailing silences. As we saw when discussing both the score follower and the interpretation switcher, the presence of non musical silence can considerably impact alignment performance, and it could be easily handled by establishing a magnitude floor below which all samples in the beginning and end of the recording would be removed.

Another possible feature to be added is part selection. By clicking in different measures of the score, the user could be capable of looping certain parts of the piece in order to easily compare interpretations. Right now this requires navigating using the score or progress bars, and switching between the interpretations at the correct moments, which can be cumbersome after a while.

In terms of visualization, an interesting possibility could also be showing the progression of the pieces with animations displaying the warping path over the cost matrices for each pair of recordings. The idea would be plotting horizontal and vertical bars that would move along the warping path as the audio progresses, creating an animation of the alignment. Using this, it would be possible to clearly see horizontal and vertical patches in the frame equivalences.

Some corrections need to be done as well. As we explained in Chapter 5, there are bugs related to navigating while clicking on the score and to fast forwarding the recordings, specially while the playbacks are not paused. For the former, the main suspect is floating point precision while calculating the timestamps, while for the latter it might be a good idea to closely examine callbacks and their queuing system.

Other possible continuations include modifying the method proposed here in order to obtain better results. In [14], the authors suggest aligning audio to score by first transcribing the recordings to MIDI format using recurrent neural networks (RNNs) and then using the DTW to find the equivalences already in the note domain. They argue that the transcription could be regarded as a learned feature representation, and be the equivalent of the more traditionally used *chroma* features, but with better performance.

An analogous idea to this one would be creating *chroma* representations for the scores directly after parsing them with music21. To do this, we could try to find the note onsets using a similar procedure to the one used by the front end, and then, use unit vectors with magnitude equally spread across the *chromas* present in each frame as columns of an artificial chromagram. This could avoid the issues with MIDI synthesis, and also reduce the overall complexity of the system.

Enhancing *chroma* features is also a possibility. To further improve robustness of *chroma* features, the authors in [65] propose a different method for extracting them. Instead of using a filter bank as we have seen here, they suggest using a neural network trained to find *chroma* representations of chords. As their ultimate goal was chord recognition, their model was trained and tested to perform this task, but this representation could be useful since it seemed to extract clearer *chroma* content in each frame.

In [66] the authors propose combining one-dimensional onset features with traditional *chromas* to create a modified chromagram with high temporal accuracy. Based on the fact that attacks on many instruments result in a sudden energy increase, they suggest dividing the audio signal in pitch inspired subbands, and then using the peaks in each band as the onset instants for each pitch. Pitches are then pooled into *chromas*, similarly to Section 2.2.2, hence creating *chroma* onset (CO) features. These are later further modified and combined with the standard chromagram to produce better results in music synchronization tasks.

An implementation of the *chroma* onset features can be found in [67, 68], where the authors also include a handful of tools for music synchronization that could be useful for this project, including: strictly monotonical warping path calculation [3], and multi-scale dynamic time warping [69].

The first of these two is a modification to the warping path calculation procedure that forces frame equivalences to be strictly monotonical, prohibiting horizontal stretches. This requires potentially skipping some of the frames of the slower recording, but nevertheless could provide good results for the system considering how problematic these patches were in this work.

The second is another variant of the DTW that uses constraint regions calculated using chromagrams derived using larger windows. Audio features are obtained for the recordings using larger windows, resulting in chromagrams with less columns. Then, a warping path is calculated using this coarse resolution, and the result is used as a constraint region for a DTW performed in a more detailed chromagram constructed using a smaller window length. As a consequence, the algorithm becomes less computationally expensive, which can be very useful for long pieces of music.

Finally, it would be interesting to see if there is any change in performance when using other signal representations as the basis for feature extraction, or when using them directly as input to the DTW. To counter the trade off in resolution in time and frequency, the author of [15] suggests combining time-frequency representations of different resolutions to create new high-resolution signal depictions that could be used in different music information retrieval tasks.

Bibliography

[1] Tom Service. Symphony guide: Beethoven's 5th. *The Guardian*, September 2013. `https://www.theguardian.com/music/tomserviceblog/2013/sep/16/symphony-guide-beethoven-fifth-tom-service` (Acessed on 12/08/2021).

[2] Harold C. Schonberg. But would Bach play it like this? *The New York Times*, Aug 1973. `https://www.nytimes.com/1973/08/26/archives/but-would-bach-play-it-like-this-music.html` (Acessed on 12/08/2021).

[3] Meinard Muller. *Fundamentals of Music Processing*. Springer, Erlangen, Germany, 2015.

[4] Alexander Lerch, Claire Arthur, Ashis Pati, and Siddharth Gururani. An interdisciplinary review of music performance analysis. *Transactions of the International Society for Music Information Retrieval*, 1(3):221–245, June 2020.

[5] Merriam-Webster. Fermata. In *Merriam-Webster.com dictionary*. Merriam-Webster, 2021. (Accessed on 10/22/2021).

[6] Hélio M. de Oliveira and Raimundo C. de Oliveira. Understanding MIDI: A painless tutorial on MIDI format, 2017.

[7] Michael Good. MusicXML: An internet-friendly format for sheet music. In *International XML Conference & Expo (XMLEdge)*, pages 12–16, Santa Clara, October 2001. SYS-CON.

[8] Frank Kurth, Meinard Müller, David Damm, Christian Fremerey, Andreas Ribbrock, and Michael Clausen. Syncplayer – an advanced system for multimodal

music access. In *Proceedings of the 5th International Society for Music Information Retrieval Conference (ISMIR)*, pages 381–388, London, UK, September 2005.

[9] Frank Kurth, David Damm, Christian Fremerey, Meinard Müller, and Michael Clausen. A framework for managing multimodal digitized music collections. In *13th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, pages 334–345, Aarhus, Denmark, September 2008.

[10] Verena Konz and Meinard Müller. Introducing the interpretation switcher interface to music education. In *CSEDU 2010 - 2nd International Conference on Computer Supported Education, Proceedings*, pages 135–140, Valencia, Spain, January 2010.

[11] Verena Thomas, David Damm, Christian Fremerey, Michael Clausen, Frank Kurth, and Meinard Müller. Probado music: a multimodal online music library. In *Proceedings of the 38th International Computer Music Conference (ICMC)*, Ljubljana, Slovenia, September 2012.

[12] Chris Cannam. Sonic visualiser. `https://www.sonicvisualiser.org/index.html`. (Accessed on 11/12/2021).

[13] Simon Dixon and Gerhard Widmer. Match: A music alignment tool chest. In *Proceedings of the 6th International Society for Music Information Retrieval Conference (ISMIR)*, pages 492–497, London, UK, September 2005.

[14] Taegyun Kwon, Dasaem Jeong, and Juhan Nam. Audio-to-score alignment of piano music using rnn-based automatic music transcription. In *Proceedings of 14th Sound and Music Computing Conference (SMC)*, July 2017.

[15] Maurício do Vale Madeira da Costa. *Novel Time-Frequency Representations for Music Information Retrieval*. PhD thesis, PEE/COPPE/UFRJ, Rio de Janeiro, Brasil, April 2020.

[16] Musescore BVBA. Free music composition and notation software | musescore. `https://musescore.org/en`. (Accessed on 07/20/2021).

[17] Avid Technology Inc. Music notation software - sibelius. https://www.avid.com/sibelius. (Accessed on 07/20/2021).

[18] MakeMusic Inc. Finale | music notation software that lets you create your way. https://www.finalemusic.com/. (Accessed on 07/20/2021).

[19] Arshia Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *Proceedings of the 44th International Computer Music Conference (ICMC)*, pages 33–40, Belfast, Northern Ireland, August 2008.

[20] Antescofo. Home | metronaut : Music practice app for classical musicians. https://www.metronautapp.com/en. (Accessed on 11/13/2021).

[21] Frank Zalkow, Sebastian Rosenzweig, Johannes Graulich, Lukas Dietz, El Mehdi Lemnaouar, and Meinard Müller. A web-based interface for score following and track switching in choral music. In *Late-Breaking and Demo Session of the International Conference on Music Information Retrieval (ISMIR)*, Paris, France, September 2018.

[22] Python Software Foundation. Python 3.8.11 documentation. https://docs.python.org/3.8/. (Accessed on 07/17/2021).

[23] Brian McFee, Alexandros Metsai, Matt McVicar, Stefan Balke, Carl Thomé, Colin Raffel, Frank Zalkow, Ayoub Malek, Dana, Kyungyun Lee, Oriol Nieto, Dan Ellis, Jack Mason, Eric Battenberg, Scott Seyfarth, Ryuichi Yamamoto, viktorandreevichmorozov, Keunwoo Choi, Josh Moore, Rachel Bittner, Shunsuke Hidaka, Ziyao Wei, nullmightybofo, Darío Hereñú, Fabian-Robert Stöter, Pius Friesch, Adam Weiss, Matt Vollrath, Taewoon Kim, and Thassilo. librosa/librosa: 0.8.1rc2, May 2021.

[24] Simon Haykin and Barry Van Veen. *Signals and Systems*. John Wiley & Sons, Inc., Hoboken, USA, 2nd edition, 2002.

[25] Marina Bosi and Richard E. Goldberg. *Introduction to Digital Audio Coding and Standards*. Kluwer Academic Publishers, Norwell, USA, 2002.

[26] Paulo S. R. Diniz, Eduardo A. B. da Silva, and Sergio L. Netto. *Digital Signal Processing: System Analysis and Design*. Cambridge University Press, Cambridge, UK, 2nd edition, 2010.

[27] Goldwave Inc. Goldwave - audio editor, recorder, converter, restoration, & analysis software. (Accessed on 02/03/2021).

[28] Audacity. Audacity ® | free, open source, cross-platform audio software for multi-track recording and editing. `https://www.audacityteam.org/`. (Accessed on 02/03/2021).

[29] Anssi Klapuri and Manuel Davy. *Signal Processing Methods for Music Transcription*. Springer, New York, USA, 2010.

[30] William Keith Nicholson. *Linear Algebra with Applications*. PWS, 3rd edition, 1995.

[31] Leon Cohen. *Time-Frequency Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, USA, 1995.

[32] Paul E. Black. Manhattan distance. In *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology, 2019. (Accessed on 12/11/2021).

[33] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, 3rd edition, 1976.

[34] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, USA, 4th edition, 1996.

[35] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, Orlando, USA, 3rd edition, 1986.

[36] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, USA, 2nd edition, 2001.

[37] Steven S. Skiena. *The Algorithm Design Manual*. Springer, London, UK, 2nd edition, 2008.

[38] Bernardo Vieira de Miranda. Extração de informação musical – uso de estruturas harmônicas para separação de fontes e de medidas de similaridade para análise de performance. Technical report, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, 2017.

[39] Brian McFee, Colin Raffel, Dawen Liang, Daniel P.W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th Python in Science Conference*, pages 18–25, Austin, July 2015.

[40] Bastian Bechtold. Soundfile — pysoundfile 0.10.3post1-1-g0394588 documentation. `https://pysoundfile.readthedocs.io/en/latest/`. (Accessed on 07/16/2021).

[41] Erik de Castro Lopo. libsndfile. `http://www.mega-nerd.com/libsndfile/`. (Accessed on 07/16/2021).

[42] Fréderic Chopin. *Préludes, Op. 28.* Breitkopf & Härtel, Leipzig, Germany, 1865.

[43] Musescore BVBA. Musescore.com | the world's largest free sheet music catalog and community. `https://musescore.com/`. (Accessed on 07/20/2021).

[44] Mathieu Virbel, Gabriel Pettier, Akshay Arora, Alexander Taylor, Matthew Einhorn, Richard Larkin, Andre Miras, and Mirko Galimberti. Kivy: Cross-platform python framework for nui development, august 2021. (Accessed on 08/30/2021).

[45] Makemusic Inc. Musicxml for exchanging digital sheet music. `https://www.musicxml.com/`. (Accessed on 09/01/2021).

[46] PhonicScore. Open sheet music display. `https://opensheetmusicdisplay.org/`. (Accessed on 09/01/2021).

[47] Mohit Muthanna Cheppudira. Vexflow - html5 music engraving. `https://www.vexflow.com/`. (Accessed on 09/01/2021).

[48] PhonicScore. Github - opensheetmusicdisplay/opensheetmusicdisplay: Opensheetmusicdisplay renders sheet music in musicxml format in your web browser based on vexflow. https://github.com/opensheetmusicdisplay/opensheetmusicdisplay. (Accessed on 09/02/2021).

[49] Michael Cuthbert. music21 documentation — music21 documentation. https://web.mit.edu/music21/doc/index.html. (Accessed on 09/13/2021).

[50] Tom Moebert. Fluidsynth | software synthesizer based on the soundfont 2 specifications. https://www.fluidsynth.org/. (Accessed on 09/14/2021).

[51] Bohumír Zámečník. midi2audio: Play and synthesize midi to audio - easy to use python/cli api to fluidsynth. https://github.com/bzamecnik/midi2audio. (Accessed on 09/14/2021).

[52] Mozilla Developer Network. An overview of http. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. (Accessed on 10/16/2021).

[53] James F. Kurose and Keith Ross. *Computer Networking: A Top Down Approach*. Pearson, Upper Saddle River, 6th edition, 2013.

[54] Mozilla Developer Network. Server-side web frameworks. https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks. (Accessed on 10/16/2021).

[55] Django Software Foundation. The web framework for perfectionists with deadlines | django. https://www.djangoproject.com/. (Accessed on 10/16/2021).

[56] Pallets Projects. Welcome to flask — flask documentation (2.0.x). https://flask.palletsprojects.com/en/2.0.x/. (Accessed on 10/16/2021).

[57] Michael Driscoll. Chapter 25 - decorators — python 101 1.0 documentation. https://python101.pythonlibrary.org/chapter25_decorators.html, 2017. (Accessed on 10/17/2021).

[58] Douglas Crockford. Json. https://www.json.org/json-en.html. (Accessed on 10/18/2021).

[59] Mozilla Developer Network. Using fetch - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. (Accessed on 10/20/2021).

[60] Mozilla Developer Network. Async function - javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function. (Accessed on 12/13/2021).

[61] Mozilla Developer Network. Introduction to the dom - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. (Accessed on 10/22/2021).

[62] Mathias Miller and Simon Schmid. Converting osmd coordinates to svg pixel coordinates · issue #504 · opensheetmusicdisplay/opensheetmusicdisplay · github. https://github.com/opensheetmusicdisplay/opensheetmusicdisplay/issues/504. (Accessed on 11/02/2021).

[63] Bernardo Vieira de Miranda. Fred - music education web app. https://github.com/bvm810/term-paper. (Accessed on 11/02/2021).

[64] Adrien Ycart and Emmanouil Benetos. A-maps: Augmented maps dataset with rhythm and key annotations. In *Late-Breaking and Demo Session of the International Conference on Music Information Retrieval (ISMIR)*, Paris, France, September 2018.

[65] Filip Korzeniowski and Gerhard Widmer. Feature learning for chord recognition: The deep chroma extractor. In *Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR)*, pages 37–43, New York, USA, August 2016.

[66] Sebastian Ewert, Meinard Müller, and Peter Grosche. High resolution audio synchronization using chroma onset features. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, page 1869–1872, Taipei, Taiwan, April 2009.

[67] Meinard Müller, Yigitcan Özer, Michael Krause, Thomas Prätzlich, Frank Zalkow, and Jonathan Driedger. Sync toolbox: A python package for efficient,

robust, and accurate music synchronization. *Journal of Open Source Software (JOSS)*, 6(64), 2021. paper 3434.

[68] Meinard Müller, Yigitcan Özer, Michael Krause, Thomas Prätzlich, Frank Zalkow, and Jonathan Driedger. Sync toolbox - python package with reference implementations for efficient, robust, and accurate music synchronization based on dynamic time warping (dtw). `https://github.com/meinardmueller/synctoolbox`. (Accessed on 12/18/2021).

[69] An Efficient Multiscale Approach to Audio Synchronization. An efficient multiscale approach to audio synchronization. In *Proceedings of the 7th International Conference on Music Information Retrieval (ISMIR)*, pages 192–197, Victoria, Canada, October 2006.

[70] Bhagwandas P. Lathi. *Linear Systems and Signals*. Oxford University Press, New York, USA, 2nd edition, 2009.

[71] Luca Salasnich. *Quantum Physics of Light and Matter*. Springer, Cham, Switzerland, 2014.

[72] Sergey Gurbatov, Oleg Rudenko, and Alexander Saichev. *Waves and Structures in Nonlinear Nondispersive Media: General Theory and Applications to Nonlinear Acoustics*. Springer, Berlin, Germany, 2011.

Appendix A

Fourier representation of signals

In Section 2.1 we briefly explained the basic concepts of signal processing needed to understand the audio features used as input for the interpretation switcher. Since the focus of Chapter 2.2 was more on explaining the construction of *chroma* features and their advantages for audio alignment, rather than on being a reference for Fourier representation of signals, we only showed the intuitions behind the important results for the DFT and STFT, and left out more detailed explanations while pointing to good references for those that might be interested.

In this appendix, we will go a little further in the reasoning behind these two representations, starting from the basic modeling of continuous-time signals, going through the effects of sampling on the spectrum, and finally arriving at the windowing process and the construction of the STFT. Our goal is to present a bit more formally the concepts seen before, while still pointing to the references mentioned earlier for proofs and further details.

A.1 Fourier representations of continuous signals

As explained earlier in Section 2.1.1, signals are nothing more than mathematical functions modeling a physical phenomenon. As a function, a continuous-time signal x can assume values $x(t)$ at any given real time t, like all functions whose domain is $\mathbb{R}$. Supposing that a function is periodic[1], meaning that there

[1]Intuitively, this is equivalent of saying that the signal repeats itself every T time interval.

exists some T for which $x(t) = x(t+T)$ for any t, and also that it respects the Dirichlet conditions [70]:

- x is bounded,
- x has a finite number of local maxima and minima in one period,
- x has a finite number o discontinuities in one period,

a well-known result of real analysis states that x can always be written in the form

$$x(t) = a[0] + \sum_{k=1}^{\infty} a[k]\cos(k\omega_0 t) + \sum_{k=1}^{\infty} b[k]\sin(k\omega_0 t), \tag{A.1}$$

where $a[0]$, $a[k]$, and $b[k]$ are coefficients calculated by [70]

$$a[0] = \frac{1}{T}\int_{-T/2}^{T/2} x(t)dt, \tag{A.2}$$

$$a[k] = \frac{2}{T}\int_{-T/2}^{T/2} x(t)\cos(k\omega_0 t)dt, \tag{A.3}$$

$$b[k] = \frac{2}{T}\int_{-T/2}^{T/2} x(t)\sin(k\omega_0 t)dt, \tag{A.4}$$

and $\omega_0 = 2\pi/T$ is the angular fundamental frequency of the signal, measured in radians per second.

This is known as the Fourier series (FS) of a function, and it is the origin of the idea of representing a signal based on its frequency content, rather than on its temporal characteristics. Just like in the guitar string example of Section 2.1.2, a signal can be decomposed in its basic frequencies, in the case of the example the sine waves corresponding to the pure tones whose sum composes the note.

Due to Euler's formula[2], the Fourier series can also be written in the complex exponential form

$$x(t) = \sum_{k=-\infty}^{\infty} X[k]e^{jk\omega_0 t}, \tag{A.5}$$

with coefficients

$$X[k] = \frac{1}{T}\int_{-T/2}^{T/2} x(t)e^{-jk\omega_0 t}dt. \tag{A.6}$$

[2] $e^{jt} = \cos(t) + j\sin(t)$

The exponential form can be used to derive Fourier representations are not limited to periodic signals. As demonstrated in [24, 70], a non periodic signal can be considered to be a periodic one consisting of replicas of it spaced by zeros, as long as the interval between these repetitions tends to infinity. This intuition can be seen in Figure A.1, where it is also possible to observe that extending the separation between the replicas up to this limit is equivalent to infinitely increasing the period of the extended signal.

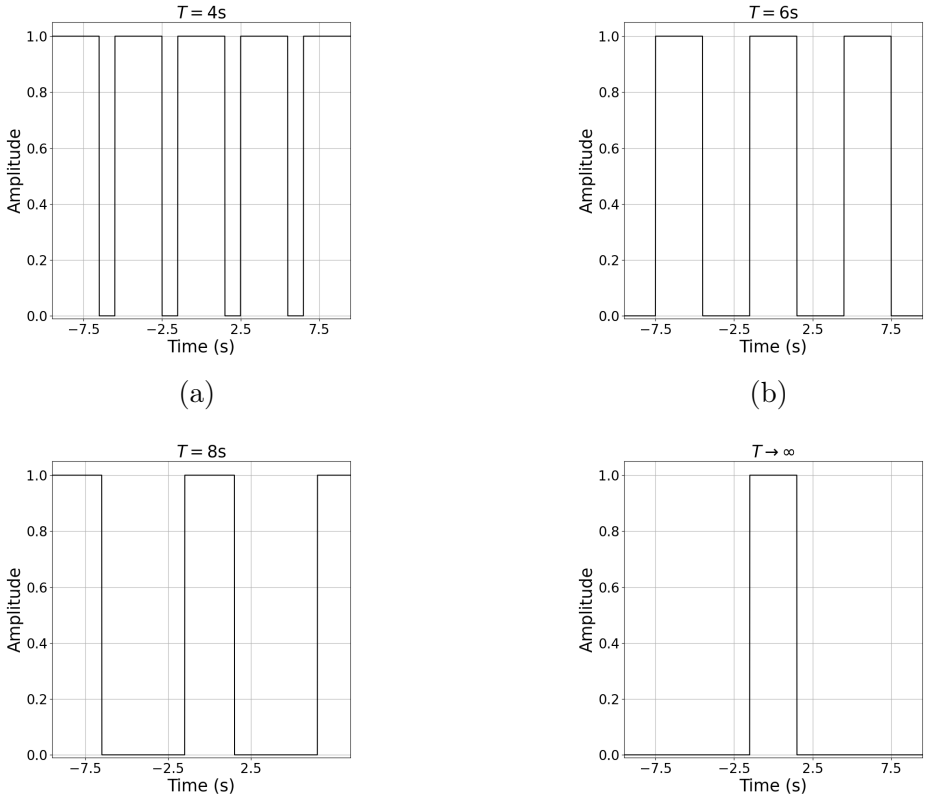


Figure A.1: Periodic signal used as a replacement for a non periodic one. Starting from (a), we can see that as the distance between the replicas of the square wave centered in zero increases, the period gets larger and we get closer to the non periodic rectangular window shown in (d).

By applying the complex Fourier series to the periodic signal while taking the limit $T \to \infty$, a generic frequency representation defined by

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(j\omega) e^{j\omega t} d\omega, \tag{A.7}$$

$$X(j\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt \tag{A.8}$$

and known as the Fourier transform (FT) can be deduced for all signals respecting the Dirichlet conditions for the transform [24]:

- x is absolutely integrable,
- x has a finite number of local maxima, minima, and discontinuities in any finite interval,
- The size of each discontinuity is finite.

Note that, this time, unlike in the Fourier series, ω is continuous and the representation exists for all frequencies. This happens because, as the period goes to infinity, ω_0 becomes infinitely small, and thus summing its multiples is in fact equal to integrating over all possible frequency values.

A.1.1 Relation between duration and bandwidth

The Fourier transform X of a signal x is often called the spectrum of x, and has many interesting properties. For instance, the width of the spectrum, which is also known as bandwidth, is inversely related to the time duration of a signal. Even though it is not simple to analytically define bandwidth or duration for signals infinite in frequency or time, it is possible to establish a bounding relation between what is known as the effective value of these two quantities.

Supposing a signal in time x with values $x(t)$, we can calculate the ratio of the signal's total energy in a single instant by defining a normalized energy density as

$$P(t) = \frac{|x(t)|^2}{\int_{-\infty}^{\infty} |x(t)|^2}. \tag{A.9}$$

This density can be regarded as a probability density function, since it only assumes positive values and integrates to one, and being so, one can calculate moments such as mean and variance for the energy distribution of the signal. With this in mind, the effective time duration T_d of a signal can be defined as the spread of its energy content in time, which can be seen as the standard deviation of P:

$$T_d = \sqrt{\frac{\int_{-\infty}^{\infty} t^2 |x(t)|^2}{\int_{-\infty}^{\infty} |x(t)|^2}}. \tag{A.10}$$

Similarly, in the frequency domain for a signal X with values $X(j\omega)$, we can define a normalized energy density spectrum

$$P(\omega) = \frac{|X(j\omega)|^2}{\int_{-\infty}^{\infty} |X(j\omega)|^2} \tag{A.11}$$

and calculate its effective bandwidth as the spread of $P(\omega)$

$$B_w = \sqrt{\frac{\int_{-\infty}^{\infty} \omega^2 |X(j\omega)|^2}{\int_{-\infty}^{\infty} |X(j\omega)|^2}}. \tag{A.12}$$

These definitions ensure that having large values of x or X for t or ω far from the origin will increase duration or bandwidth, regardless of the value of the signal/spectrum and of the side of the plane the value is on, and furthermore can be adapted to the case of non centered signals [31].

It can be shown that the product $T_d \times B_w \leq \frac{1}{2}$ [24, 31], meaning that even though a signal can be arbitrarily small in time or frequency, it comes at the price of having a larger width on the conjugate domain. This is also valid in discrete time, and explains one of the phenomena intuitively explained in Section 2.1.3. Choosing smaller time frames implies spreading frequency content over more bins due to the lower bound of the duration bandwidth product, creating a resolution trade-off that cannot be avoided.

A.1.2 Modulation and convolution

Another convenient property of the Fourier transform is the relation between convolution and multiplication in the time and frequency domains. Convolution is a mathematical operation between two functions that generates a third one through the sum of the point-wise products of the first input with a mirrored and shifted version of the second, as shown below for two signals in time, but also applicable to frequency representations:

$$(x * y)(t) = \int_{-\infty}^{\infty} x(\tau) y(t - \tau) d\tau. \tag{A.13}$$

Here the asterisk symbol denotes the convolution operation between two signals, and the sum of the products is expressed with an integral due to the fact that each function assumes a continuum of values in time.

As explained in [24, 70], the Fourier transform of the convolution between two signals in time is equivalent to the product of their spectra; conversely, the spectrum of the product between two signals — an operation also called modulation — is (up to a constant factor) equivalent to the convolution between the Fourier transforms of both original signals. In other words:

$$(x * y)(t) \xleftrightarrow{FT} X(j\omega)Y(j\omega), \tag{A.14}$$

$$x(t)y(t) \xleftrightarrow{FT} \frac{1}{2\pi}(X * Y)(j\omega). \tag{A.15}$$

The second relation is the reason why windows introduce frequency content during STFT windowing, as discussed in Section 2.1.3. Because the windowing process consists of multiplying chunks of the original signal by a window function, the spectrum for the windowed signal is actually the convolution of the window and signal spectra. As exemplified in Figure A.2, where the result of the multiplication of a sine wave by a rectangular window is shown in both domains, if the windowing function has non negligible high frequency components, the convolution operation will enlarge the original signal in frequency due to the substantial superposition between the two spectra, thus introducing undesirable frequency spreading in the representation.

A.2 Spectrum of a sampled signal

Until now, we only showed the Fourier representations and some of their properties for continuous-time signals. Yet, as we mentioned throughout Section 2.1, continuous functions cannot be handled neither in time or in frequency by computer processors, and so need to be converted into discrete signals through sampling to be analyzed.

Uniform sampling in the way described in Section 2.1.1 has the effect of replicating the spectrum of the original continuous-time signal along the frequency

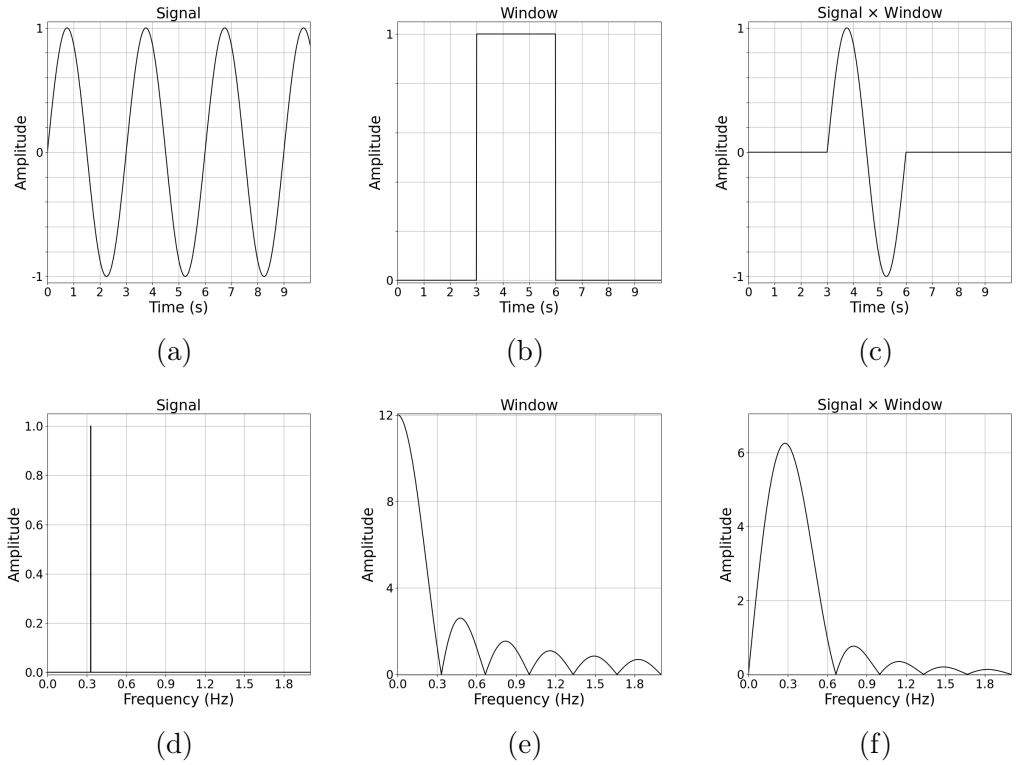


Figure A.2: Effects in both domains of multiplying a sine wave by a rectangular window in time. Images (a) to (c) show signals in time, while (d) to (f) show them in frequency. The spectrum of the sine wave in figure (a) is a Dirac delta, which is introduced in Section A.2 along with some of its properties.

axis, as can be seen in [24, 26]. This is because the act of selecting uniformly spaced samples from a signal can be modeled in continuous time as multiplying it by a sampler signal

$$p(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT_\mathrm{s}), \tag{A.16}$$

where T_s is the desired sampling period, and δ is the Dirac delta function[3], which is a signal defined as zero everywhere except at the origin, while at the same time integrating to one over the real line [24, 71].

In frequency, the representation of the sampler signal P is conveniently also a sequence of shifted Dirac deltas [24, 26], scaled and spaced by $\omega_\mathrm{s} = 2\pi/T_\mathrm{s}$:

$$P(j\omega) = \sum_{k=-\infty}^{\infty} \frac{2\pi}{T_\mathrm{s}} \delta\left(\omega - k\frac{2\pi}{T_\mathrm{s}}\right). \tag{A.17}$$

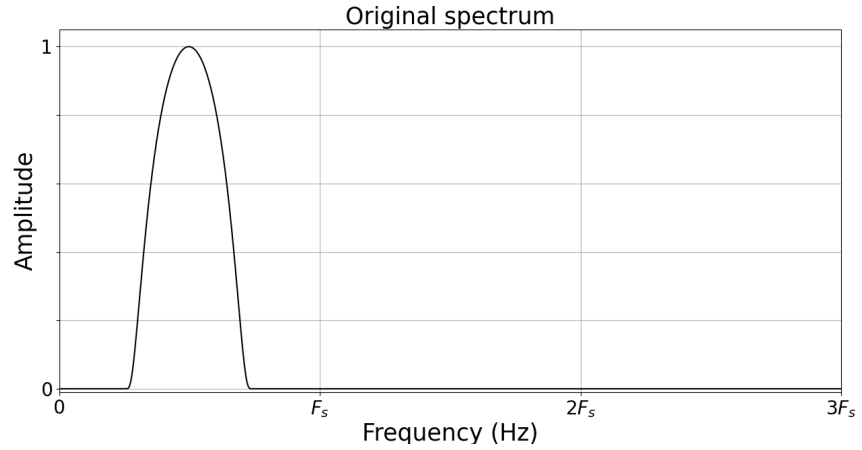
[3]Technically speaking, the Dirac delta is not a function in the traditional sense, but a generalized function [71, 72], which is an extension of the classical definition used in mathematical analysis.

Since convolution is commutative, and since convoluting a signal with a shifted impulse results in a shifted replica of the signal [24, 26], it follows that the spectrum of the continuous-time sampled version of a signal x will be

$$X_\delta(j\omega) = \frac{1}{T_\mathrm{s}} \sum_{k=-\infty}^{\infty} X\left(j\left(\omega - k\frac{2\pi}{T_\mathrm{s}}\right)\right), \tag{A.18}$$

where X is the spectrum of x, therefore showing that uniform sampling in time creates equally spaced replicas of the original spectrum in frequency.

(a)



(b)

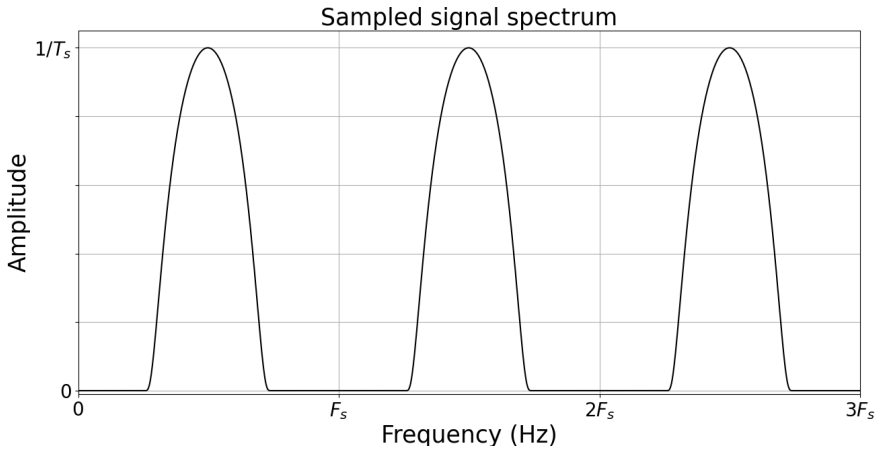


Figure A.3: Example of the effect of uniform sampling on signal with finite spectrum while using sampling period respecting the condition for lossless sampling. Image (a) shows the original spectrum, and (b) the one corresponding to the continuous version of the sampled signal. For convenience, the frequency axis was scaled to hertz by dividing by 2π, so $\omega_\mathrm{s} = 2\pi/T_\mathrm{s}$ becomes F_s.

A.2.1 Fourier representations in discrete time

Still, the resulting spectrum X_δ is the frequency representation of the continuous-time sampled version of the original signal x, defined by $x_\delta(t) = p(t)x(t)$. Even though it has value zero, x_δ is still defined for times $t \neq nT_\mathrm{s}$, and so we need to go a little further to find the Fourier depiction of a discrete signal x_d corresponding to the samples of x with values $x_d[n] = x(nT_\mathrm{s})$.

Similarly to their continuous counterparts, discrete periodic signals can also be accurately portrayed through a series representation, as demonstrated in [24]. Supposing a discrete signal x_d with values $x_d[n]$, and period of N samples, its discrete-time Fourier series, or DTFS in short, is calculated by

$$x_d[n] = \sum_{k=-N/2}^{N/2} X_d[k] e^{jk\Omega_0 n}, \tag{A.19}$$

where $\Omega_0 = 2\pi/N$ is a discrete equivalent of the fundamental angular frequency ω_0, and where the coefficients $X_d[k]$ are equal to

$$X_d[k] = \frac{1}{N} \sum_{n=-N/2}^{N/2} x_d[n] e^{-jk\Omega_0 n}. \tag{A.20}$$

For non periodic signals, the same approach used in continuous time of considering them the limit of periodic versions of themselves when their period is symmetrically extended to infinity applies [24], and a signal x_d can be rewritten thanks to its discrete-time Fourier transform (DTFT) X_d in the form

$$x_d[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} X_d(e^{j\Omega}) e^{j\Omega n} d\Omega, \tag{A.21}$$

where the representation in the frequency domain X_d takes values $X_d(e^{j\Omega})$, and can be found through the relation

$$X_d(e^{j\Omega}) = \sum_{n=-\infty}^{\infty} x_d[n] e^{-j\Omega n}. \tag{A.22}$$

In this case, as previously with the continuous Fourier series and transform, the frequency variable Ω_0 became infinitely small due to the extension of N to

infinity, and being so, the summation over all N becomes in fact an integral over continuous Ω, given that $k\Omega_0$ can now assume any real value. However, unlike before, the frequency representation X_d is calculated through an infinite summation, thanks to the discrete nature of the signal. This demands the care of guaranteeing summation convergence, which is why the signals transformed by the DTFT are usually absolutely summable.

The integral limits in Equation (A.21) come from the fact that, as N goes to infinity, $\Omega_0 N/2$ tends to π, and furthermore it is possible to show that the spectrum calculated through the DTFT is 2π periodic [24, 26]. Because discrete-time sinusoids are indistinguishable if their frequency differs by multiples of 2π[4] [24], all complex exponentials in Equation (A.22) separated by more than that will have an equivalent inside $[-\pi, \pi)$.

Since all the other previously shown properties of the continuous Fourier transform hold for the DTFT with some minor adjustments [24], we use this to bridge the spectrum X_δ of the continuous-time sampled version x_δ of a signal x and the DTFT representation $X_d(e^{j\Omega})$ of a discrete signal x_d.

In order to assure that x_d is the discrete version of x, the relation $x_d[n] = x(nT_\mathrm{s})$, with T_s sampling period, must hold. If this is true, then rewriting x_d and x we have that

$$x_d[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X_d(e^{j\Omega})e^{j\Omega n}d\Omega \tag{A.23}$$

must be equal to

$$x(nT_\mathrm{s}) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(j\omega)e^{j\omega nT_\mathrm{s}}d\omega, \tag{A.24}$$

which can in turn be reformulated using X_δ as

$$x(nT_\mathrm{s}) = \frac{1}{2\pi}\int_{-\omega_\mathrm{s}/2}^{\omega_\mathrm{s}/2} X_\delta(j\omega)e^{j\omega nT_\mathrm{s}} \tag{A.25}$$

because the spectrum X_δ consists of periodic repetitions of X. Equations (A.23) and (A.25) are equal, and so it follows that we must have

$$\Omega = T_\mathrm{s}\omega \tag{A.26}$$

[4]Think of $g[n] = \sin(n\pi/2)$ and $h[n] = \sin(n5\pi/2)$; even though their continuous counterparts are different, in discrete time they are the same due to the samples chosen, because they will have period $N = 4$ and samples $[0, 1, 0, -1]$.

with

$$X_\delta(j\omega)\Big|_{\omega=\frac{\Omega}{T_\mathrm{s}}} = X(e^{j\Omega}). \tag{A.27}$$

Therefore, the spectrum of the discrete version x_d of a continuous-time signal x, will simply consist of repetitions of the original spectrum spaced by 2π. In comparison to the previous result, for a continuous-time sampled version of x, the frequency axis was scaled. Previously, higher frequencies were mapped to close to $\omega_\mathrm{s}/2$, but now since the relation between discrete frequency and continuous frequency is $\Omega = T_\mathrm{s}\omega$, these same frequencies appear close to π, once that $\omega_\mathrm{s} = 2\pi/T_\mathrm{s}$.

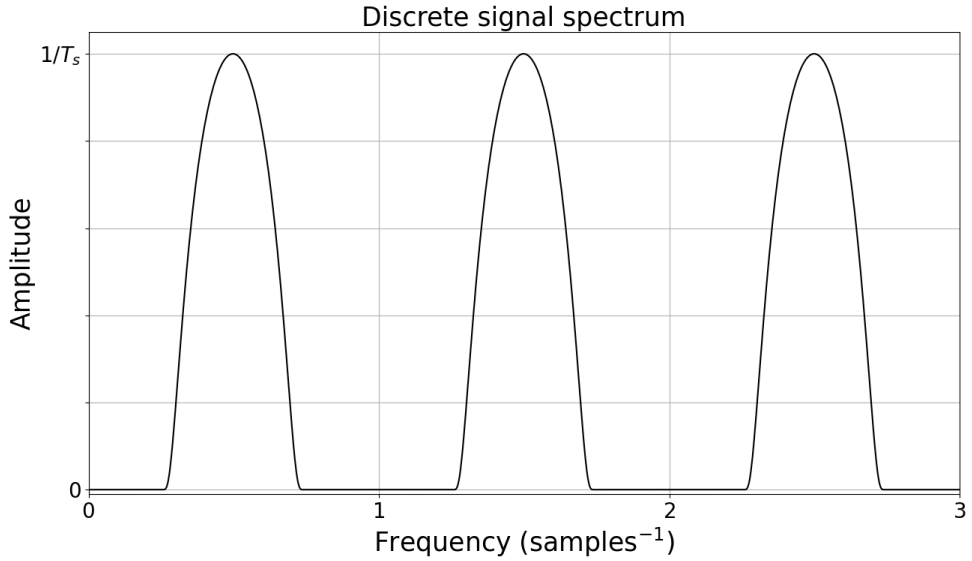


Figure A.4: Continuation of Figure A.3 showing the spectrum of a discrete version of the signal in Figure A.3a. To be coherent with previous figures the frequency axis was normalized by 2π like before with the conversion from radians per second to hertz.

A.2.2 The Discrete Fourier transform

Having found a frequency representation for discrete signals, one part of the problem is solved. It is useful to know that digital signals actually have frequency-domain representations, but the infinite summation and continuous frequency of the DTFT are still troublesome. As mentioned in Section 2.1.2, we need a Fourier transform which is discrete in both domains, otherwise it is impossible to perform signal manipulations in frequency.

Thankfully for us, there exists a Fourier mapping which is discrete in both time and frequency. The discrete Fourier transform (DFT), introduced in Section 2.1.2, is actually a frequency sampled version of the DTFT, as shown in [26].

A DTFT spectrum X of a discrete signal x can be uniformly sampled between zero and 2π with N samples through multiplication by a function similar to the one in Equation (A.16), but in frequency, resulting in a continuous sampled spectrum X' with values

$$X'(e^{j\Omega}) = X(e^{j\Omega}) \sum_{k=-\infty}^{\infty} \delta\left(\Omega - k\frac{2\pi}{N}\right). \tag{A.28}$$

As suggested in [26], the time-domain version x' of this continuous sampled spectrum can be found by applying the inverse DTFT of Equation (A.21) to X', which gives

$$x'[n] = \frac{1}{2\pi} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}kn}, \tag{A.29}$$

where the values $X[k] = X(e^{j\frac{2\pi}{N}k})$ are the N samples extracted from the original spectrum.

On the other hand, by using the convolution property in Equation (A.28), x' can be rewritten as

$$x'[n] = \frac{N}{2\pi} \sum_{p=-\infty}^{\infty} x[n - Np], \tag{A.30}$$

since (as in the case of time sampling) the inverse DTFT of the sampler signal gives a sequence of impulses [26], yielding replicas of x spaced by N samples. Hence, if the number of frequency samples N is bigger than the length L of x, it is possible to perfectly reconstruct x just from samples of its DTFT representation, since that, for $0 \leq n \leq N - 1$, we have

$$x[n] = \frac{2\pi}{N} x'[n]. \tag{A.31}$$

By substituting x' from Equation (A.29) in Equation (A.31), we finally arrive at the transform pair announced in Section 2.1.2, by writing

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}kn} \tag{A.32}$$

and noting that, since $X[k] = X(e^{j\frac{2\pi}{N}k})$, if we assume the minimum acceptable case of $N = L$, then, by using Equation (A.22), we obtain

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}. \tag{A.33}$$

The discrete Fourier transform tells us that, in fact, for finite discrete signals it is possible to find a frequency representation discrete in frequency as well, despite the fact that the spectrum of a sampled signal is continuous. By truncating the summation of the DTFT after the last non-zero samples, we are in fact able to sample the spectrum of the signal without losing any information, which means that a continuous representation is not needed to reconstruct a discrete finite signal in time from the frequency domain.

A.2.2.1 From bins to frequencies

With the representation given by the DFT, frequencies are no longer displayed in radians per sample or samples^{-1}, but in discrete samples called bins. Each bin k corresponds to a discrete domain angular frequency Ω, and consequently also to a continuous domain one ω, and in order to analyze and process signals, we often need to convert from bins to each of those frequencies and the other way around.

In Section 2.1.2, we gave an intuitive explanation using units for converting a bin to a frequency in hertz. Here, based on the results of sampling and of the DFT, we can be more formal by noting that, since $X[k] = X(e^{j\frac{2\pi}{N}k})$, we have that

$$\Omega = \frac{2\pi}{N}k, \tag{A.34}$$

which can be translated into a continuous angular frequency using the relation $\Omega = \omega T_\mathrm{s}$, derived from Equations (A.23) and (A.25), resulting in

$$\omega = \frac{2\pi}{N}kF_\mathrm{s}, \tag{A.35}$$

where $F_s = 1/T_s$, as usual. If we convert from angular frequency to frequency by normalizing by 2π, we obtain Equation (2.3), reproduced here for convenience:

$$f = \frac{kF_s}{N}. \tag{A.36}$$

A.2.2.2 Zero padding

The number of frequency bins N is also a subject of interest for us. When deducing Equation (A.31) we stated that N had to be at least equal to the length L of the original signal, so that the signal x could be periodically reproduced in x', enabling the construction of the DFT.

However, N can be larger than L, meaning that the replicas of x in x' will be more spaced in time. As can be seen in Figure A.5, where we show x' and the DFT for the length $L = 8$ signal:

$$x[n] = \begin{cases} \left(-\frac{1}{2}\right)^n, & \text{for} \quad 0 \leq n < 8 \\ 0, & \text{otherwise,} \end{cases} \tag{A.37}$$

if we make N larger, the discrete Fourier transform will include more samples of the continuous spectrum of x, and so it is possible to say that the DTFT of x will be better approximated by the DFT.

This is called zero padding, and it is a common technique for enhancing the spectrum readability when using the DFT. Since there are algorithms for calculating the DFT that are more efficient on signals whose length is a power of two [26], the standard procedure for using the discrete Fourier transform includes zero padding as a way to get a few more bins while making sure that the signal size is appropriate for calculating the frequency representation in the fastest way possible.

There are two interpretations for zero padding and why it enhances our approximation of the DTFT. The first one is simple, and was already stated here: by adding a few zeros at the end of the signal, we force the use of more coefficients in the DFT, which means more samples of the same spectrum.

The second one comes from the interpretation of the DFT as a midpoint between the discrete-time Fourier series and the DTFT. If we remember the fact

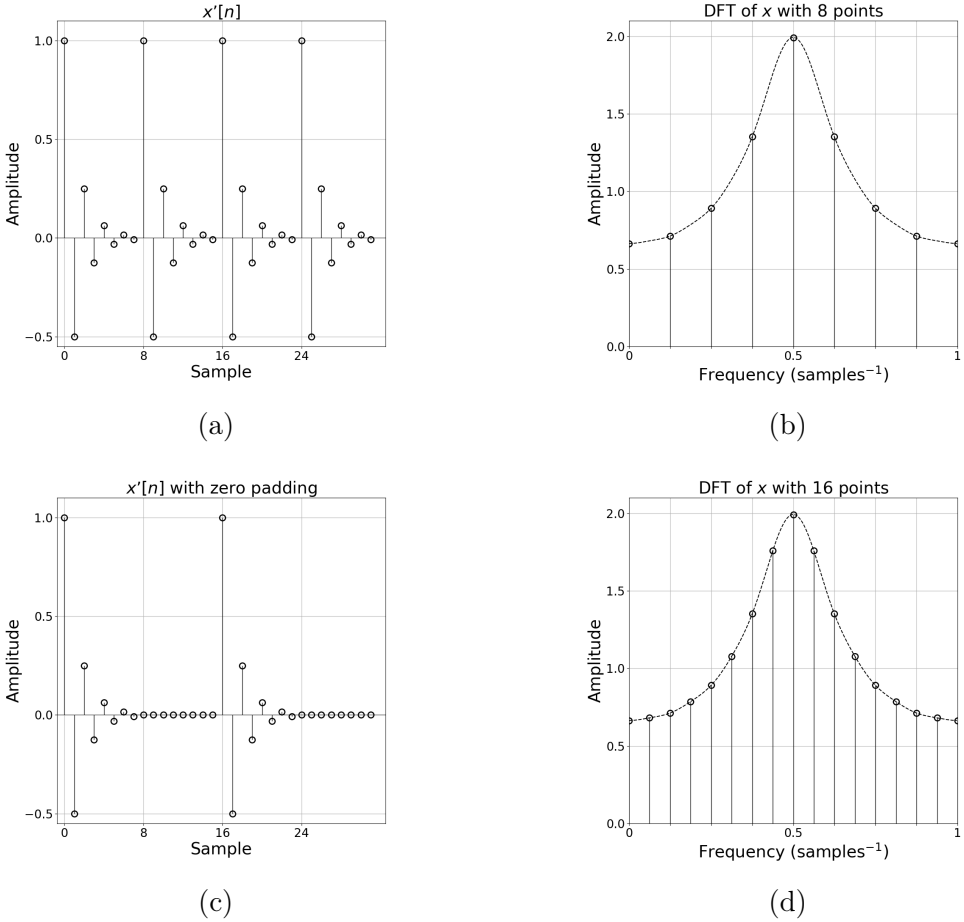


Figure A.5: Effect of zero padding on x' and the DFT. Image (a) shows x' if no zero padding is applied, while (c) shows the padded auxiliary function. (b) and (d) show the DFT of x in both cases, with the continuous DTFT behind them as the dashed line. The frequency axis were converted from bins to samples^{-1} to maintain the units from the previous figures.

that the series is discrete in frequency and consists of a finite number of coefficients, we can interpret the DFT as the series for x', the signal consisting of periodic replicas of x. Knowing that the DTFT is constructed by replicating a non periodic signal over time and extending the interval between the copies to infinity while calculating the series, it is possible to say that zero padding is nothing more than making x' approach this limit, thus increasing frequency sampling and better approximating the continuous spectrum.

A.3 Revisiting the STFT

With a deeper knowledge of the frequency representation in hand, we can revisit the STFT presented in Section 2.1.3 to make precise sense of each one of its parameters. As stated in Chapter 2, the STFT consists of taking a discrete finite signal x, such as a *.wav* file, dividing it in chunks called frames, and applying the DFT to each of them, yielding a matrix where each element of index (i, j) contains the magnitude of frequency bin j at time frame i.

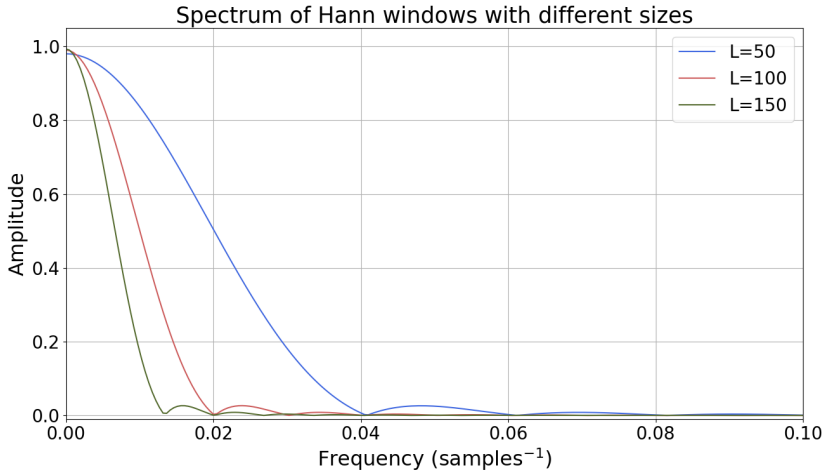
By cutting up the signal into frames, the first two parameters to lookout when calculating the STFT show up. As seen in Figure A.2 and briefly reminded just before, windowing is in fact equivalent to multiplication between a signal and a window function in the time domain, and since time multiplication becomes convolution in frequency, shape and size of a window can in fact enlarge the DFT of a single frame.

Size comes into play because of the trade-off between duration and bandwidth, which causes small windows in time to have large bandwidths, thus spreading the spectrum of the windowed signal. Shape on the other hand, is important because, as a rule of thumb, smoother windows in time tend to have smaller lobes in frequency, diminishing the natural distortion caused by windowing. Both of these effects can be seen on Figures A.6a and A.6b, respectively.

It is very important to note that these two effects are independent. If a frame has very few samples, it does not matter how smooth is the window function; the resulting DFT will be large because of the main window lobe, and convolution will spread frequency content across bins. Similarly, even if a window is large to ensure high frequency resolution, if the chosen window function is discontinuous, there will be lobes with high magnitudes across all the spectrum, and convolution will introduce more frequency content to the original spectrum of the signal being windowed.

The third parameter to be accounted for is the number of samples used in the DFTs. The number of signal samples that will be transformed in each frame is determined by the window size L, but it is perfectly possible to zero pad the windowed signal to obtain a larger number of frequency bins.

(a)



(b)

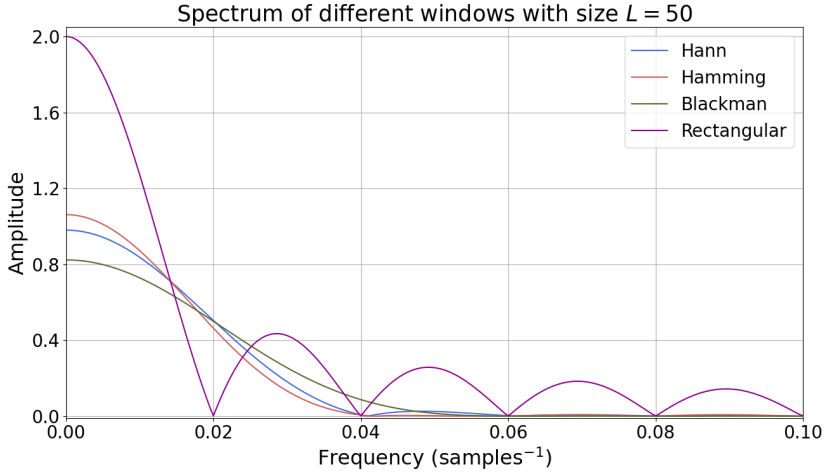


Figure A.6: Images showing the effect of window shape and size in frequency. Image (a) portrays how window length L in samples affects the spectrum, while (b) displays the shape of the spectrum of different windows with size $L = 50$. Here we plotted the DTFT, but the same applies to the DFT since it consists of samples of the former. Frequency axis converted to samples^{-1} as before.

As we explained before when talking about the DFT, this will increase the spectrogram readability since the DTFT will be better approximated, but, once again, this effect is independent from the others. Increasing N will create bins, but if frequency content is already spread due to bad window shape or size, there is no large N that will be able to solve the problem. The number of samples must only be at least larger than the desired frame size, while providing an adequate approximation to the DTFT spectrum. This can be a bit tricky for our website, since window size is chosen in milliseconds while N is in samples, but this was a project decision due to the fact that windows in seconds are easier to size according to the musical piece being provided.

Finally, the last parameter to be seen in a bit more detail is overlap. Since during the windowing process, multiplication by the window function may undesirably scale the signal being analyzed, it is commonplace to overlap some samples between adjacent frames to ensure amplitude in the time domain is not changed.

In frequency, overlap has the advantage of partially solving the time-frequency resolution issue. If better resolution is needed in frequency, but without much loss in the time domain, it is possible to increase frame size while also overlapping more samples between frames. Larger frames do cause loss of time information because all samples inside a frame are converted into a single frequency representation, but overlapping adjacent time frames can compensate for this by duplicating time information into more than one frame. While this does not solve entirely the problem because two windows cannot be fully overlapped, it certainly can help prevent loss due to the trade-off between time and frequency blurring.

These four variables control the source time-frequency representation that was used for interpretation alignment. Alongside the parameters shown in Sections 2.2 and 2.3, they are central to ensure correct alignment, and consequently switching and score following. In this appendix we hope to have helped better understand how they work, and in this process better justify some of the heuristics presented in Section 3.2.2. If the subject of Fourier transforms and signal processing interests the reader, we highly recommend looking at references [24, 26, 70, 31].