Pontificia Universidad Javeriana Cali
Facultad de Ingeniería.
Ingeniería de Sistemas y Computación.
Proyecto de Grado.

Structured and Interactive Improvisation Scenarios

Maria Paula Carrero Rivas

Director: Dr. Jaime Arias Almeida
Co-Director: Dr. Camilo Rueda

13 de Agosto del 2021

Santiago de Cali, 13 de Agosto del 2021.

Señores
Pontificia Universidad Javeriana Cali.
Dr. Gerardo Mauricio Sarria
Director Carrera de Ingeniería de Sistemas y Computación.
Cali.

Cordial Saludo.

Por medio de la presente me permito informarle que la estudiante de Ingeniería de Sistemas y Computación Maria Paula Carrero Rivas (cod: 8911963) trabaja bajo mi dirección en el proyecto de grado titulado "Structured and Interactive Improvisation Scenarios".

Atentamente,

Jaime E. Arias A.

Dr. Jaime Arias Almeida
Co-Director: Dr. Camilo Rueda

Santiago de Cali, 13 de Agosto del 2021.

Señores
Pontificia Universidad Javeriana Cali.
Dr. Gerardo Mauricio Sarria
Director Carrera de Ingeniería de Sistemas y Computación.
Cali.

Cordial Saludo.

Me permito presentar a su consideración el proyecto de grado titulado "Structured and Interactive Improvisation Scenarios" con el fin de cumplir con los requisitos exigidos por la Universidad para posteriormente optar al título de Ingeniero de Sistemas y Computación.

Al firmar aquí, doy fe que entiendo y conozco las directrices para la presentación de trabajos de grado de la Facultad de Ingeniería aprobadas el 26 de Noviembre de 2009, donde se establecen los plazos y normas para el desarrollo del trabajo de grado.

Atentamente,

Maria Paula C.R

Maria Paula Carrero Rivas
Código: 8911963

Acknowledgements

First of all, I want to express my most sincerest gratitude to my supervisor and thesis director Jaime Arias. He has been supporting me since day 1, even before I went to meet him in France, he was always attentive and willing to help me with my doubts about the project. Afterwards, he still supported me in every step of this project, with all the information I needed and the guidance I requested, he always tried his best to solve my doubts and explain concepts in the best way possible. I am incredibly fortunate to have worked with him and to have met him, because his enthusiasm and love for Computer Science helped me understand what my priorities in this field were and what my dreams could be.

Secondly, I want to express my gratitude to Prof. Camilo Rueda. He showed me this project and encouraged me to work with Jaime in this amazing project. He is also one of the teachers who inspired me to join the Research Group AVISPA, and for that I am most grateful. I admire him and think that without him, this project wouldn't have been completed.

I want to express my gratitude to the members of AVISPA, because without them, I wouldn't have found this project. I want to thank them for their astonishing work.

I would like to express my gratitude to l'Université Sorbonne Paris Nord with the project BQR MEASURE and la Maison des Sciences de l'Homme Paris Nord for partially financing this project.

I would like to thank my boyfriend Mateo Valencia, for being always there for me, for encouraging me to be better and to work harder for the things I want, for helping me in every step of the way and for also helping bring this project to life.

Finally, I would like to thank my parents, for being an unconditional support in my life, for giving me the best possible gift: education. For encouraging me to become a better person who is willing to help others in need and for being always there for me.

Abstract

The musical improvisation system VMO-Score has been proposed to mitigate the problems of lack of fluidity in the musical result and use of static improvisation scenarios present in the current mechanisms of musical improvisation. VMO-Score has been successfully tested by different artists, but its use is completely dependent on Max/MSP. This software is private and only works on Windows and OS X operating systems, so the use of VMO-Score is limited to a small audience. The objective of this project is to develop the core components of the VMO-Score system as plugins for the ossia-score sequencer to provide musicians with a completely open-source, multiplatform, ergonomic and independent music improvisation platform allowing (1) the construction of interactive scenarios; (2) the generation of guided transitions; and (3) the interaction between the performers or the environment with the system during improvisation.

Keywords: Factor Oracle, Formal Methods, Interactive Scores, Music Improvisation, ossia-score, Audio Oracle, Variable Markov Oracle.

Contents

1 Introduction **11**
1.1 Problem Statement . . . 12
1.2 Motivation . . . 13
1.2.1 Related Work . . . 13
1.2.2 Formulation . . . 14
1.2.3 Systematization . . . 14
1.3 Objectives . . . 15
1.3.1 General Objective . . . 15
1.3.2 Specific Objectives . . . 15
1.4 Methodology . . . 15
1.5 Contributions . . . 16
1.6 Document Structure . . . 17

2 Preliminaries **19**
2.1 Factor Oracle (FO) . . . 19
2.2 Audio Oracle (AO) . . . 20
2.3 Variable Markov Oracle (VMO) . . . 20
2.4 ossia-score . . . 20
2.5 Clustering . . . 21
2.6 Euclidean distance . . . 22
2.7 Audio Signal Processing . . . 22
2.8 Feature extraction libraries . . . 30

3 Factor Oracle **33**
3.1 Definition . . . 33
3.2 Construction . . . 35
3.3 Implementation Process . . . 37
3.3.1 Current Version of `ossia-score` . . . 40
3.4 Automaton Creation . . . 46

4 Audio Oracle **53**
4.1 Definition . . . 53
4.2 Implementation . . . 55
4.2.1 C++ Feature Extraction Library . . . 56
4.2.2 Implementation with Marsyas . . . 58
4.2.3 Implementation with Gist . . . 63
4.3 Audio Oracle in `ossia-score` . . . 67
4.4 Automaton Creation . . . 73

5 Variable Markov Oracle **77**
5.1 Definition . . . 77
5.2 Implementation . . . 79
5.3 Variable Markov Oracle in `ossia-score` . . . 82
5.4 Graph Creation . . . 88

6 Concluding Remarks **95**
6.1 Overview . . . 95
6.2 Future Work . . . 95

Bibliography **97**

Chapter 1

Introduction

To begin with, Machine Improvisation [AD04] refers to musical performance where computers creatively contribute to the musical outcome. Nowadays, musicians use music analysis tools to automatically generate sample-based accompaniment. This allows computers to generate music that preserves the style of the examples without human intervention.

Lately, artists and researchers have developed different algorithms and software to do musical improvisation, for instance, Shlomo Dubnov and Cheng-i Wang developed the Variable Markov Oracle (VMO) at the University of California at San Diego (UCSD). VMO [WHD16] is an innovative musical improvisational machine that analyzes automatically the musical structure of a recording and extracts the repeated sequences. The most innovative fact about this machine is that the system plays these sequences repetitively according to the characteristics of the rhythm and the harmony found in the recording to improvise in real time or deferred time.

Moreover, in 2016, Jaime Arias, Myriam Desainte-Catherine and Shlomo Dubnov, proposed the VMO-Score [ADCD17] improvisation system (`https://vmo-score.github.io`) which allows to generate *interactive musical pieces* from audio recordings, respecting the logical and temporal structure (a scenario) defined by the musicians, and the musical style of the recording. VMO-Score uses the musical analysis tools provided by the VMO algorithm to generate an interactive scenario that respects the inferred musical structure from a musical recording. This scenario can be controlled dynamically by the interpreters' real-time interactions without violating the structure or constraints defined in the scenario by the composer.

Especially, the scenarios generated by VMO-Score are played by the interactive multimedia sequencer system **ossia-score** (`https://ossia.io`). This interactive sequencer is open-source and allows the definition of logical and temporal constraints, and to write precise process parameters. In addition, it works as well with real-time interactions allowing temporal offsets, conditioned events, or choices giving rise to different interpretations of the same scenario. This system is actively maintained by the Laboratoire Bordelais de Recherche en Informatique (LaBRI), the start-up Blue Yeti, its community of users and the researcher Dr. Jean-Michaël Celerier.

In this chapter, we will introduce the problem that will be tackled in the thesis. The problem is defined based on a set of objectives, defining the scope, the limits of the project and the deliverables. We present all the theoretical framework necessary for the project.

Furthermore, we know that there are a lot of problems regarding musical improvisation, this is why,

we will describe what kind of problems we can encounter in the next section.

1.1 Problem Statement

Nowadays, people tend to use machines to help them alleviate their jobs, hobbies and way of living. These machines such as computers, cellphones and lot of others, can do tasks in seconds that people would do in hours, days or weeks.

On topics such as music, there have been some systems that help musicians make musical improvisation but most of these musical improvisation machines are either programmed in advance to behave according to a specific scenario prepared, or controlled during the show by the performer.

In the first case, the scenario defined in advance is static (a queue of actions), which means the interaction between the machine and the interpreter is completely limited during the improvisation and the output will only depend on the delivered queue of actions.

Moreover, in the second case, the performer's control makes the fluidity of the output less natural and less structured, because as the performer controlling it does not know what is going on inside the machine nor the possible transitions.

As a proposal to alleviate these problems, there is a software that generates *interactive musical pieces* from audio recordings that respect the logical and temporal structure defined by the musicians, this software is the VMO-Score. VMO-Score [ADCD17] is a prototype that has been successfully tested by artists of SCRIME (Creative Studio and Research in Computer Science and Experimental Music) in Bordeaux as well as by the French and American composer and musician Etienne Rolin (e.g., http://bit.ly/2nW9waw and http://bit.ly/2LVvjrS).

However, the current implementation of VMO-Score uses the PyOracle improvisation machine (https://github.com/surgesg/PyOracle) which is no longer maintained and which is completely dependent on Max/MSP software (https://cycling74.com/products/max) to do the necessary sound processing.

Additionally, Max/MSP is a commercial and private software that only works on Windows and OS X computers, which is the reason why the use of VMO-Score is limited to a small audience with the financial ability to buy Max/MSP, plus it forces GNU/Linux users to install virtual machines with Windows and it becomes incredibly hard to handle complex mechanisms to capture and reproduce audio.

Therefore, a new musical improvisation system is needed, one that allows the construction of interactive scenarios, is open-source and can be used easily by anyone. This is our main motivation, creating a new musical improvisation system that can give users more tools to create audio improvisations.

1.2 Motivation

The improvisation in VMO-Score is based on the analysis of audio recording. Thanks to the use of the sequencer ossia-score, performers, musicians and any user can integrate different multimedia systems to complement their improvisation. Some examples are, PureData (https://puredata.info), Max/MSP (https://cycling74.com), OpenFrameworks (https://openframeworks.cc), Processing (https://processing.org), Faust (http://faust.grame.fr), Unity 3D (https://unity3d.com), SuperCollider (https://supercollider.github.io), among others.

In addition to the technical aspects of integration with different multimedia technologies, VMO-Score will also offer a new framework and a new paradigm for musical improvisation. It combines two conceptual approaches: (1) a stochastic time series modeling that captures the properties of the musical surface; and (2) a modeling of the orchestration of multimedia processes by the formal model of Petri net (i.e. the underlying model of ossia-score).

Currently, the project "Rain of Music" (https://scrime.u-bordeaux.fr/Arts-Sciences/Residences2/Residences/Rain-of-Music), supported by the IdEx of the University of Bordeaux in collaboration with the University of Bilbao, aims to create a multimedia system based on robots and drones for artistic performances (eg, music, choreography, scenography). One of these results will be an opera of robots and drones with musicians and soundpainters, so that our project will contribute to solving the question of including interactive improvisation in this show.

Moreover, the result of our project will also have an innovative impact in the industry. For example, the start-up Zuzor (https://zuzor.co) produces interactive media guided by movements. It collaborates with a video game company in visual improvisation and the synthesis of dance and video animated by an augmented reality system. This is why Zuzor finds in VMO-Score a promising system for creating improvisation scenarios that capture such interactions.

Especially, this work is also focused to allow users to interact with all the features that we provide, using an open-source application.

In addition, we must know that there are other improvisation applications or software, that allow users to create audio improvisations. However, some of the systems, that we are going to mention in the next sub-section, do not allow users to interact with all their features without a paid license.

1.2.1 Related Work

- **VMO-Score:**
  VMO-Score [WD15] is a system that helps to generate *interactive musical pieces* from audio recordings and respect the logical and temporal structure (a scenario) defined by the musicians and the musical style of the recording. It is derived from the *Factor Oracle* and *Audio Oracle* structures. It was originally proposed in 2016 by Jaime Arias, Myriam Desainte-Catherine

and Shlomo Dubnov. VMO-Score uses the musical analysis tools provided by the VMO algorithm to generate an interactive scenario that respects the inferred musical structure from a musical recording. This scenario can be controlled dynamically by the interpreters' real-time interactions without violating the structure or constraints defined in the scenario by the composer.

- **PyOracle:**
  PyOracle [SD13] is a music improvisation and analysis system, it was proposed by Greg Surges and Shlomo Dubnov for their Feature Selection and Composition System in 2013. This project is in the family of software built around the Factor Oracle and Audio Oracle. Furthermore, it uses Audio Oracle, which is a graph structure built using features derived from the audio input. The PyOracle can be divided into two parts: PyOracle Analyzer and PyOracle Improviser. PyOracle Analyzer is a Python library for Audio Oracle analysis of music. Moreover, PyOracle Analyzer contains generative functions. PyOracle Improviser embeds PyOracle Analyzer enabling real-time analysis of audio input and generation of new musical content based on that input. The PyOracle Improviser is a software that improvises learning from the performer style and at the same time, it generates audio accompaniment, doing everything in real-time. This software has the capacity of modifying the accompaniment, in real-time, or according to a predefined script, in some parameters, constraints, and probabilities.

- **Improtek:**
  Improtek [NCA17] is a music improvisation software, it is dedicated mainly to guide or compose music improvisation. This software uses a "scenario" which is an integration of temporal specifications defined by the user before using it. Moreover, it is recognized because it covers the various levels involved in machine improvisation, such as, the incorporation of anticipation at a symbolic level in a directed generation process relative to a predefined structure, an architecture combining this anticipation with reactivity, and an audio rendering module performing live re-injection of captured material.

Consequently, after seeing the different musical improvisation systems and our motivation, we can ask ourselves, what do we want to accomplish with this musical improvisation system, taking into account its most important characteristics.

1.2.2 Formulation

Is there a multi-platform and interactive musical improvisation system that allows interpreters to interact with it in real-time?

1.2.3 Systematization

- What characteristics should an interactive musical improvisation system have to be able to allow the interpreter to interact with it?
- How should the integration of Factor Oracle and Audio Oracle with ossia-score be?

- How must one validate that the VMO-Score is well developed?
- What is the value of this software for interpreters?

Therefore, with the proposed questions in the Formulation sub-section, we can introduce the objectives of our project.

1.3 Objectives

1.3.1 General Objective

The goal of this project is to develop VMO-Score as an ossia-score sequencer plugin. In this way, the musicians will have **a completely open-source, multi-platform, user-friendly and independent musical improvisation system**, allowing the construction of interactive scenarios generating guided transitions on the musical material extracted from a recording and (2) allowing interpreters or the environment to interact with the system during improvisation, maintaining the logical-temporal relationships imposed by the composer.

1.3.2 Specific Objectives

1. Develop, document and test the Factor Oracle in C++ for symbols.
2. Integrate the Factor Oracle implementation as an ossia-score plugin.
3. Develop, document and test the Audio Oracle in C++ based on the Factor Oracle.
4. Integrate the Audio Oracle implementation as an ossia-score plugin.
5. Develop, document and test the Variable Markov Oracle (VMO) in C++ based on the Audio Oracle.
6. Integrate the VMO implementation as an ossia-score plugin.
7. Extend VMO-Score in order to use the VMO ossia-score plugin.

After describing our objectives, we will show the methodology that was used to achieve them and what were our accomplishments based on these objectives.

1.4 Methodology

We used the agile methodology, Scrum. This approach allowed for (1) a fast and constant delivering (each 2 weeks) of new features; (2) taking into account the feedback of the users (researchers and musicians) during all the stages of development and (3) working with the users and developers together.

This methodology not only allowed the efficient transmission of information between users and developers, but it also allowed the measurement of the progress in the development, thanks to the

delivered features. In fact, the excellent relationship and environment between the participants of the project allowed to have an organized and efficient team able to deliver a simple and high quality product. Moreover, this methodology is used by the supervisor of the thesis in his laboratory (LIPN) for the development of the software of his research team, as well as the LaBRI, for the development of ossia-score. In this regard, the project has an excellent working environment for its development.

The proposed methodology is orchestrated with time-limited meetings. Each two weeks (sprint meeting), the development team, Jaime Arias and Maria Paula Carrero Rivas, chose the main features that were to be implemented in this interval of time. Finally, each two/three days, the development team met for 15 minutes maximum to answer three important questions: *What features did I finish? What features will I finish for the next meeting? What are the problems that delay the development?*

With this methodology, firstly, we started analyzing the Factor Oracle papers [AD04] and [LLA03], after understanding the algorithm, we programmed the Factor Oracle algorithm in C++ and then, we integrated it with `ossia-score` as a plug-in.

Secondly, we analyzed the Audio Oracle papers [DAC07] and [DAC11], afterwards, we started implementing the Audio Oracle algorithm, but for it to work we needed to find a C++ library for audio analysis and feature extraction. When the library was defined, we finished the C++ implementation and finally, we integrated it with `ossia-score` as an addon.

Thirdly, we used the bases of the Audio Oracle to start the Variable Markov Oracle, subsequently, we read and analyzed the VMO papers [WHD15], [WHD16] and [WD15]. Then, we did the VMO implementation in C++ and ultimately, we integrated the VMO with `ossia-score` as an addon.

As a result, after the development process, we can present the final contributions of our project.

1.5 Contributions

Therefore, after seeing the tools and systems that exist for musical improvisations, the results that were obtained from the project were the implementation of the Factor Oracle in **ossia-score** as a plug-in, in total there were two plug-ins for the Factor Oracle, one for symbol inputs and another one for MIDI inputs.

Additionally, we implemented the Audio Oracle and Variable Markov Oracle as addons, these concepts will be explained in the document.

The importance of our results, is that they give **ossia-score** users new benefits while using the application, they can create symbol or MIDI improvisations using the Factor Oracle, or create audio improvisations using the Audio Oracle or Variable Markov Oracle.

Consequently, this is important because the users can create with improvisations more easily and

they have new features in the application, which give them new opportunities to explore sequences, audios and different audio features. To conclude main contributions associated with this project are three plugins in C++:

- Factor Oracle plug-in in `ossia-score` with its corresponding documentation and tests.
- Audio Oracle addon in `ossia-score` with its corresponding documentation and tests.
- Variable Markov Oracle addon in `ossia-score` with its corresponding documentation and tests.

These plugins are open-source and available on the sequencer's official repository (`https://github.com/OSSIA/score`). In this regard, users will be able to download ossia-score binary with VMO-Score or compile it themselves. Users will also be able to contribute to its constant improvement through their feedback, as is the case with the success of ossia-score.

Finally, to give a summary of our project, we list our Document Structure.

1.6 Document Structure

In what follows, we describe the structure of this document.

Chapter 2 [Preliminaries]. In this chapter, we establish the basic concepts and terminology used in this document. We describe the main data structures used in this project, the Factor Oracle, Audio Oracle and Variable Markov Oracle. We then introduce the ossia-score application, music and audio definitions and finally the C++ feature extraction libraries used in the project.

Chapter 3 [Factor Oracle]. This chapter presents the Factor Oracle and the definitions and algorithms it includes. Moreover, this chapter explains the implementation process with the different versions of `ossia-score`, the problems and obstacles we encountered and the solutions that were implemented. It also includes automaton examples created with our implementation.

Chapter 4 [Audio Oracle]. In this chapter we introduce the Audio Oracle, its definitions and algorithms. Furthermore, we introduce the C++ feature extraction libraries that were used during the implementation process. We also explain the implementation process and the obstacles that we faced. Finally, we show the different audio improvisations and automata that were generated using our implementation.

Chapter 5 [Variable Markov Oracle]. In this chapter we first introduce the Variable Markov Oracle, its definitions and algorithms. Additionally, we explain the implementation process, including the `ossia-score` integration. Also, we show the different audio improvisations and graphs that were generated using our implementation.

Chapter 6 [Concluding Remarks]. This chapter presents the accomplished results with this project and gives an idea of possible future work.

CHAPTER 2

Preliminaries

In this chapter, we include the necessary material for the understanding and development of the proposed work, covering relevant concepts and definitions for it. We describe each of the main data structures and algorithms that were implemented, such as, Factor Oracle (FO), Audio Oracle (AO) and Variable Markov Oracle (VMO). Moreover, we present ossia-score, which is the application where the above algorithms were implemented as plug-ins. We also introduce some music and audio terminology, which are needed to understand the features extracted in the AO and VMO. Finally, we present some feature extraction libraries, for example, Marsyas and Gist. We encourage the reader to read this chapter to understand different concepts that are needed to have a better understanding of this work.

2.1 Factor Oracle (FO)

The Factor Oracle (FO) [LLA03] is a data structure that allows to store suffixes efficiently because of its improved algorithm. It is an on-line linear heuristic method which finds the repeated substrings in a string. FO accepts a sequence of symbols, strings or words, then creates an automaton that contains forward and backward transitions connecting states that have repeated substrings or subsequences equal to each other. The output of the algorithm is an automaton, which identifies the longest repeated subsequence in a sequence of symbols, strings or words.

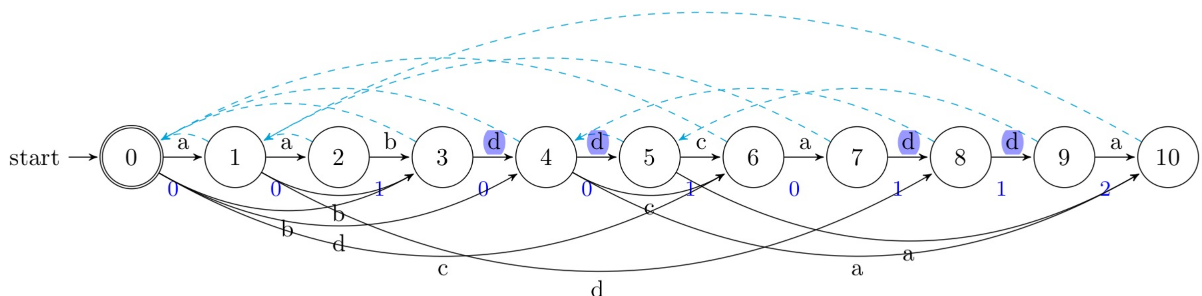


Figure 2.1: Factor Oracle automaton

Here we give an example of the automaton created by the input `a`, `a`, `b`, `d`, `d`, `c`, `a`, `d`, `d`, `a`. The longest repeated subsequence is `d`, `d`, which is highlighted in purple in Figure 2.1.

Moreover, complexity of the FO algorithm is O(n) when using the improved implementation. The reader can find more information regarding the implementation of the FO in Chapter 3.

2.2 Audio Oracle (AO)

The Audio Oracle (AO) [DAC07] is an indexing structure for audio data that captures repeating sub-clips of variable length called “audio factors”. AO accepts a continuous (audio) signal stream as input, transforms it into a series of characteristic vectors, each of these vectors is created extracting a specific audio feature, and submits those vectors to AO analysis. AO outputs an automaton that consists of suggestions to different locations within the audio data that fulfill certain similarity criterion. This similarity criterion, is defined using a particular threshold depending on the feature that wants to be extracted, as determined through the algorithm in [DAC11]. Chapter 4 presents the implementation of the AO in more detail.

2.3 Variable Markov Oracle (VMO)

The Variable Markov Oracle (VMO) [WD15], is the successor of the FO and AO. This data structure is very similar to FO and AO as it resembles the suffix structure used in FO. The difference is that it does not have symbols attached to states. The VMO improves AO by especially assigning labels to frames connected by suffix links during AO construction, creating clusters to organize the similarity of audio feature vectors. We present in Chapter 5 more information regarding the implementation of the VMO.

2.4 ossia-score

ossia-score (`https://ossia.io/`) is a free and open-source intermedia sequencer. It is a software that allows users to create flexible and interactive scenarios. Its main purpose is to be used for live performance, art installations and any context or situation requiring a precise and interactive execution of timed events [CR00]. Its strength lies in its timeline-based precise automation authoring, its incredible user-friendly experience (see Figure 2.2[1]), its plugins and documentation [Cel].

The integration of the FO with ossia-score is primordial because it is the way users can interact with our FO implementation. As mentioned before, ossia-score has a user-friendly interface, this helps users know how to use our algorithm as a plugin. Moreover, ossia-score will be integrated with AO and VMO. As we saw above, these two algorithms work with audio input, thus the user will have the ability to input any audio they want, and ossia-score will create an audio output according to it.

[1] ossia-score UI, `https://github.com/ossia/score`

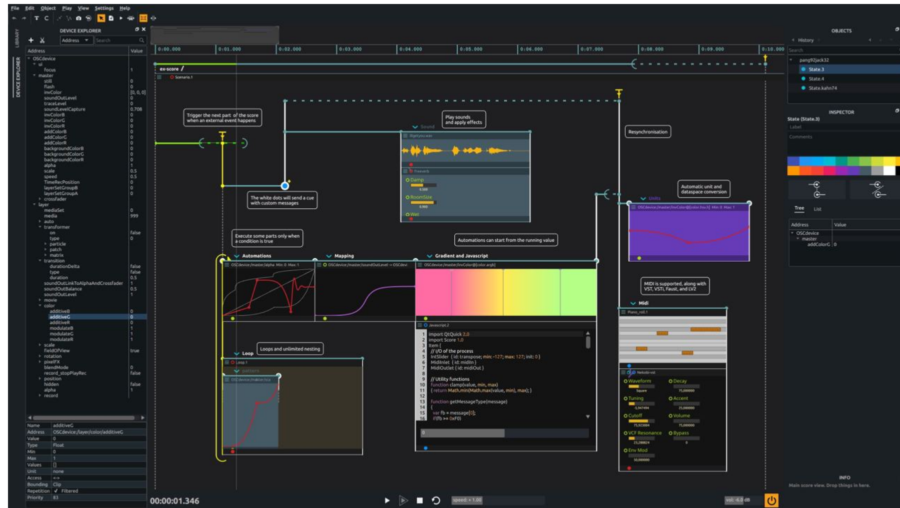


Figure 2.2: ossia-score *UI*[1]

2.5 Clustering

Clustering is the process of grouping a set of objects so that those in the same group (called a cluster) are more similar (in some sense) to those in other groups (clusters). It is a common methodology for statistical data analysis used in many domains, including pattern recognition, image analysis, information retrieval, bioinformatics, data compression, computer graphics, and machine learning, and is one of the main objectives of exploratory data analysis. [2] A general example of clustering can be seen in 2.3, where we see items that are similar get grouped together.

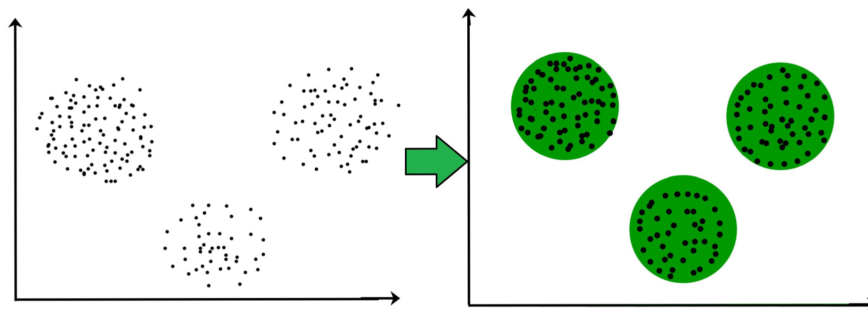


Figure 2.3: Clustering process [Sur20]

[2]Developers Google. What is Clustering?. https://developers.google.com/machine-learning/clustering/overview

2.6 Euclidean distance

The length of a line segment connecting two locations in Euclidean space is called the Euclidean distance[3]. It is sometimes referred to as the Pythagorean distance, since it can be determined from the Cartesian coordinates of the points using the Pythagorean theorem like in the Figure 2.4[4].

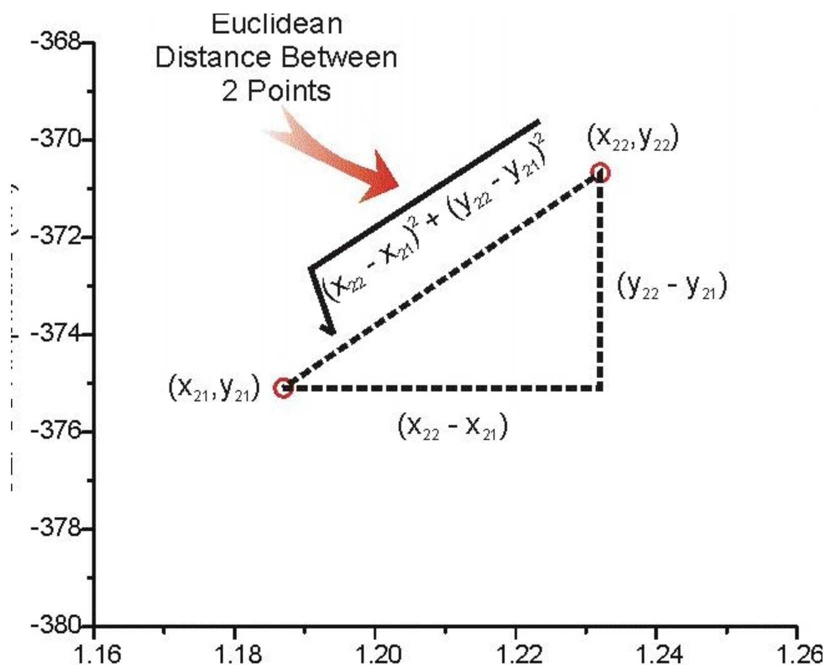


Figure 2.4: Euclidean *distance*[4]

2.7 Audio Signal Processing

An audio signal [Ler12] is an electronic representation of sound waves—longitudinal waves which travel through air, consisting of compression and rarefaction. Audio signals can be represented in digital or analog format, processing can occur in either domain. The differences between these two are, digital processors operate mathematically on its digital representation, while analog processors operate on the electrical signal directly.

Audio signal processing [Ler12] is a sub-field of signal processing, that works especially with the electronic manipulation of audio signals. In the following, we present some important concepts.

[3]ScienceDirect. Euclidean Distance. https://www.sciencedirect.com/topics/mathematics/euclidean-distance

[4]Euclidean distance. https://vitalflux.com/different-types-of-distance-measures-in-machine-learning/

Hanning window: The Hanning window[5], Hanning Function or Hanning, was given its name after Julius von Hann probably because of the Hamming window's linguistic and formulaic similarities. Another name it is known for is the raised cosine, because the zero-phase version,$w_0(n)$, is one lobe of an elevated cosine function. This function is known as a member of the power-of-sine and cosine-sum families. Moreover, the Hanning Window touches zero at both ends of its function, this removes any discontinuity there could be, unlike the Hamming Window, which stops just shy of zero and can create discontinuity (see Figure 2.5[6]). Moreover, the math equation [7] is presented below:

$$w(n) = 0.5 - 0.5cos(\frac{2\pi n}{M-1}), 0 \leqslant n \leqslant M-1$$

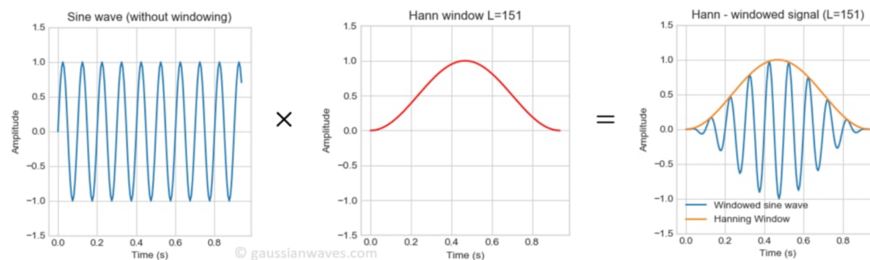


Figure 2.5: Hanning Window effect[6]

Digital audio: In a digital audio system[8], an analog electrical signal representing sound is converted into a digital signal using an analog-to-digital converter (ADC) as in Figure 2.6[9]. This digital signal can be captured, edited, manipulated, and copied by using computers, audio playback machines, and other digital instruments. A digital-to-analog converter (DAC) converts a digital signal back into an analog signal, which is then sent through an audio power amplifier and finally to a loudspeaker. This process is carried out when a user wants to listen, for instance, a sound file.

Buffering: Buffering[10] in audio refers to preloading data into a memory space that has been set aside. Buffering, on the other hand, refers to the process of downloading a particular quantity of data before beginning to listen to music or watch a movie via the Internet.

Audio buffer: Buffers are audio storage containers that we can change in real time in a variety of ways. Once audio is contained within a buffer, it can be played back at different speeds, looped,

[5]Center for Computer Research in Music and Acoustics (CCRMA), Stanford University. Hann or Hanning or Raised Cosine. https://ccrma.stanford.edu/~jos/sasp/Hann_Hanning_Raised_Cosine.html

[6]Hanning Window effect. https://www.gaussianwaves.com/2020/09/window-function-figure-of-merits/

[7]Mathworks. Hann. https://www.mathworks.com/help/signal/ref/hann.html

[8]Digital audio. https://en.wikipedia.org/wiki/Digital_audio

[9]Analog to digital process. https://soundcheck.com.mx/algunos-detalles-del-sample-rate/

[10]Definition of buffering, https://www.pcmag.com/encyclopedia/term/buffering

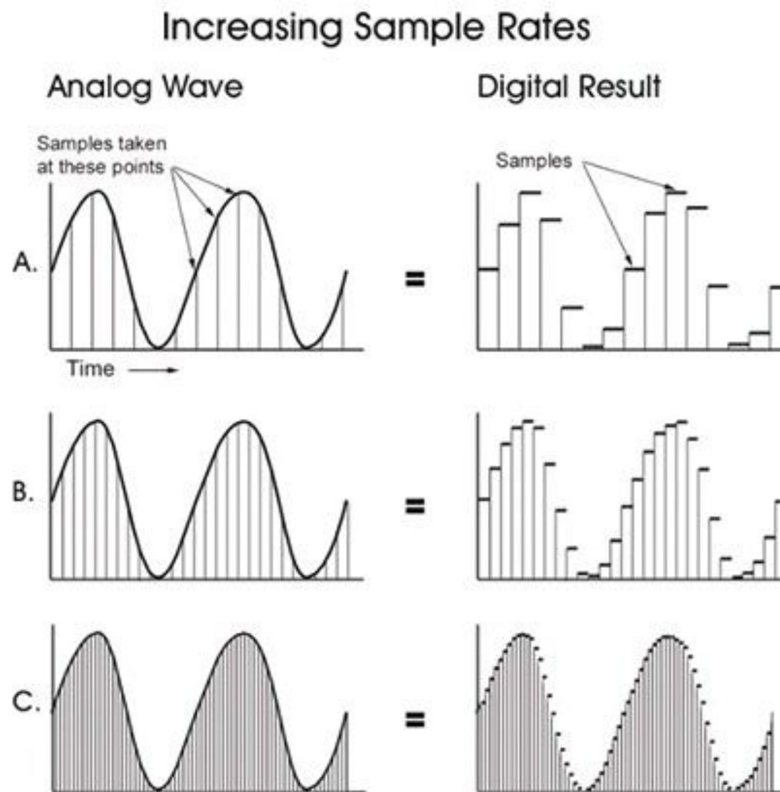


Figure 2.6: Analog to digital audio process[9]

reversed, and subjected to other modifications [Man11].

Latency: The time it takes for an audio signal to enter and exit a system is referred to as audio latency[11]. Buffering, signal processing and conversion, air density, and transmission speed are all possible contributors to this occurrence. Digital editing and the use of audio equipment can help reduce latency.

Buffer size: The buffer size specifies how much time your computer has to process the audio from your sound card or audio interface[12].

Window size: The window size, also known as the time record, is the amount of time during which a waveform is sampled and is expressed in samples. For example, similar to Figure 2.7, a

[11]Latency Definition - What is Latency?, https://backtracks.fm/resources/podcast-dictionary/latency
[12]Sample Rate, Bit Depth & Buffer Size Explained, https://support.focusrite.com/hc/en-gb/articles/115004120965-Sample-Rate-Bit-Depth-Buffer-Size-Explained

time record of 128 samples x 1/48000 seconds = 0,0027 seconds or 2,7 milliseconds is equal to a window size of 128 samples at a sampling rate of 48 kHz [vV13].

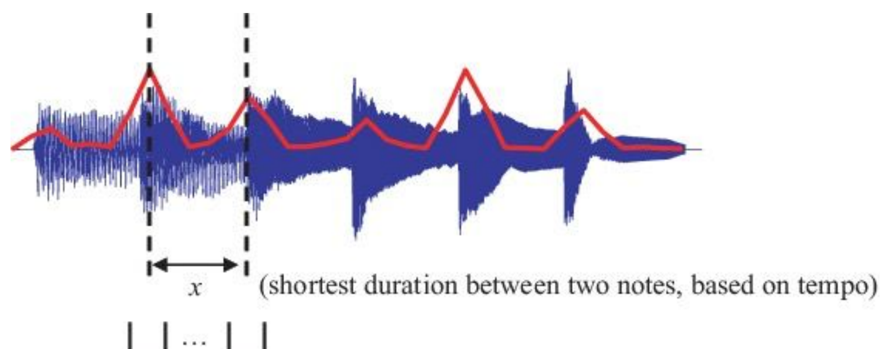


Figure 2.7: Window size representation[CC21]

Layering: Simultaneously recording or playing a musical part using numerous comparable sound patches[13].

Audio gain: A unit of measurement for audio loudness is gain[14]. It also refers to the input volume from the recording source, as opposed to volume, which relates to the final output volume. To put it another way, gain controls the signal's volume before it passes through mixing and recording devices, whereas volume refers to the signal's final output level as it comes out of headphones, speakers or other audio playback device.

Sample rate: The sample rate is the number of samples (or snapshots taken) per second obtained by a digital audio instrument in audio production. The frequency is measured in hertz (Hz) or thousands of hertz (kHz). For instance, the standard sample rate of a Compact Disc (CD) recording, 44100 times per second, would be expressed as 44100 Hz or 44.1 KHz using this method. Note that increasing sample rates generally increases the digital file size (see Figure 2.8[15]). There is a trade-off which occasionally reaches a point of diminishing returns for the particular audio use case[16].
In audio, the highest frequency that can be reliably captured based on the sampling rate is the highest frequency. It is based on the Nyquist Theory[17], which can be applied to a variety of fields where data is collected. The Nyquist Theory, in general, is the smallest number of resolution elements required to accurately characterize or sample a signal. Enough samples must be recorded to capture the peaks and troughs of the original waveform in order to rebuild (interpolate) a signal

[13] Layering, https://www.freemusicdictionary.com/definition/layering/
[14] Gain Definition - What is Gain?, https://backtracks.fm/resources/podcast-dictionary/gain
[15] Sample Rate, https://www.realhd-audio.com/?p=2866
[16] Sample Rate Definition - What is Sample Rate?, https://backtracks.fm/resources/podcast-dictionary/sample+rate
[17] Nyquist Theory, https://www.sweetwater.com/insync/nyquist-frequency/

from a sequence of samples. The Nyquist Frequency in digital audio is half of the sampling rate. The Nyquist frequency of a digital recording with a sample rate of 44.1kHz, for example, is 22.050kHz.

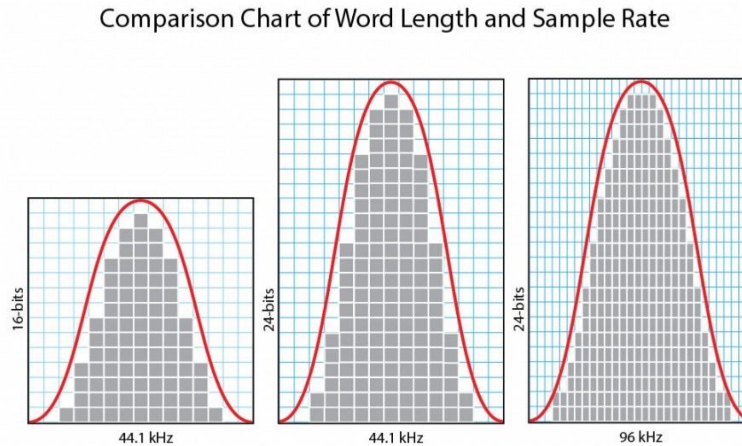


Figure 2.8: Sample rate[15]

Audio channel: A storage "location" for a certain track in a piece of audio is referred to as an audio channel[18]. In analog recording, for example, you would use a 4-tape recorder, which would give you four independent channels. Software allows you to record in several channels while using digital recording. Individual tuning of distinct components of an audio piece is possible when recording in multiple channels. For example, a band with guitar, bass, drums, and a vocalist might record in four channels so that each instrument can be mixed and tweaked separately before the channels are combined to produce a single piece of music. The number of channels in which a piece of audio is played is also referred to as "channel", with stereo audio being "two channels", and surround sound being four, five, or more channels. This is, however, a slang term for "channel," not the traditional definition.

Root Mean Square: The average power of an electrical signal is measured using Root Mean Square (RMS). It is sometimes used in audio to figure out how loud an audio transmission is on average. The word RMS derives from an equation that squares a signal's amplitude, calculates the average mean of the squared amplitude, and then takes the square root of that value. This yields a value that is comparable to the average power of an audio signal across its full length[19].

[18] Channel Definition - What is a Channel?, https://backtracks.fm/resources/podcast-dictionary/channel

[19] ROOT MEAN SQUARE Definition - What is ROOT MEAN SQUARE?, https://backtracks.fm/resources/podcast-dictionary/root+mean+square

Spectrum: Humans can hear frequencies in the range of 20 Hz to 20,000 Hz. This range is known as the audio spectrum, and it may be effectively divided into seven frequency bands, each of which has a particular impact on the overall sound [Wal20]. The seven frequency bands are shown in Figure 2.9[20].

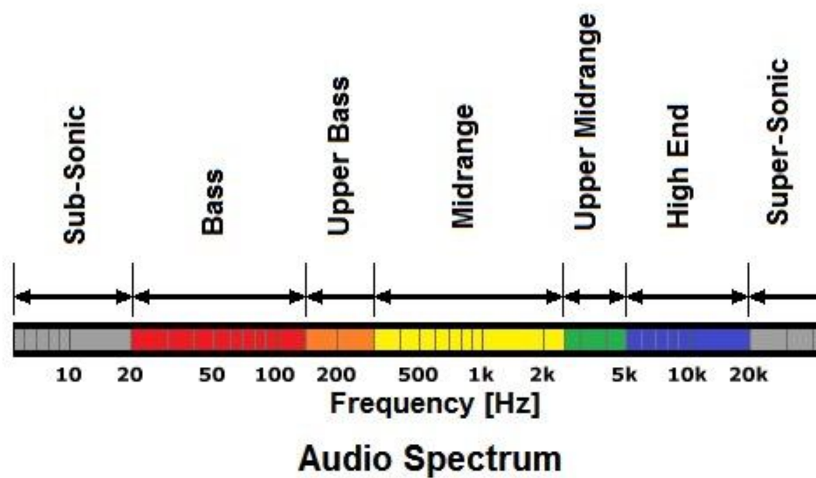


Figure 2.9: Audio spectrum division^{20}

Pitch: In music, pitch refers to the position of a single sound throughout the entire range of sound. The vibration's frequency of the sound waves that produce them determines how high or low the pitch of the sound is. A high frequency (e.g., 880 Hz) is seen as a high pitch, while a low frequency (e.g., 55 Hz) is regarded as a low pitch (see Figure 2.10[21]). [MLA+16]

[20] Audio Spectrum, https://www.audioreputation.com/audio-frequency-spectrum-explained/
[21] Pitch, https://la.mathworks.com/help/audio/ref/pitch.html

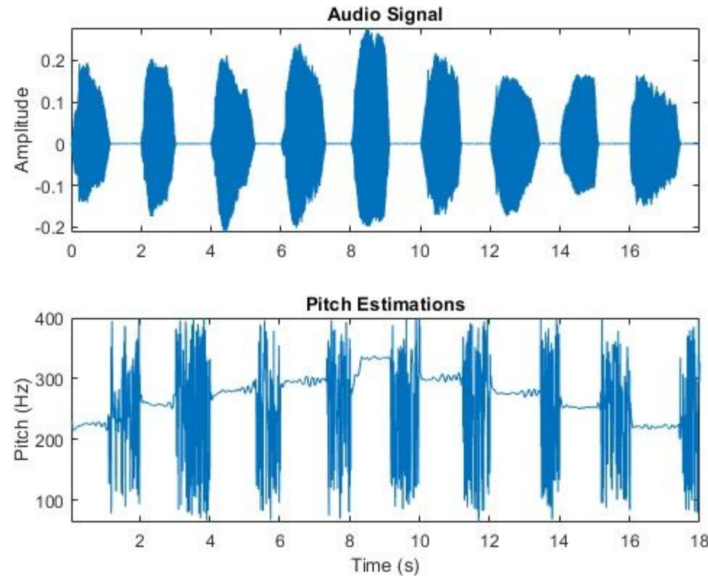


Figure 2.10: *Pitch*[21]

FFT magnitude spectrum: The basic measurement of the Fast Fourier Transform (FFT) is the spectrum. The magnitude is equal to the FFT's square root multiplied by its complex conjugate. That is, the square root of the sum of the squared real (sine) and imaginary (cosine) parts. The magnitude is a real number that indicates the entire amplitude of the signal in each frequency bin, regardless of phase. The real (cosine) or imaginary (sine) part of the phase may be displayed if there is phase information in the spectrum, i.e. the time record is triggered in phase with some component of the signal. The arctangent of the ratio between the imaginary and real components of each frequency component is the phase. The phase always refers to the beginning of the triggered time record[22] (see Figure 2.11[23]).

Zero crossings: An audio frame's Zero-Crossing Rate (ZCR) is the rate at which the signal's sign changes during the frame. In other words, it's the number of times the signal's value changes from positive to negative and back, divided by the frame's length. The ZCR is calculated using the following formula [GP14]:

$$\frac{1}{2W_L}\sum_{n=1}^{W_L}|sgn\left[x_i(n) - sgn[x_i(n-1)]\right|$$

[22] About FFT Spectrum Analyzers, https://www.thinksrs.com/downloads/pdfs/applicationnotes/AboutFFTs.pdf

[23] FFT magnitude spectrum, https://www.researchgate.net/figure/a-FFT-magnitude-spectrum-of-normal-displacement-at-300-mm-for-127-mm-thick-plate-with_fig1_321175008

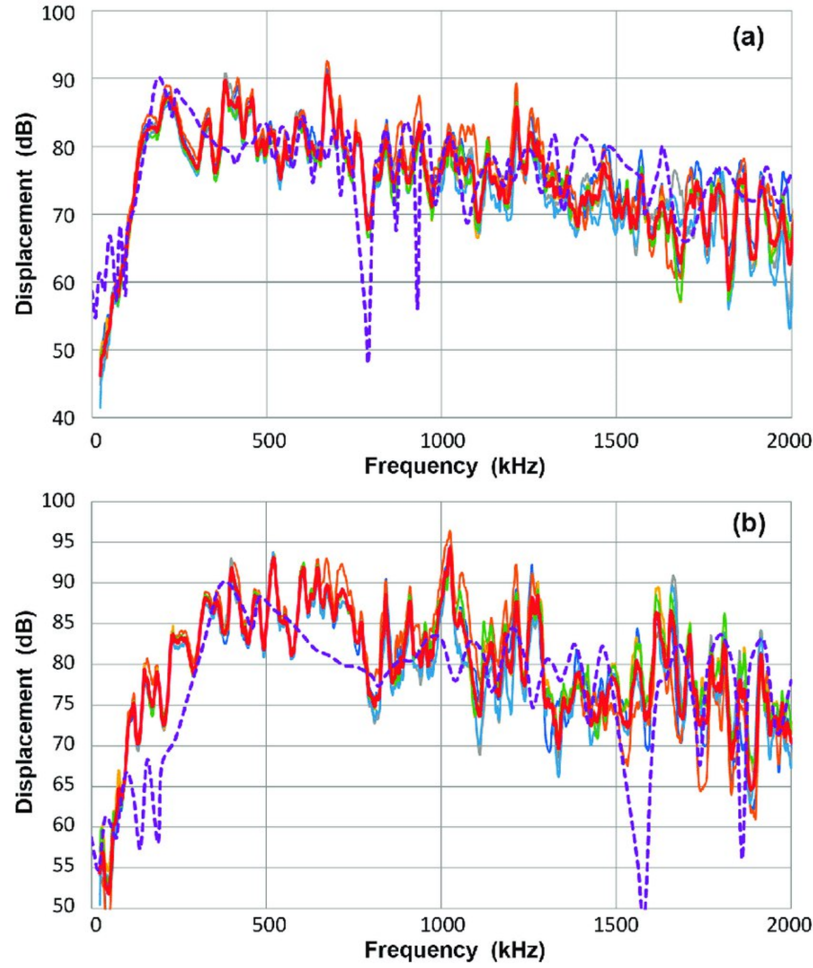


Figure 2.11: FFT magnitude spectrum[23]

where sgn(.) is the sign function, i.e

$$sgn\left[x_i(n))\right] = \begin{cases} 1, & x_i(n) \geq 0, \\ -1, & x_i(n) < 0 \end{cases}$$

Mel Frequency Cepstral Coefficients (MFCC): The Mel Frequency Cepstral Coefficients (MFCCs) can be seen as a compact description of the shape of the spectral envelope of an audio signal. The jth coefficient $v^j_{MFCC}(n)$ can be calculated with:

$$v^j_{MFCC}(n) = \sum_{k'=1}^{\kappa'} log(\left|X'(k',n)\right|) \cdot cos(j \cdot (k' - \frac{1}{2})\frac{\pi}{\kappa'})$$

with $|X'(k', n)|$ being the mel-wraped magnitude spectrum at the signal block, that can be found at [Ler12].

Spectral Rolloff: The spectral rolloff measures the bandwidth of the analyzed block n of audio samples. The spectral rolloff $v_{SR}(n)$ is defined as the frequency bin below which the STFT X(k,n) cumulative magnitudes reach a given percentage of the overall sum of magnitudes:

$$v_{SR}(n) = i \Big| \sum_{k=0}^{i} |X(k,n)| = \kappa \cdot \sum_{k=0}^{\frac{\kappa}{2}-1} |X(k,n)|$$

with common values for κ being 0.85 (85%) or 0.95 (95%). [Ler12]

Spectral Centroid: The spectral centroid represents the COG of spectral energy. [Ler12]. It is the frequency-weighted sum of the power spectrum adjusted by the unweighted sum:

$$v_{SR}(n) = \frac{\sum_{k=0}^{\frac{\kappa}{2}-1} k \cdot |X(k,n)|^2}{\sum_{k=0}^{\frac{\kappa}{2}-1} |X(k,n)|^2}$$

Normalization: Normalization[24] is a data-shifting and rescaling technique in which data points are shifted and rescaled until they're in the 0 to 1 range (see Figure 2.12[25]). It's sometimes referred to as min-max scaling. The following is the formula for calculating the normalized score:

$$Xnew = \frac{(X^{-}X_{min})}{(X_{max}{}^{-}X_{min})}$$

Here, X_{max} and X_{min} are the maximum and minimum values of the feature respectively.

2.8 Feature extraction libraries

Furthermore, we know that to implement the Audio Oracle and Variable Markov Oracle, we needed a C++ feature extraction library, and a C++ library for reading a writing audio files. The libraries shown afterwards are the ones we used in the implementation process of our project.

[24] Normalization definition. https://towardsdatascience.com/normalization-vs-standardization-explained-209e84d0f81e

[25] Normalization. https://kharshit.github.io/blog/2018/03/23/scaling-vs-normalization

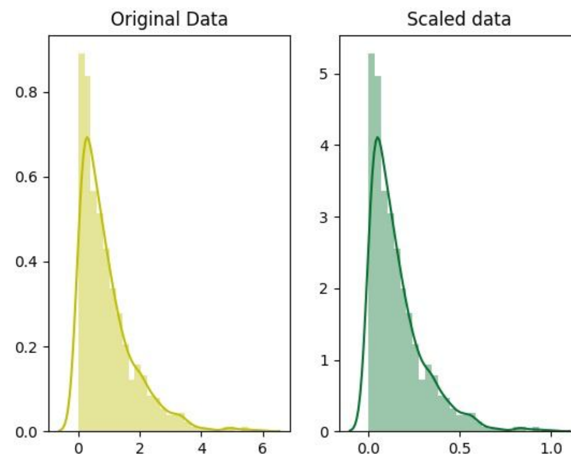


Figure 2.12: Normalization[25]

Marsyas: Marsyas [Leb17] which stands for "Music Analysis, Retrieval and Synthesis for Audio Signals" is an open source software framework for audio processing with specific emphasis on Music Information Retrieval applications. In addition, George Tzanetakis developed and wrote this library with the support of researchers and students from all around the world. Marsyas has been utilized in a range of academic and industrial initiatives. The last time the library was updated was in 2015.

Gist: Gist [Sta17b] is an audio analysis library developed in C++. Its main purpose is to process audio frames and extract their audio features. We show a similar process in Figure 2.13[26].

This library was created in 2014, and it was last updated in 2020. The six audio feature groups that can be extracted are:

- Core Time Domain Features
- Core Frequency Domain Features
- Onset Detection Functions
- FFT Magnitude Spectrum
- Pitch
- Mel-frequency Representations

Moreover, Gist can depend on one of the following FFT libraries: FFTW, Kiss FFT, and Apple Accelerate FFT.

[26] Audio features processing, https://www.sciencedirect.com/topics/engineering/audio-feature

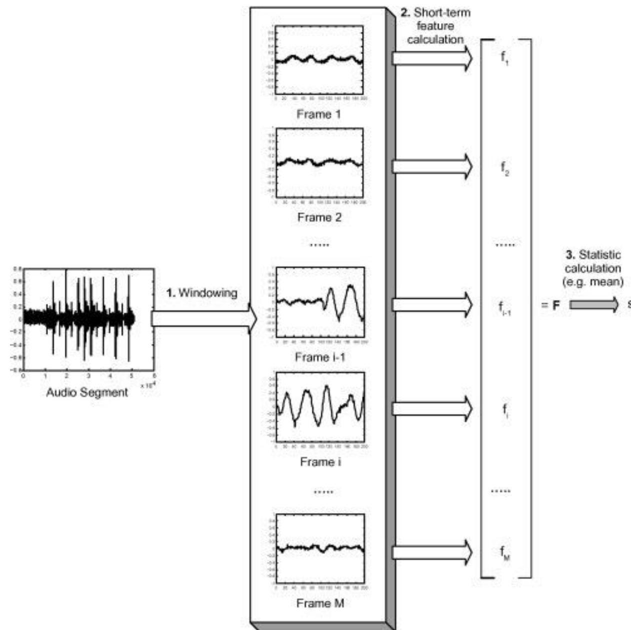


Figure 2.13: Audio features processing[26]

AudioFile: AudioFile is a C++ library for reading and writing audio files, it is a header-only library [Sta17a].

Chapter 3

Factor Oracle

The Factor Oracle (FO) algorithm started as a way to compute very long strings, such as the ones generated in computational biology. Basically, one of the most important applications for this algorithm was to improve experiments on genomic sequences and in data compression. At first, there were different versions of the FO, nonetheless, we used the improved algorithm shown in [LLA03], which in it was shown in different papers, that it gives better results than the previous versions.

On the other hand, the FO could also be used for musical improvisation. Only based on the FO algorithm, we could develop a machine that "listens" to three synchronized sources: a metric source that provides a stream of pulses, a harmonic source that sends harmonic labels, and a melodic source that delivers a stream of time-tagged note events. These processes could collect information by analyzing and listening to the signals generated by performers. When triggering a signal, the FO could process this information, create the automaton and finally create an improvisation [AD04].

The implementation of the algorithms is done in the programming language C++. The main reason of choosing C++ over different programming languages, is that these algorithms will be plug-ins inside an application called `ossia-score`. `ossia-score` was developed in C++ and provides an C++ API to develop its plugins, thus the use of the same programming language is the most convenient solution to implement new plug-ins and features.

Moreover, in this chapter we show the process of the Factor Oracle development, this development starts by defining the Factor Oracle. Afterwards, we start the C++ implementation, then the FO integration with `ossia-score`. Ultimately, we create the automatons representing the Factor Oracle.

We shall start by defining the Factor Oracle, which is shown in the next section 3.1.

3.1 Definition

Factor Oracle (FO) [LLA03] is a data structure that allows to store the suffixes efficiently. It is an on-line linear heuristic method which finds the repeated substrings in a string. This method uses the structure of a finite automaton. The FO of a word p of length m is a deterministic automaton (Q, q_0, F, δ), where $Q = 0, 1, \ldots, m$ is the set of states, $q_0 = 0$ is the initial state, $F = Q$ is the set of final states and δ is the transition function. FO is equipped with the following main functions:

- $\mathsf{lrs}(p)$ is the longest repeated suffix of p : $\mathsf{lrs}(p) = v$ such that v is a suffix of p and v is a factor

of $p[1..|p|-1]$ and $|v|$ is maximal.

- $\mathsf{S}(i)$ is the suffix link of a state i of the factor oracle of a word p. It is equal to the state in which the longest repeated suffix of $p[1 \ldots i]$ is recognized: $\mathsf{S}(i) = \delta(0, \mathsf{lrs}(p[1 \ldots i]))$.

In Figure 3.1, we present the FO generated using the string `abbcabcdabc` as input. Note that several sequences of letters are repeated, such as `a`, `b`, `c`, `ab`, `bc`, `abc`. The FO has different elements that are described below. *Backward arrows* (colored in light blue) are the suffix links. Every state except state 0 has a suffix link. *Forward arrows* (colored in black) are the forward transitions. The lrs of each state is shown next to each state (numbers colored in blue). Finally, we also have forward transitions that show us the input of the automaton.

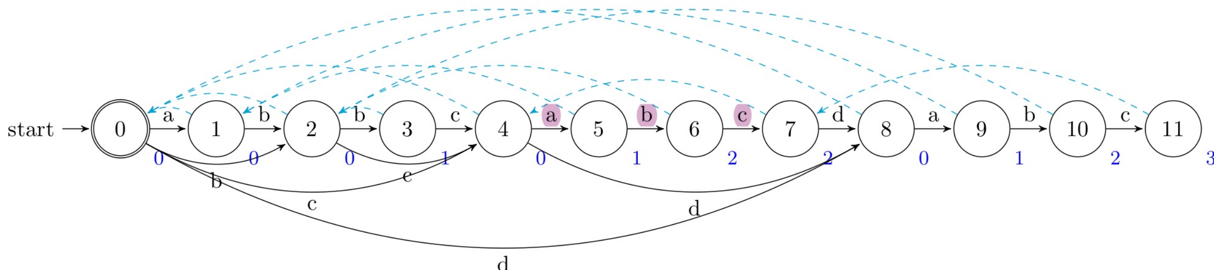


Figure 3.1: Factor Oracle automaton where the longest repeated subsequence is `abc`

The longest repeated subsequence (lrs) is the longest of all the subsequences in the input sequence. In our case, the lrs of the string `abbcabcdabc` is `abc`. The suffix links that can be seen in Figure 3.1, are every blue backward transition. A suffix can be understood as the state that has the longest subsequence from state i to state j, for instance, state 11 has a suffix transition to state 7. That is, if we go back the number of states the lrs of state 11 says (i.e. 3) from state 7 and state 11, we will find the subsequence `abc` which is the longest in the sequence. Another example is, the state 5 has a suffix transition to state 1, if we go back one state, which is the lrs it has (the dark blue number on its right), the letter `a` is the only sequence equal in both states.

We also see forward transitions, which are divided into two types: *internal transitions* and *external transitions*. The former is a transition that goes from a state i to a state $i+1$. The latter, is a transition from state i to state j such that $j - i > 1$. An *external transition* is defined by the $\mathsf{AddLetter}$ function. Two examples of these transitions are the ones in Figure 3.1; one from state 4 to state 8 with the letter d, and another from state 0 to state 8 with the letter d.

3.2 Construction

The algorithm to construct a FO is formed by four main functions, which we will describe in this section. These functions are: Oracle(p, m), AddLetter(i), FindBetter(i, a) and LengthCommonSuffix(Π_1, Π_2).

The principal advantage of the FO is that it is a polynomial algorithm. The Oracle function (see Algorithm 1) has a loop that goes from 1 to the number m, which is the length of the string. That means it does not have to run multiple times over the string to find its repeated subsequence, it is only one loop and that simplifies its complexity to $O(n)$.

Oracle receives a string p and the number of states m, the number of states is defined by the length of the string. For each state there is, Oracle calls another function called AddLetter.

Algorithm 1 Oracle(p, m) function: builds, on-line, the factor oracle of the word p of length m

Require: Word p and length m

```
1: Create state 0
2: S(m) ← -1
3: for i ← 1 to m do
4:     do AddLetter(i)
```

AddLetter (see Algorithm 2) receives an integer number i which is, the actual state of the loop. This function creates a new state i, defines a new transition from the state $i-1$ to the state i with the letter in the position i of the string in question [LLA03]. Moreover, T is a data structure that stores the suffix of each state. More precisely, $T[i]$ is defined as:

$$T[i] = \left\{ j \mid S'[j] = i \wedge i < j \leq m \right\} \text{ for } 0 \leq i \leq m-1$$

For the algorithm to work, there is a variable k which stores mainly the values of the suffix transition of a desired state and another variable Π_1 which stores a desired state. When the value of k is defined, there are different options the algorithm can choose. The first one is when k is bigger than -1 and there is no transition from k with the letter $p[i]$. In this case, it creates a new transition from the state k to the state i with the letter $p[i]$, it also defines Π_1 as k, and k becomes the suffix transition of the state k. This happens until one of the two main conditions becomes false. If the algorithm finds it necessary, it will call the functions LengthCommonSuffix and FindBetter.

Algorithm 3 FindBetter function

```
1: for all the elements j of T[i] in increasing order do
2:     if lrs[j] = lrs[i] and p[j - lrs[i]] = a then
3:         return j
4: return 0
```

FindBetter (see Algorithm 3) is a really important function because by using a stored vector of suffix transitions (which updates each cycle of the loop), it finds a better suffix transition of a state. For example, there could be times when there is a string `abbcabcdabc`, in this case, the suffix transition of the last state should be `abc` and its lrs should be 3, but if the algorithm does not have this function the suffix transition of the

Algorithm 2 AddLetter(i) function

1: Create a new state i
2: Create a new transition from m to $m+1$, $\delta(i-1, p[i]) \leftarrow i$
3: $\Pi_1 \leftarrow i$ - 1
4: $k \leftarrow S[i$ - 1]
5: **while** $k >$ - 1 and $\delta(k, p[i])$ is undefined **do**
6: $\quad \delta(k, p[i]) \leftarrow i$
7: $\quad \Pi_1 \leftarrow k$
8: $\quad k \leftarrow S[k]$
9: **if** $k =$ -1 **then** ▷ No suffix exists
10: $\quad S[i] \leftarrow 0$
11: $\quad lrs[i] \leftarrow 0$
12: **else**
13: $\quad S[i] \leftarrow \delta(k, p[i])$
14: $\quad lrs[i] \leftarrow$ LengthCommonSuffix$(\Pi_1, S[i]-1)+1$
15: $k \leftarrow$ FindBetter$(i, p[i-lrs[i]])$
16: **if** $k \neq 0$ **then**
17: $\quad$ **then** $lrs[i] \leftarrow lrs[i]+1$
18: $\quad S$[i] $\leftarrow k$
$T[S[i]] \leftarrow T[S[i]] \cup \{i\}$

last state would be bc and its lrs would be 2. The above is not correct and the FO algorithm would not achieve its goal.

This example is presented in the automaton shown in Figure 3.2. Here, we can see that the longest repeated subsequence is 2. The reason for this is that the suffix transition of the state 11 is pointed to the state 4, and as the *lrs* is 2, it is telling us that the longest repeated subsequence is bc.

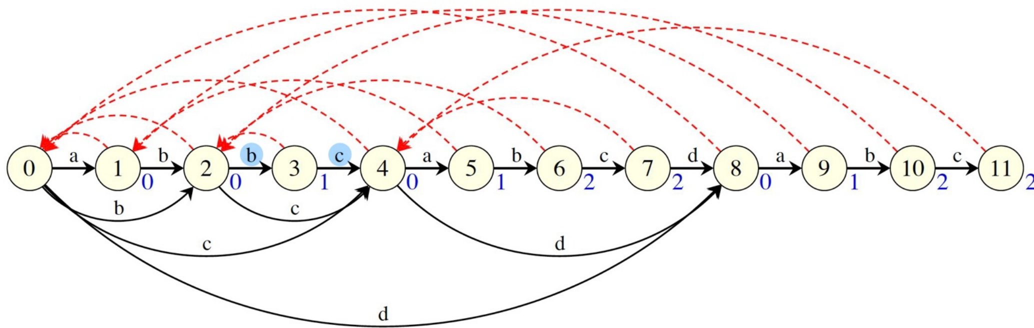


Figure 3.2: Factor Oracle automaton with wrong longest repeated subsequence length [LLA03]

Nonetheless, we know this result is not right. The longest repeated subsequence of this sequence is `abc`, and it is repeated twice. Knowing this, the automaton in Figure 3.3 shows the right result. The first two differences we see are:

- The suffix transition of state 11 is pointed to state 7 instead of 4.
- The state 11 shows that the longest repeated subsequence length is 3 instead of 2.

These two differences are really important. We can see that, indeed, the suffix transition is pointing to the true longest repeated subsequence: `abc`.

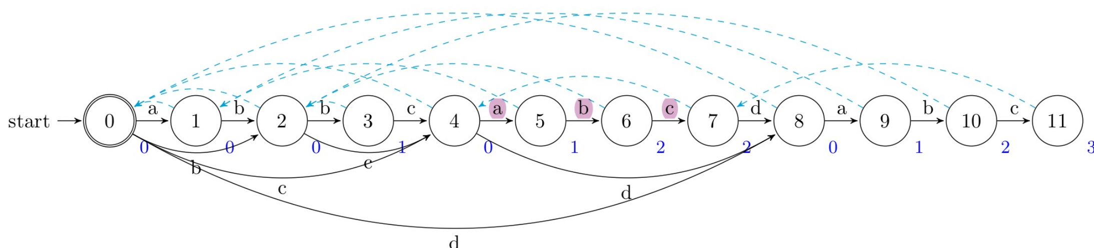


Figure 3.3: Factor Oracle automaton with the longest repeated subsequence length equals to 3

Afterwards, LengthCommonSuffix (see Algorithm 4) checks if the *Longest Common Suffix* of a state which was defined before is indeed the longest in the whole sequence. Corroborating that the FO is working as it should be and giving the best automaton.

Algorithm 4 LengthCommonSuffix(Π_1, Π_2) function

1: **if** $\Pi_2 = S[\Pi_1]$ **then**
2: **return** $lrs[\Pi_1]$
3: **else**
4: **while** $S[\Pi_2] \neq S[\Pi_1]$ **do**
5: **do** $\Pi_2 \leftarrow \Pi_2$
return min($lrs[\Pi_1]$,$lrs[\Pi_2]$)

To finish the process, after the FO automaton is created, the improvisation can start. The improvisation algorithm presented in Algorithm 5 generates a sequence of symbols v depending on a probability q, $0 \leq q \leq 1$ [AD04].

Furthermore, in the next section we will introduce the implementation process of the Factor Oracle, the obstacles we encountered and the solutions we developed.

3.3 Implementation Process

The FO implementation process was divided into two different processes. The first one was creating the C++ code on its own, testing it and creating the visual representation of the automaton (image as a PDF file) when the automaton was created. The second one was modifying the C++ implementation to be able to run FO as a plug-in on `ossia-score`, which included a lot of changes to

Algorithm 5 Improvisation function

Require: Oracle Σ, probability q, improvisation vector v
1: Generate random number p
2: **if** Σ length $= 2$ **then**
3: $v \leftarrow vp[0]$
4: **else**
5: **if** $p < q$ **then**
6: $i \leftarrow i + 1$
7: $v \leftarrow vp[i]$
8: **else**
9: Choose a random symbol p in $\{\ p[j] \in \Sigma \mid \delta(S[i],\ p[j]) \neq \bot\ \}$
10: $i \leftarrow \delta(S[i],\ p)$
11: $v \leftarrow vp$
12: **return** $\boldsymbol{v}$

be able to test the plug-in inside the `ossia-score` application. The `ossia-score` implementation code can be found in the ossia repository in GitHub (https://github.com/ossia/score).

The second process, which is the modification of the FO code to work in `ossia-score`, depends incredibly on the version of `ossia-score`. This application has been in constant updates through the months. This is why we created a first version of the improved FO and FO MIDI in an older, but stable version of `ossia-score`.

As `ossia-score` has been updated within time, the first implementation of the FO plug-in was in an older version of `ossia-score`, to find these plug-ins the user needed to go to the `Processes` tab and then `Impro`, as shown in Figure 3.4. Furthermore, there was a FO plug-in developed in `ossia-score` before the new improved algorithm we were implementing, this is why, for the first version, the names of the FO plug-ins were `Factor 2` and `Factor 2 MIDI`, these names were changed afterwards.

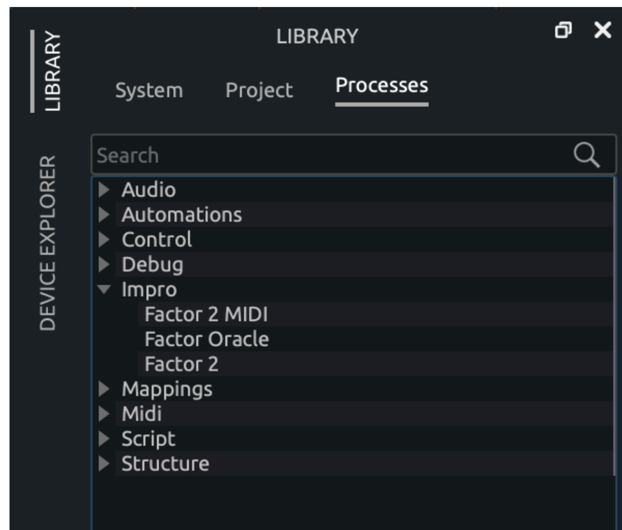


Figure 3.4: Factor Oracle plug-ins in an older version of `ossia-score`

Besides the difference of the `Processes` tab, the flow of the FO plug-in was completely different from the updated version. The flow that was implemented before is shown in Figure 3.5, in this Figure we have numbers that describe the different sections of `ossia-score` elements which create the Factor Oracle flow. These numbers

will be defined as (#.#.#), which are referenced in the next paragraph.

The flow that is shown here starts with the inputs (4.2.1), each one of the visible dots is an input, in this case the inputs can be any symbol from the alphabet or any word. Afterwards, a bang input (4.2.2) is triggered and the generation of the improvisation sequence can start. Finally, there is a loop of bangs (4.2.3 and 4.2.4) that creates new improvisations every time it is triggered. Another thing that is shown, is the box in the bottom of the image, this is the visual representation of the FO plug-in.

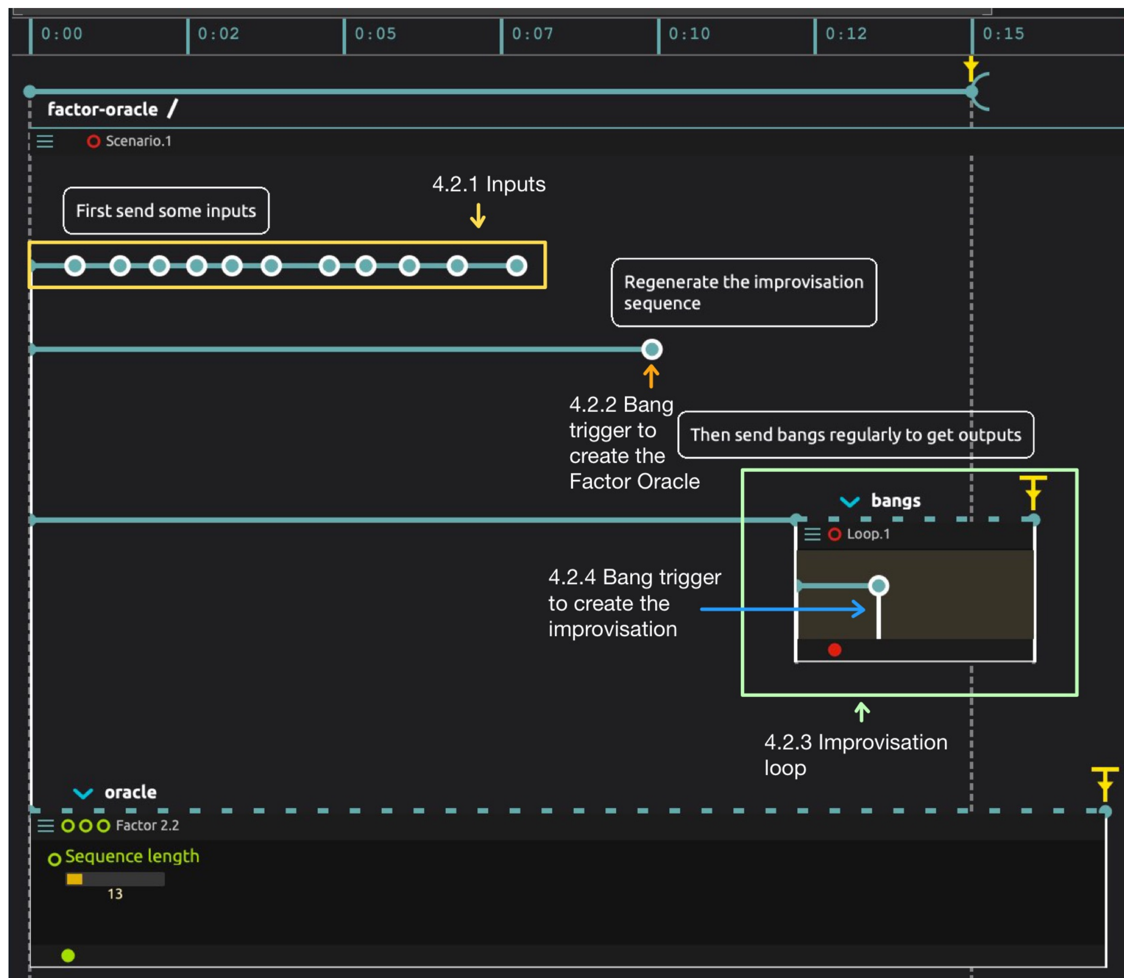


Figure 3.5: Factor Oracle flow in an older version of `ossia-score`

3.3.1 Current Version of ossia-score

3.3.1.1 Factor Oracle in ossia-score

After the updates of the ossia-score application, some changes needed to be made to the algorithm and the file that had the FO flow shown before. The different versions made the files to not be compatible with the new version of ossia-score, which delayed the deployment process approximately two months and some changes were made inside the flow file. Additionally, the final changes of the Factor Oracle are here (https://bit.ly/3iCNQg6), we created a pull request to integrate it with ossia-score.

The main difference of the new version of ossia-score is the organization of the sections inside the application, the creation of loops and other processes. For the organization of the Processes tab, it is located on the left side of the screen, the Processes now have icons to recognize them more easily and the new name of the Factor Oracle and Factor Oracle MIDI are New Factor Oracle and New Factor Oracle MIDI, as seen in Figure 3.6.

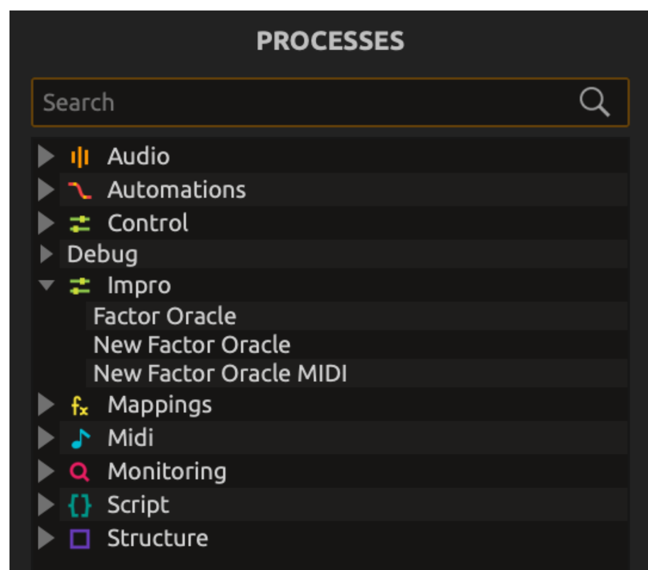


Figure 3.6: ossia-score processes tab

For the FO with symbols, the input process and the trigger to finish the automaton creation are really similar to the way it was performed before. Nonetheless, the loop to keep creating the improvisation is now completely different. For instance, this new flow is shown in Figure 3.7, the loop (4.4.3) is the fourth trigger inside the flow, every time it touches the bang trigger (4.4.4) it creates a new improvisation symbol.

On the left side panel of application, while on the last section of the tabs, when the flow starts

running, it shows the inputs and outputs that are being triggered during the flow run. This can be seen in Figure 3.8 and Figure 3.9. In the former, we can see green horizontal lines and a thin vertical line. The vertical line shows us where the flow is at that exact time. We can see that at that moment, it is in the middle of the flow. Meanwhile, in the latter, the green line is at the end, showing us only a green line in the loop. This means that the loop is still running and is creating new improvisations.

In Figure 3.9, we can see on the left side of the application that it says `[push]/output string: "<string>"`. These are the improvisations that are being created every time the bang is triggered. This is why it also says `[push]/bang "impulse"` right before a new output, this is the improvisation trigger. Moreover, what we can see is that if the flow is not stopped, the improvisations will be created forever, accordingly, the user needs to stop the flow when the improvisations they want are completed. In this example, some improvisations are 2, 3, 1, 2, 2, 1, 2, 3, 2, 3, 1, 2, 2, 1, 2.

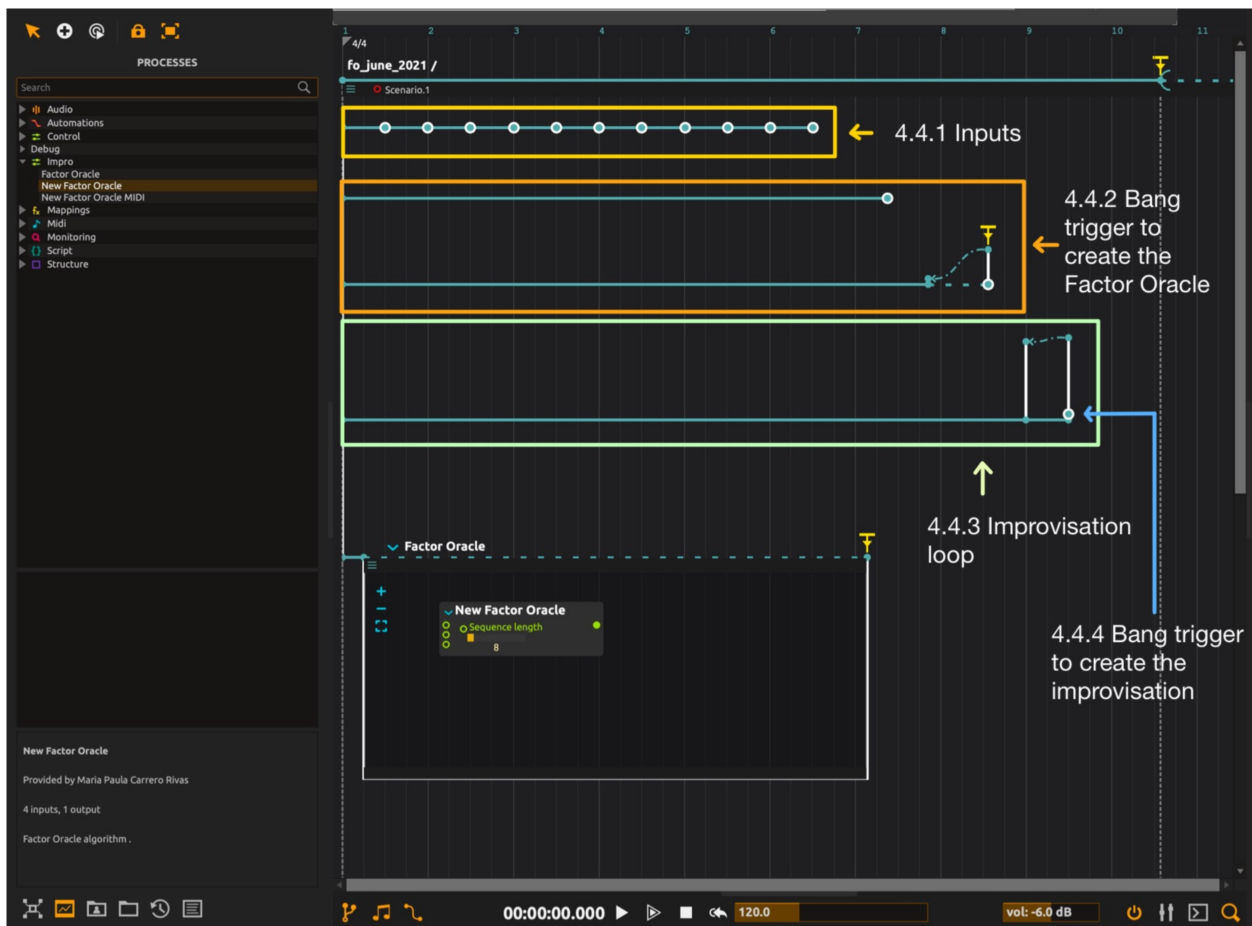


Figure 3.7: Factor Oracle flow (https://bit.ly/2VWSaOY) in ossia-score

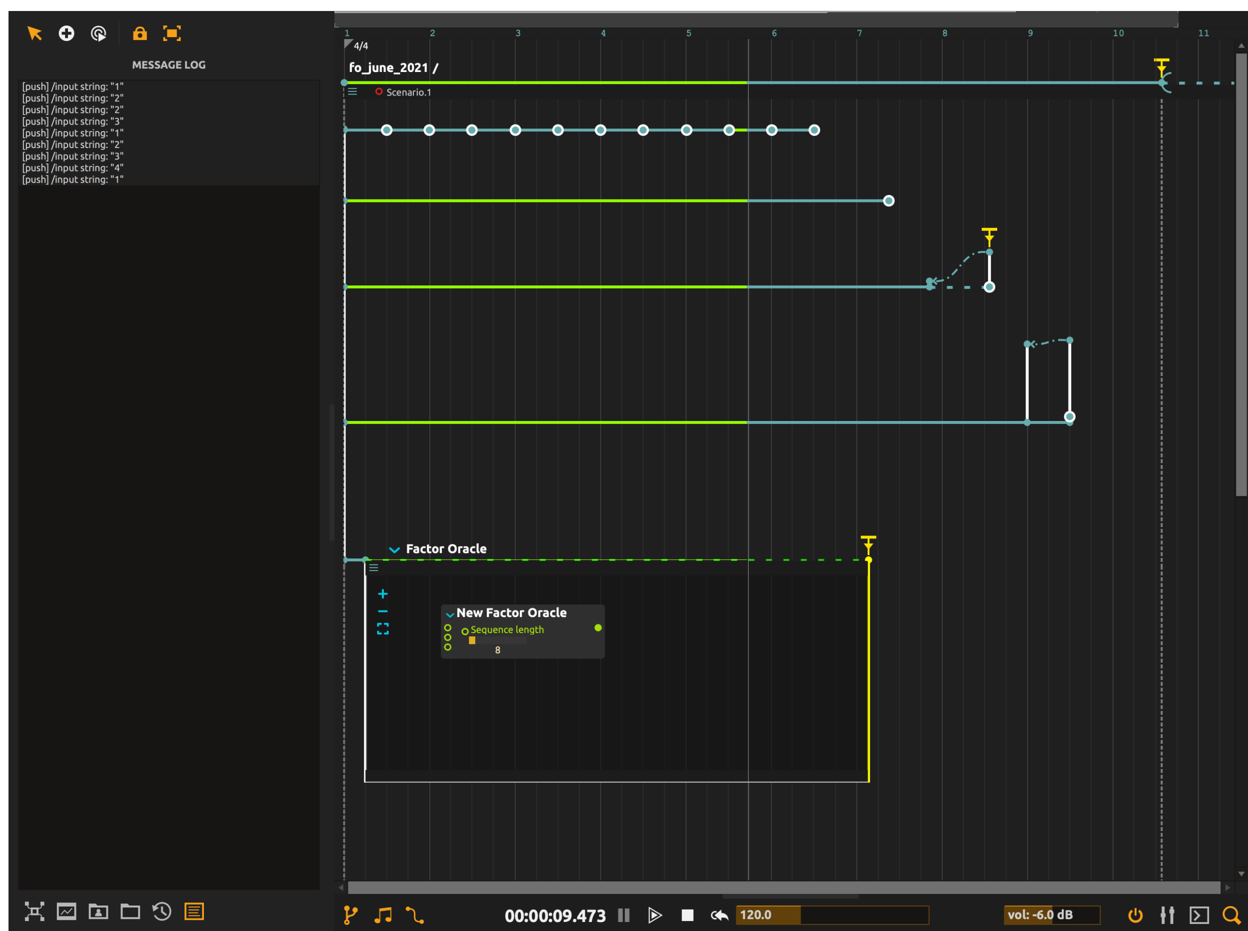


Figure 3.8: Factor Oracle flow (`https://bit.ly/2VWSa0Y`) in `ossia-score` with the improvisation

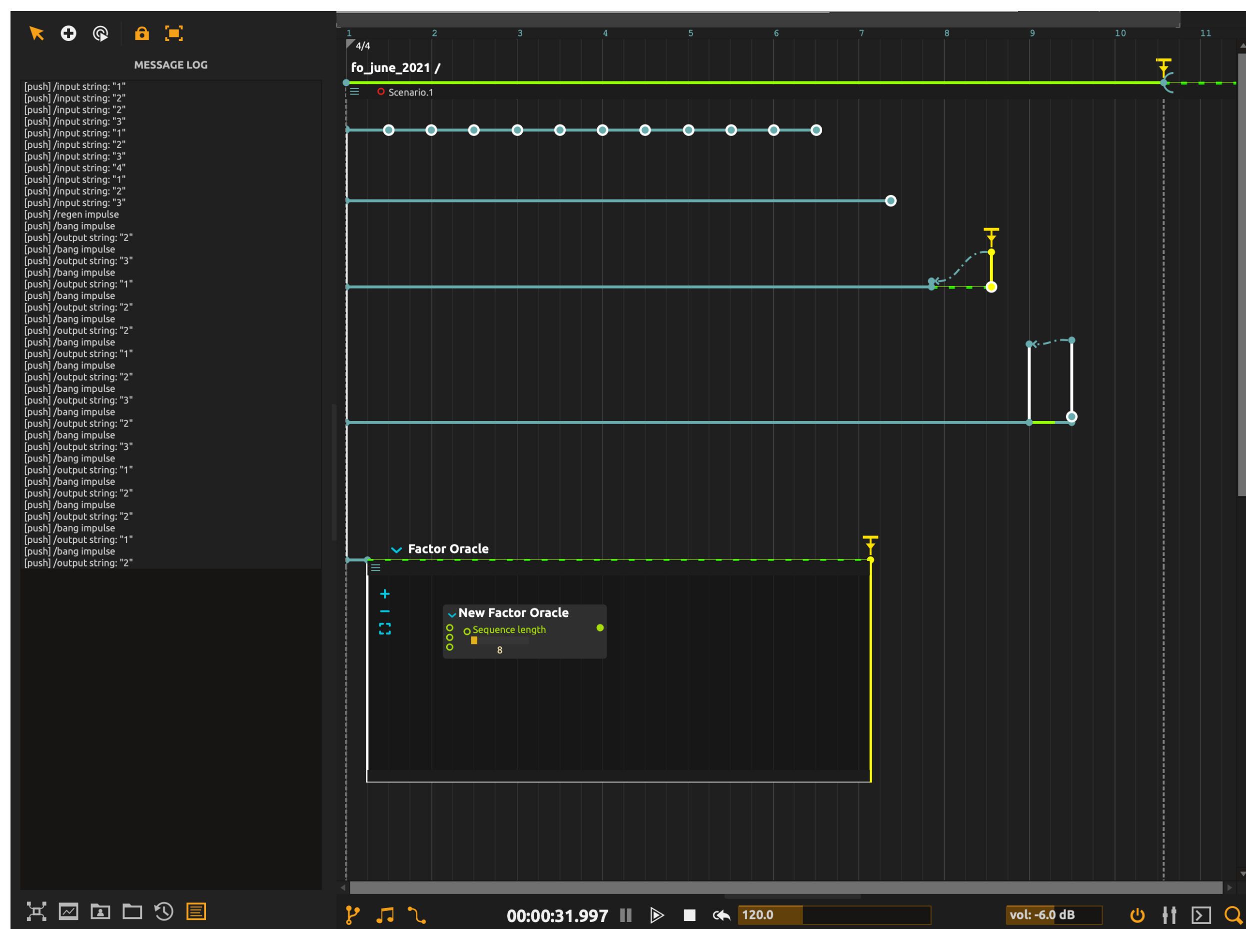


Figure 3.9: Factor Oracle flow (`https://bit.ly/2VWSaOY`) in `ossia-score` running the loop to create improvisations

3.3.1.2 Factor Oracle MIDI in `ossia-score`

After finishing the Factor Oracle flow in `ossia-score`, we started creating the Factor Oracle MIDI flow, which is slightly different from the normal Factor Oracle because as its name says, the input and the output are MIDI. The last changes done to the Factor Oracle MIDI are here (`https://bit.ly/3iHzbQN`), the pull request with `ossia-score` is being processed. For this implementation, even though they are both Factor Oracles, the MIDI needed to have some changes in the code, because the input was not the same. In the Factor Oracle the input can be symbols, strings or words, while, in the Factor Oracle MIDI, the input is MIDI notes.

In this case, the input we used for the flow was a Piano Roll, it can be found in the `Processes` tab, then `Midi` and finally `Piano Roll`. The Piano Roll can be seen in Figure 3.10 at the bottom (4.10.1),

there is the Regen (4.10.2) which triggers the Factor Oracle MIDI creation, the improvisation loop (4.10.3), every time it touches the bang trigger (4.10.4) it creates a new improvisation, for the MIDI, the improvisation is translated as a note which is generated with the MIDI Pitch (4.10.5) and afterwards, passed through the Audio Generator (4.10.6) to generate the note.

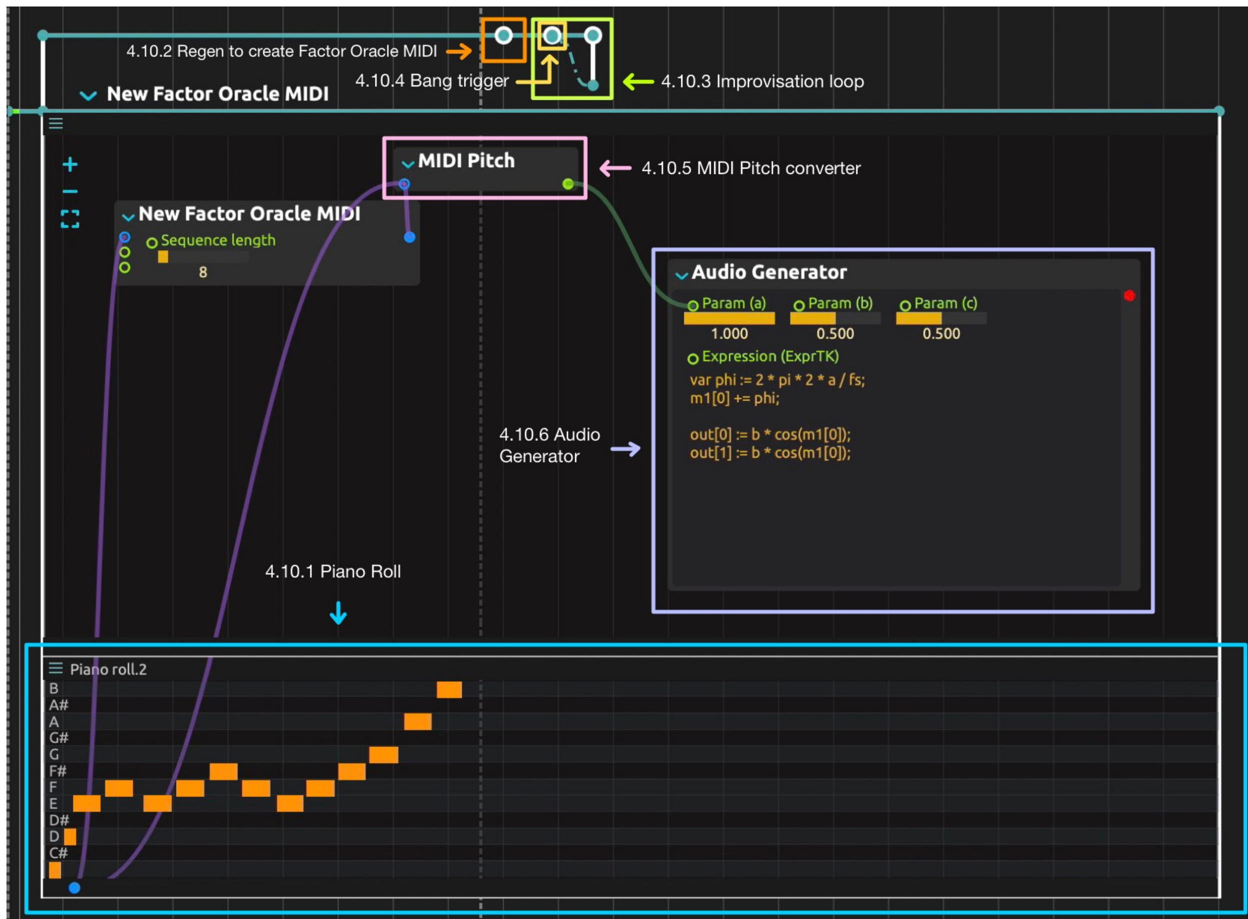


Figure 3.10: Factor Oracle MIDI (`https://bit.ly/3yO4cbm`) flow in `ossia-score`

In Figure 3.11, the flow is being run and the piano's keys input is being received. If we take a closer look at the piano roll, we can see that we have the longest subsequence from the fourth key until the seventh key and then from the eight key until the ninth key, the longest repeated subsequence is `F`, `E`, `F`, `F#`. Which can be seen in the automaton in Figure 3.12.

Figure 3.11: Factor Oracle MIDI (https://bit.ly/3y04cbm) flow progress in ossia-score

Figure 3.12: Factor Oracle MIDI automaton of Figure 3.10

Meanwhile, in Figure 3.13, the improvisation loop is creating the MIDI improvisations, every time the bang is triggered, a new note is played. As it is difficult to show the MIDI improvisation, a video of the whole flow can be seen here (`https://bit.ly/3y04cbm`).

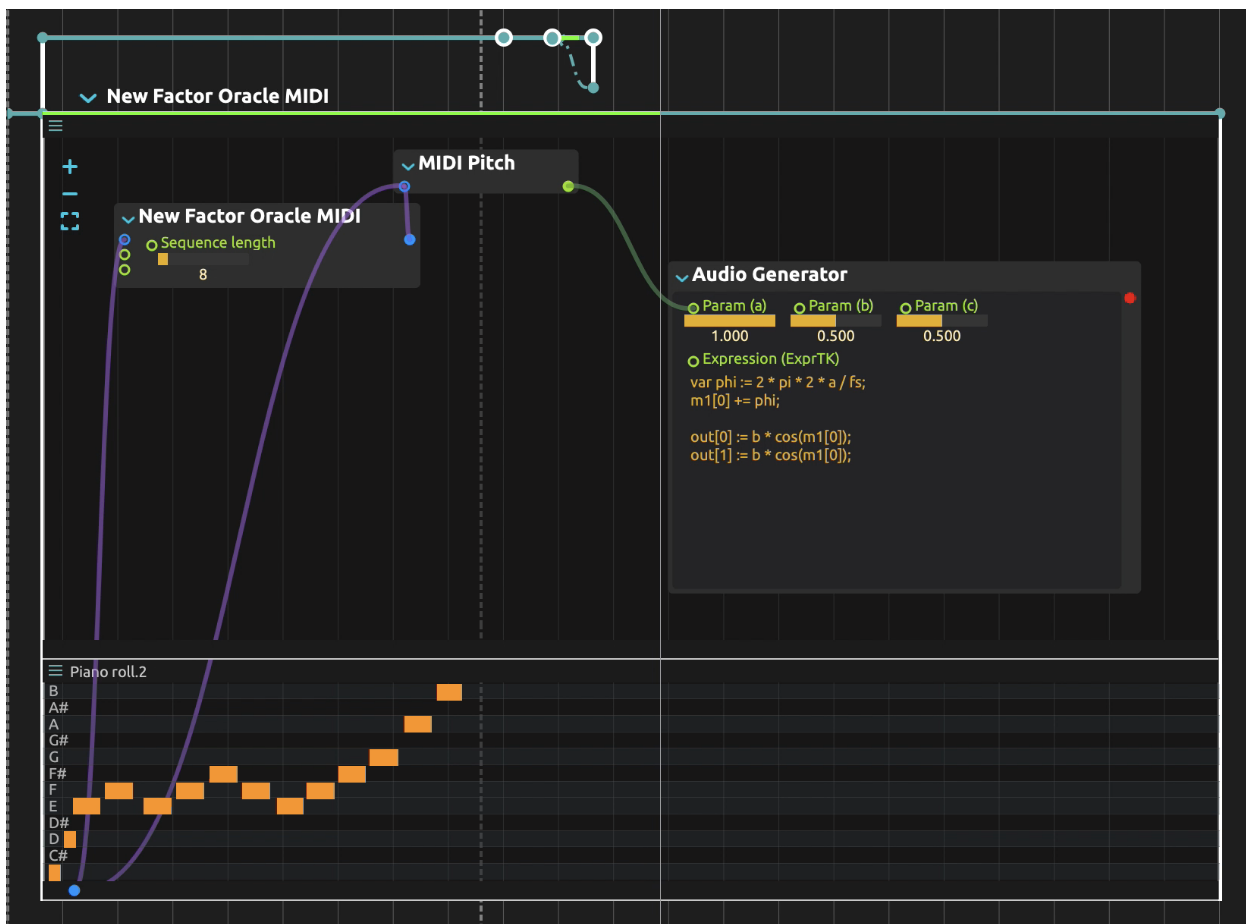


Figure 3.13: Factor Oracle MIDI (`https://bit.ly/3y04cbm`) flow loop in `ossia-score`

After finishing the implementation, we wanted to point out the difference of the automatons depending on the feature extracted and different factors. This is why the automatons will be shown in the next Section 3.4.

3.4 Automaton Creation

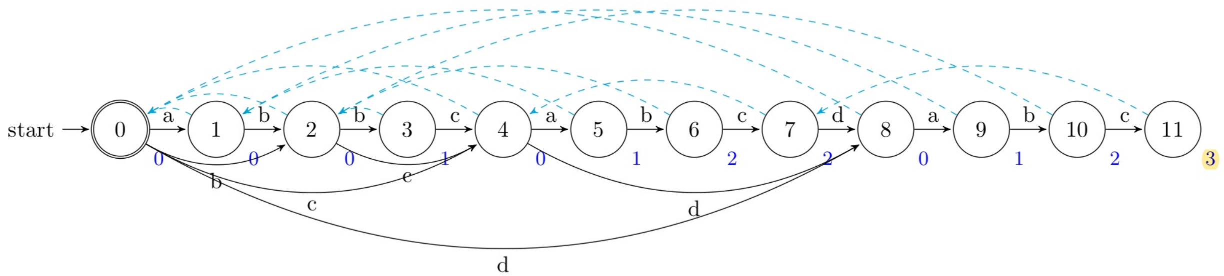
With different inputs, the automaton created using the Factor Oracle algorithm will be completely different. With the examples presented in this subsection, we can see that this algorithm finds efficiently the longest repeated subsequence, and the importance of using this algorithm over others.

In addition, the automaton figures that will be shown below were generated with the implemented tool.

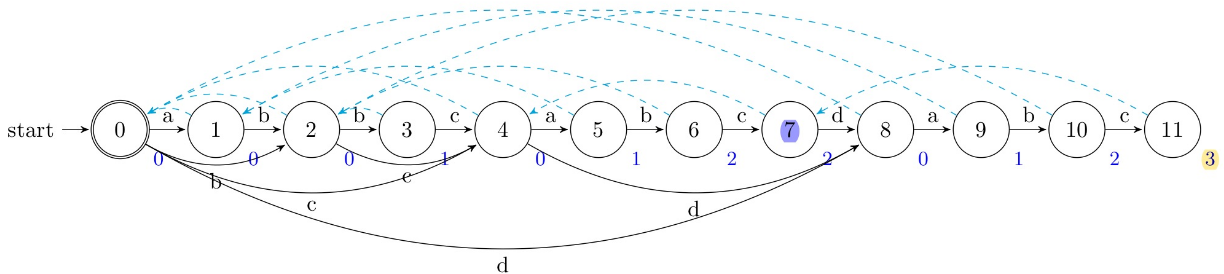
Understanding how the FO automaton works is incredibly important before trying to analyze it. What we are looking for when creating the automaton is finding the longest repeated subsequence or subsequences. The input of the Factor Oracle is any sequence of symbols, strings, words, or MIDI input. The latter is analyzed as a sequence of strings.

The steps to find those subsequences are:

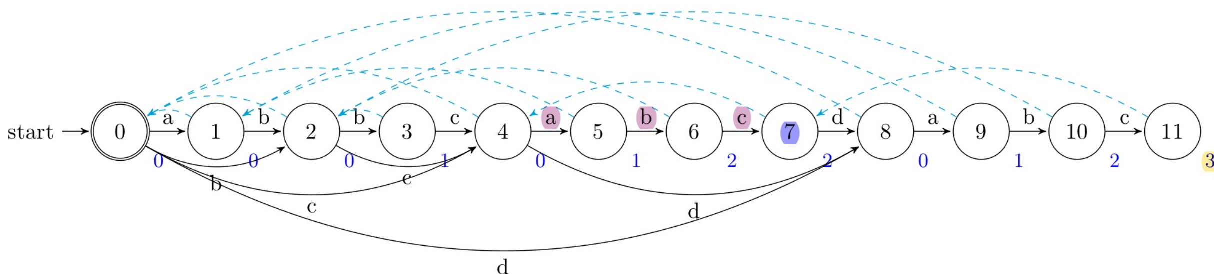
1. Find the biggest number beside the states. These numbers are the length of the repeated subsequences. In this example, the number three beside state 11 is highlighted yellow, this is the biggest number.



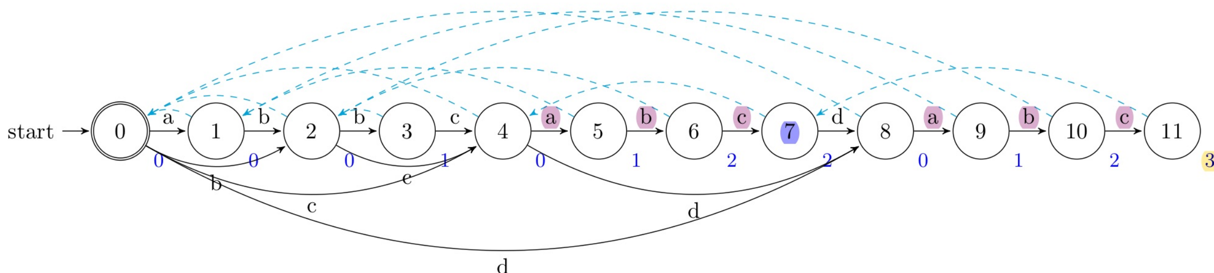
2. Go to the suffix transition from the state with the biggest number to the state that points the suffix. Now, state 7 is highlighted because state 11 has the longest repeated subsequence length, and the suffix of state 11 points to state 7.



3. From the state that pointed the suffix transition, move backwards the number of states the state with the longest repeated subsequence number has, move forward the same number taking into account the symbols the states have. Thus, as we were in state 7, the highlighted symbols are `abc` since we go back 3 states from state 7 and the lrs is that sequence.



4. Finally, as shown in the previous step, we highlight the lrs which is `abc`.



After explaining the automaton creation and identification of the longest repeated subsequence, we will use some examples to explore some scenarios that can be created using the FO. Consequently, demonstrating the importance of the FO algorithm and the improvement that was implemented.

Factor Oracle with no repeated symbols For example, the automaton shown in Figure 3.14 does not have any repeated symbol. In this case the longest repeated subsequence must be zero because of this exact reason. Moreover, the suffix of every state will be the state zero, and zero will have forward transitions to every state.

Factor Oracle with one repeated symbol Comparing the automaton in Figure 3.14 to the one in Figure 3.15, in the former there were no repeated symbols while in the latter we have b as a repeated symbol. Thus, the transition from state 1 and state 2 has b as a symbol, and the transition from state 9 to state 10 has b as a symbol. This means that the state 10 indeed must have the longest repeated subsequence length equal to 1.

Until now, we may not have seen the real importance of the improved FO because we have only seen two examples: one without repeated sequences and another one with one repeated subsequence of length one. But what would happen if there is one repeated subsequence such as `b`, `c`, and another repeated subsequence `a`, `b`, `c`. Usually, if the algorithm is not well written, it can get confused and

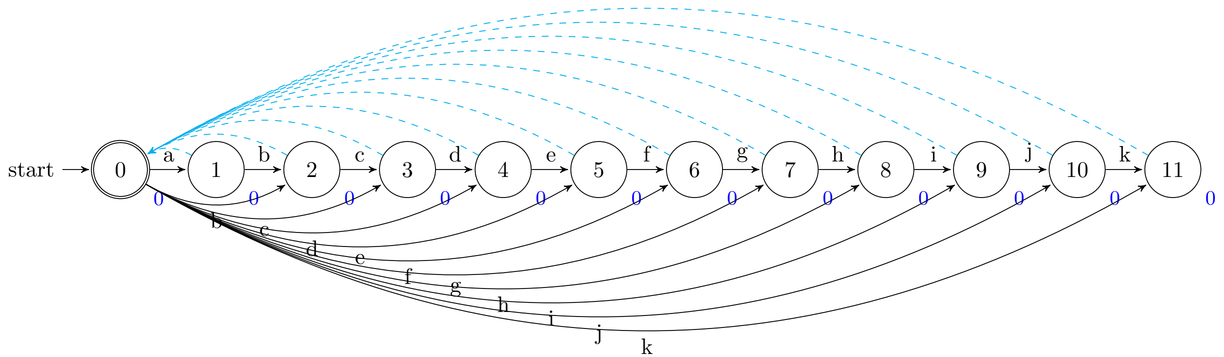


Figure 3.14: Factor Oracle automaton with the longest repeated subsequence length equals to 0

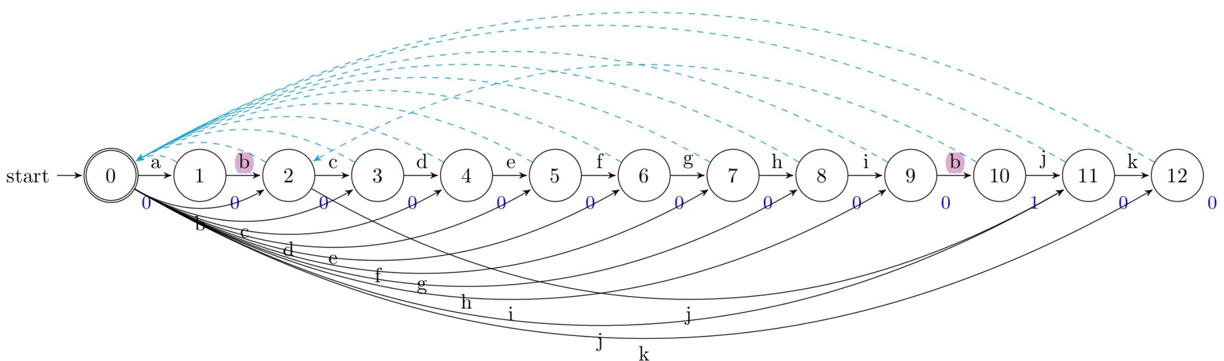


Figure 3.15: Factor Oracle automaton with the longest repeated subsequence length equals to 1

think that the longest repeated subsequence is `b`, `c`, because `b`, `c` is a subsequence of `a`, `b`, `c`. This example is shown in Figure 3.3 and in the explanation process of finding the *lrs*.

Factor Oracle with 22 states As for this example, we are exploring a larger automaton, the automaton in Figure 3.16 has 22 states. The longest repeated subsequence in this automaton is 'a', 'b', 'b', 'c', 'a', 'b', 'b', 'c', 'd' and its length is exactly nine. This *lrs* is interesting because it has repeated sub-subsequences, a sub-subsequence is a subsequence of the longest repeated subsequence. In this automaton, a sub-subsequence could be `a`, `b`, `b`, `c`, which can be seen from state 0 to state 4, from state 4 to state 8, from state 12 to state 16 and from state 16 to state 20, this means it has

four occurrences in the automaton, and two in the longest repeated subsequence.

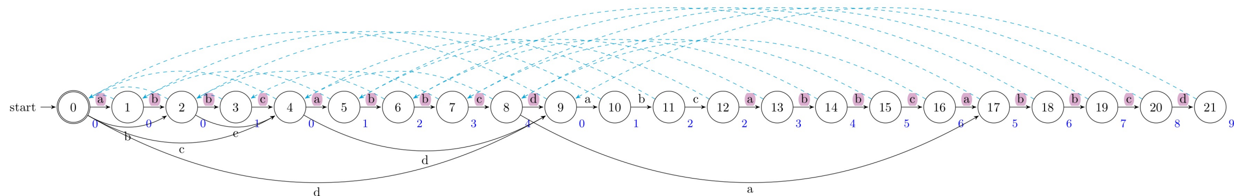


Figure 3.16: Factor Oracle automaton with the longest repeated subsequence length equal to 9

Factor Oracle with 40 states The larger the automaton gets, the more we can see the importance of the FO algorithm. Automaton in Figure 3.17 has 40 states and even though it looks like the longest repeated subsequence is way larger than the example before. The longest repeated subsequence length is 11, and its *lrs* is `a`, `b`, `b`, `c`, `a`, `b`, `e`, `c`, `d`, `a`, `b`.

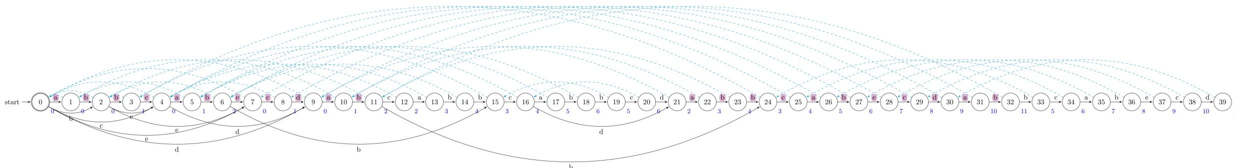


Figure 3.17: Factor Oracle automaton with the longest repeated subsequence length equals to 11

Factor Oracle with two longest repeated subsequences Figure 3.18 shows an automaton that has two longest repeated subsequences: `a`, `b`, `c` and `e`, `f`, `g`. The algorithm identifies completely the two longest lengths and their suffixes correctly. The first state with the biggest length is state 14 and the second one is state 17, both have length 3, but with different subsequences.

Factor Oracle with all symbols repeated As a final example of the FO automaton, the automaton in Figure 3.19 has every symbol repeated. This means that the length of the longest repeated subsequence should be the number of states minus one. This is why we use Figure 3.20 to show the repetition of the sequence. In this case, as all the sequence are eighth as, the longest repeated subsequence length must be seven.

Consequently, the reason why so many experiments were made is that the next two structures, *Audio Oracle* (see Chapter 4) and *Variable Markov Oracle* (see Chapter 5), are based in the FO. Therefore, our motivation to do different experiments was mainly to demonstrate it is working as

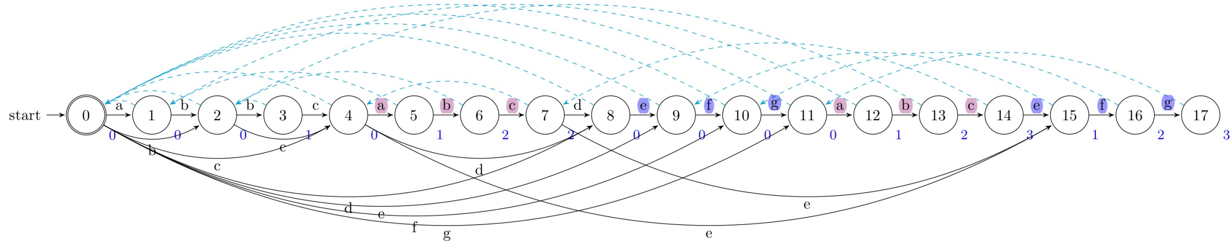


Figure 3.18: Factor Oracle automaton with two Longest Repeated Subsequences with Length 3

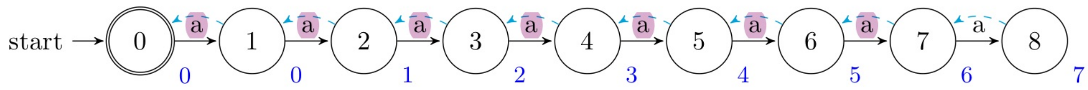


Figure 3.19: Factor Oracle automaton all symbols repeated part one

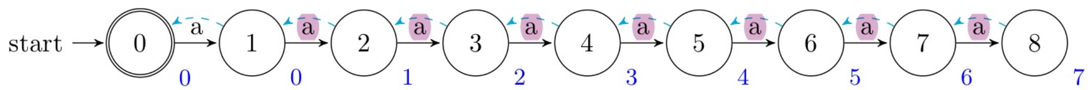


Figure 3.20: Factor Oracle automaton, all symbols repeated part two

expected and to show as many scenarios as possible. For that reason, a lot of time of the thesis was dedicated to do a lot of tests and experiments for the FO.

Once we finished the *Factor Oracle*, the next step was to implement the *Audio Oracle*. The next chapter (Chapter 4) describes the implementation process of the Audio Oracle, such as, the C++ feature extraction library decision, the problems and solutions proposed; some automaton examples and the Audio Oracle integration with `ossia-score`.

Chapter 4

Audio Oracle

The Audio Oracle (AO) algorithm started as an idea of an improvement for the Factor Oracle. As we mentioned in the previous chapter, the FO could be used for musical improvisation if it was modified. Consequently, the AO can be seen as an improvement or successor of the Factor Oracle because it uses its bases. However, the latter operates with symbols, strings, characters and MIDI inputs, while the former operates with audio files.

Equally, the implementation of the AO is done in the programming language C++, mainly because the AO would be integrated with `ossia-score` and `ossia-score` was developed in C++. Nonetheless, AO was integrated as an *addon*, while the FO was integrated in `ossia-score` as a plug-in. In general both are considered plugins, but the integration process of a plug-in in `ossia-score` is different than the integration process of an *addon*.

Furthermore, in this chapter we describe the process of the Audio Oracle development. Firstly, the development starts by defining the Audio Oracle. Secondly, we define the C++ feature extraction library and implement the AO in C++. Thirdly, we integrate the AO with `ossia-score`. And fourthly, we create the automatons representing the Audio Oracle.

Therefore, we shall begin by defining the Audio Oracle in the next Section 4.1.

4.1 Definition

The Audio Oracle (AO) [DAC07] is an indexing structure for audio data that captures repeating sub-clips of variable length called "audio factors". Audio Oracle accepts a continuous (audio) signal stream as input, transforms it into a series of characteristic vectors, and submits those vectors to AO analysis. AO outputs an automaton that consists of suggestions to different locations within the audio data that fulfill certain similarity criteria, as determined through the algorithm [DAC11].

This algorithm takes its bases from the Factor Oracle, but the difference is that it does not assign symbols to transitions, it assigns a vector of audio features to transitions and furthermore, it uses a distance function to assign the degree of similarity between the audio feature vectors using a threshold.

The Audio Oracle algorithm (see Algorithm 6) starts defining the suffixes of all the states as -1, calling the NewAddFrame with every state of the audio stream.

Algorithm 6 On − lineConstruction of Audio Oracle

Require: Audio stream as $S = \sigma_1\sigma_2 \cdots \sigma_N$
1: Create an oracle P with one single state 0
2: $S \leftarrow$ -1
3: **for** $i = 0$ to N **do**
4: $\quad$ *Oracle* $(P = p_1 \cdots p_i) \leftarrow$ NewAddFrame(*Oracle* $(P = p_1 \cdots p_{i-1})$,σ_i)
5: **return** Oracle($P = p_1 \cdots p_N$)

Every time the NewAddFrame (see Algorithm 7) function is called, it creates a new state and creates a new transition from the previous state to this newly created state. This function is similar to the Factor Oracle one, but the difference is that instead of finding the same symbol as it was in the Factor Oracle, it uses a distance function to find the distances between that state and the previous ones, using a threshold to decide if that part of the audio is similar or not to the other ones.

Algorithm 7 NewAddFrame function: Incremental update of Audio Oracle

Require: Oracle $P = p_1 \cdots p_m$, Audio Frame description vector σ and threshold θ
1: Create a new state $m + 1$
2: Create a new transition from m to $m + 1$, $\delta(m, \sigma) \leftarrow m + 1$
3: $\Pi_1 \leftarrow m$
4: $T[S[m + 1]] \leftarrow T[S[m + 1]] \cup \{\}$
5: $k \leftarrow S[m]$
6: **while** $k >$ - 1 **do**
7: $\quad$ Calculate distances between σ and $\delta(k, :)$
8: $\quad$ Find indexes of frames in $\delta(k, :)$ whose distance from σ are less than θ
9: $\quad$ **if** There are indexes found **then**
10: $\quad\quad$ Create a transition from state k to $m + 1$,
11: $\quad\quad$ $\delta(k, \sigma) = m + 1$
12: **if** $k =$ -1 **then** ▷ No suffix exists
13: $\quad$ s $\leftarrow$ 0
14: **else**
15: $\quad$ s $\leftarrow$ where leads the *best* transition (min. distance) from k
16: $\quad$ $S[m + 1] \leftarrow$ s
17: $\quad$ $lrs[m + 1]$ = LengthCommonSuffix($\Pi_1, S[m + 1] - 1$);
18: $T[S[m + 1]] \leftarrow T[S[m + 1]] \cup \{\text{m} + 1\}$

Furthermore, if there are indexes found it keeps creating transitions between state k and state $m + 1$ until there are no more indexes. Nonetheless, if there were no indexes found, the best suffix for the state $m + 1$ would be the state 0. On the other hand, if there were indexes, the suffix of the state $m + 1$ would be where leads the best transition from k.

The LengthCommonSuffix (see Algorithm 8) does the same job as the Factor Oracle one, it makes

sure the minimum distance is achieved between states, to make the automaton the most reliable possible.

Algorithm 8 LengthCommonSuffix(Π_1, Π_2)

1: **if** $\Pi_2 = S[\Pi_1]$ **then**
2:     **return** $lrs[\Pi_1]$
3: **else**
4:     **while** $S[\Pi_1]$!= $S[\Pi_2]$ **do**
5:         $\Pi_2 = S[\Pi_2]$
6: **return** $\min(lrs[\Pi_1], lrs[\Pi_2])$

At last, the basis of the improvisation algorithm (see Algorithm 9) is undoubtedly similar to the Factor Oracle improvisation algorithm, the main difference is the input and output vector V. The input and output vector of the Factor Oracle improvisation algorithm is a symbol vector sequence, meanwhile the input and output vector of the Audio Oracle improvisation algorithm is an audio stream. In this algorithm, the vector V is an audio stream vector, which contains pieces of audio that put together create the audio improvisation.

Algorithm 9 AO − GenerateFunction

Require: Oracle $P = p_1 \cdots p_m$, in active state i, generated audio stream $V = v$ and continuation parameter $0 \leq q \leq 1$

1: Generate uniformly distributed random number p
2: **if** $p < q$ **then**
3:     $v \leftarrow vp_{i+1}$
4:     $i \leftarrow i + 1$
5: **else**
6:     Choose a random symbol $\sigma \in \{ \sigma_j \in \Sigma \mid \delta(S[i], \sigma_j) \neq \perp \}$
7:     $i \leftarrow \delta(S[i], \sigma)$
8:     $v \leftarrow v\sigma$
9: **return** Audio stream $V = v$

After finishing the definition of the AO, we continue with the implementation in C++, which will be shown in Section 4.2.

4.2 Implementation

First of all, to begin the Audio Oracle implementation process, we will define what the inputs are in C++ for the two main processes of the AO algorithm. As shown in the previous section, we have the automaton construction, which includes On − lineConstruction, NewAddFrame and LengthCommonSuffix, while the improvisation generation includes AO − GenerateFunction.

The first process of the Audio Oracle algorithm is creating the Audio Oracle automaton. For this process the inputs are:

- A wav audio file
- A hop size
- The feature that wants to be analyzed from the audio
- The feature threshold

The second process is generating the improvisation based on the Audio Oracle automaton created in the first step. The inputs are:

- The total length of the improvisation, in frame numbers
- The threshold used for the improvisation, a number greater than 0 and smaller than 1
- The hop size, which should be the same as in the first process
- The window size, which usually is double of the hop size
- The wav audio file processed in the first step
- The output file name, this is the name of the improvisation audio file

4.2.1 C++ Feature Extraction Library

To start the Audio Oracle on the right path, an important task was to decide which C++ library we should use as a feature extractor. In the Table 4.1, there are six different columns:

1. The first column *Name* has the name of the C++ libraries that we are comparing
2. The second column *License*, has the license of the library, a software license is a legally enforceable document that outlines how to use and distribute software. End users are usually granted the right to make one or more copies of software without infringing on copyrights, for the library to be compatible with ossia-score, the license needed to be GPLv2, GPLv3 or MIT
3. The third column *Release Year* is the year the library was first released
4. The fourth column *Latest Release* is the year the library was last updated
5. The fifth column *Documentation* is the level of documentation the library has, High means it has a good and robust documentation, Medium means it has good documentation but not as large, Low means it does not have good or complete documentation and None means it has no documentation
6. The sixth column *C++ Integration* denotes if the library is compatible with C++ and if the integration with the algorithms is possible or not, No means it was not possible to integrate it, Yes means it was possible to integrate it

There were some C++ libraries that looked interesting enough, as shown in Table 4.1, some of the libraries were extremely good but the licenses were not compatible with the GPL Version 3, some were good but did not have good documentation. Aubio's license was not compatible with GPLv3 and the integration with C++ was almost null, Essentia did have good documentation but it was not possible to integrate it with C++, openSMILE had a lot of documentation for command line commands but not a lot for C++ integration, as a result, this left us with Marsyas and Gist, and because of the good documentation Marsyas has, the first library chosen was Marsyas. AudioFile was not mentioned to compete with the C++ feature extraction libraries, because it is a library for reading and writing audio files only.

Table 4.1: C++ Audio Libraries

C++ Libraries Information					
Name	**License**	**Release Year**	**Latest Release**	**Documentation**	**C++ Integration**
Aubio	GPL	2003	2017	High	No
Essentia	GPLv3		2021	High	No
Marsyas	GPLv3	2009	2015	Medium	Yes
openSMILE	GPL	2013	2020	High	No
Gist	GPLv3	2014	2020	Medium	Yes
AudioFile	GPLv3	2017	2021	Medium	Yes

The process of deciding which C++ Feature Extraction Library was going to be used to start implementing the Audio Oracle algorithm was complex. Some of the first decisions that were made created a massive door to progress, but with time, some of the most important things that needed to be done were possible with some libraries and not possible with others.

Weighing the pros and cons, as we mentioned, the first library that was chosen was Marsyas, as it can be seen in the table, Marsyas is a really good feature extraction library, it is a robust library with a lot of work behind and has really good documentation. Moreover, the complexity of integrating Marsyas with C++ to extract the audio features was not high, but because it was a really coupled library, from the first step of the processing until the last step of the improvisation, everything needed to be done explicitly with Marsyas.

At first, the results while extracting audio features, mainly the *centroid*, were really good. The algorithm was coming together and working as expected. But soon afterwards, when starting to try to extract other features such as Zero Crossings, Root Mean Square and most importantly MFCC (Mel-frequency cepstral coefficients), the results were not good or even in some cases, there were not results given. The reason behind these results can be that, probably the configuration of the library inside the project was not done correctly, mainly because the C++ configuration is not explained in the documentation. Therefore, after analyzing the situation, it was decided that Marsyas was not going to be the final library for the Audio Oracle algorithm, nonetheless, the progress done with Marsyas can be seen in the Section 4.2.2.

After finishing the analysis of the remaining libraries using Table 4.1, the best choice was to continue with the project using Gist. Gist is an audio analysis library developed in C++, its main purpose is to process audio frames and extract their audio features. In addition, some of the most important characteristics Gist has are: it is not a big library but it is explicitly a feature extraction library, it has an impressive documentation, and the complexity of the integration with the Audio Oracle was not high. Even though it had really good benefits, a result of changing the feature extraction library was that there was an extra effort which needed to be done to change the implementation of the Audio Oracle and complete it.

Moreover, we know that as Gist is an audio feature extraction library, we needed to find a library to read and write audio files, this library needed to work very well with C ++. After searching, we found AudioFile, which was implemented by the same developer that created Gist. AudioFile is a header-only C ++ library for reading and writing audio files, as it was a header-only library, it was not hard to add it to our project to start reading and writing audio files. This was very important because it was the core to start and finish our algorithm.

Furthermore, as mentioned before, Marsyas is a coupled library, this created a lot of dependencies between the Audio Oracle and Marsyas. For this reason, the process of changing the C++ library took more time and effort than expected, but this gave us the opportunity to use design patterns to make the code more independent and less coupled. The design patterns used to this were the Facade Pattern and the Strategy Pattern, *Facade* is a higher-level interface that simplifies the use of a subsystem. A set of interfaces in a subsystem is given a unified interface via Facade. While *Strategy* defines a set of algorithms that are encapsulated and interchangeable, it allows the algorithm to change independently of the users who utilize it.

Consequently, after having chosen Gist, we started analyzing the features it provided, and even though there were many features available for extraction, seven were chosen. The seven features that were chosen, were chosen because of two reasons, the first one, to be able to analyze the different types of features, such as, Core Time Domain Features, Core Frequency Domain Features, Onset Detection Functions, Pitch, and Mel-frequency Representations; the second one, is because they are some of the main audio features, such as, MFCC, Zero Crossing Rate, Root Mean Square, Spectral Rolloff, Spectral Centroid, Pitch and Energy. The Spectral Centroid was the feature that gave the best results using Marsyas, therefore, extracting the Spectral Centroid using Gist was an interesting experiment to do.

We shall continue by showing the implementation that was done with Marsyas, even though, this was not the final library used for the AO implementation.

4.2.2 Implementation with Marsyas

To start testing the Audio Oracle algorithm with Marsyas, we created audio improvisations using different input values. Most of the Figures shown afterwards, have an anchor and a link on the side for redirection to Youtube, this anchor is located on their label and it shows the improvisation in Youtube.

At first, the different input values were the audio stream vector, threshold q which was used to define the similarity between the original audio and the improvisation, the length of the improvisation in states, and a hanning repetition value.

These mentioned inputs can be seen in Figure 4.1 which is the original audio, Figure 4.2 and Figure 4.3, these audios represent the call of a Blue Jay bird. On a side note, the hanning repetition was being used as an input before the Hanning window algorithm was fixed, which is why after Figure 4.5 the inputs changed. This change of inputs gives us the opportunity of seeing the difference of the improvisation of the audio Faded, firstly with the Hanning window not working properly and secondly after being fixed.

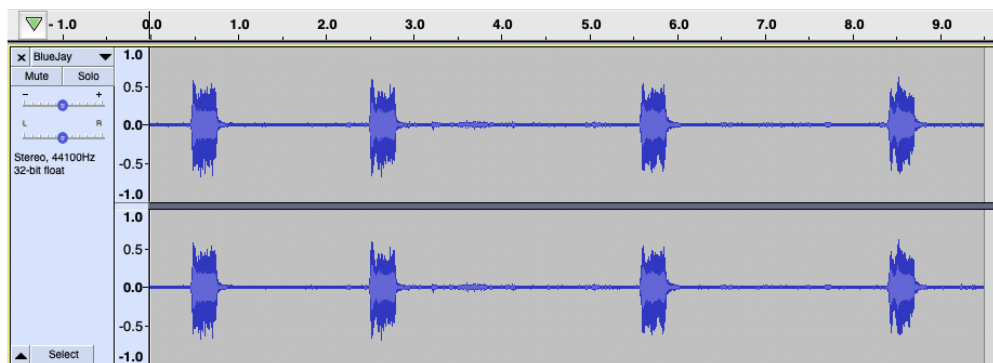


Figure 4.1: Blue Jay Original Audio

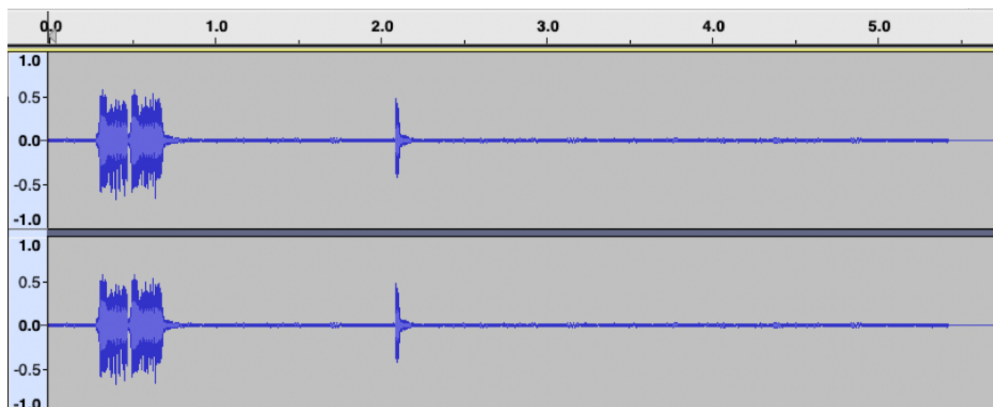


Figure 4.2: Blue Jay Improvisation, Hanning repetition=50, q=0.8, length=90, feature=centroid

Even though, the hanning window was not working as expected, changing the hanning repetition input from 50 to 35 changed the audio improvisation, making the most repeated sub audio clips repeat more times. Another thing to note from the original audio is that, despite having a lot of "silence" or "empty noise", the Audio Oracle was able to identify the blue jay bird calls and included them in the improvisation.

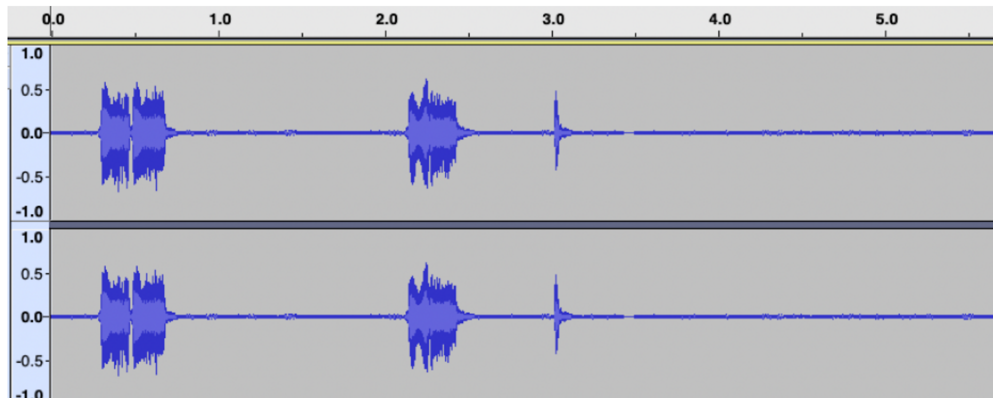


Figure 4.3: Blue Jay Improvisation, Hanning repetition=35, q=0.8, length=60, feature=centroid

After seeing the above example, we wanted to start experimenting with popular songs but only using the instrumental version of them. The first song we chose was Faded by Alan Walker, which is shown in Figure 4.4. Moreover, in Figure 4.5, we created the first improvisation extracting the centroid feature, and as seen, it created a new audio, this improvisation started in state 0. The state 0 refers to the state of the AO automaton. In this chapter, when we talk about a state, we are referring to a state of the Audio Oracle automaton.

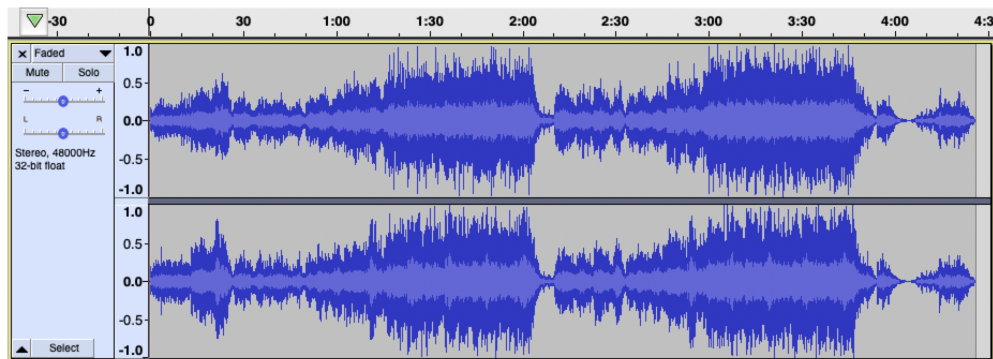


Figure 4.4: Faded Orchestra Instrumental Audio (`https://bit.ly/3xOs65c`)

In the Figure 4.5, the hanning repetition is still present, even though, it is not possible to visually see it because it was only an extra input. Also, the hanning window is not fixed, but in the Figure 4.6 it is already fixed. Hence, there are differences present in these two figures, there are silent spaces in the first figure, which should not occur constantly because the hanning window does not allow discontinuity; meanwhile in the second figure, the starting state is 1700, avoiding any silence that could be present in the audio and consequently, helped create a more natural improvisation.

The next example is created with the song Drivers License, the original instrumental audio is shown in Figure 4.7. In Figure 4.8, the starting state is 10 and what this means is that the improvisation

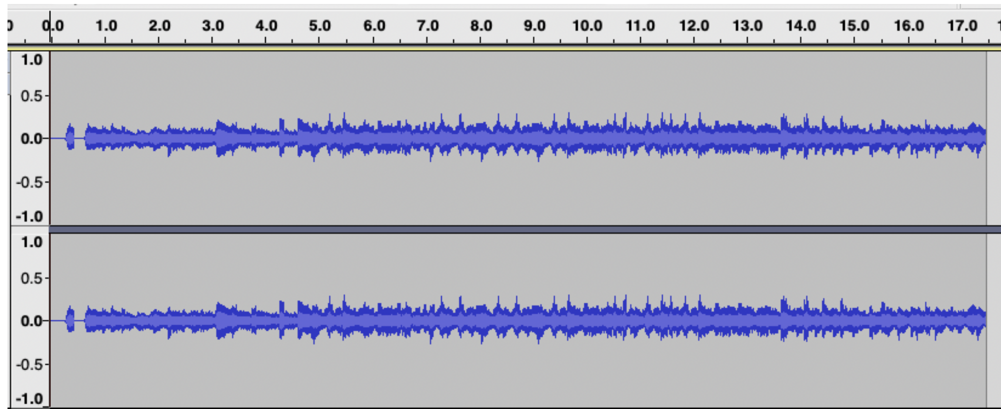


Figure 4.5: Faded Improvisation (`https://bit.ly/3jLD6LK`), Hanning repetition=35, q=0.8, length=120, feature=centroid

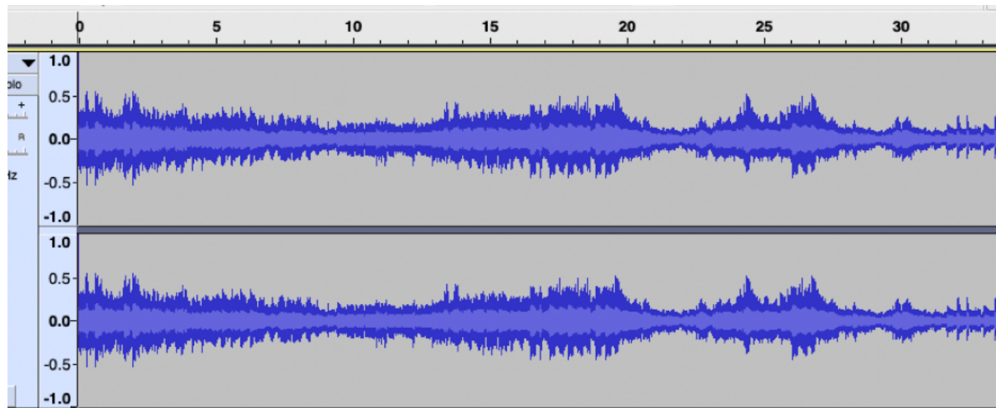


Figure 4.6: Faded Improvisation (`https://bit.ly/37GW0Ok`), starting state is 1700, q=0.8, length=400, feature=centroid

starts in state 10 and then depending on the threshold q and p it goes to the next state or chooses the longest repeated subsequence related to the state it is at. For this reason, as the threshold is 0.4, the improvisation will usually goes to the longest repeated subsequence.

After trying the centroid feature extraction, the next step was trying different features, such as, power spectrum, spectrum and others. In Figure 4.9, the feature that was extracted was power spectrum. In this case, as the threshold q is 0.4, and the starting state is 500, what can be seen, is that the same audio is being repeated over and over again, this can happen because of the threshold or because of the feature.

In Figure 4.10, the feature extracted was the spectrum, the threshold and starting state are the same as the previous figures, the difference is the hop size is 16384, instead of 4096 as some of the previous audios. The most important detail from the improvisation is that it repeats the longest

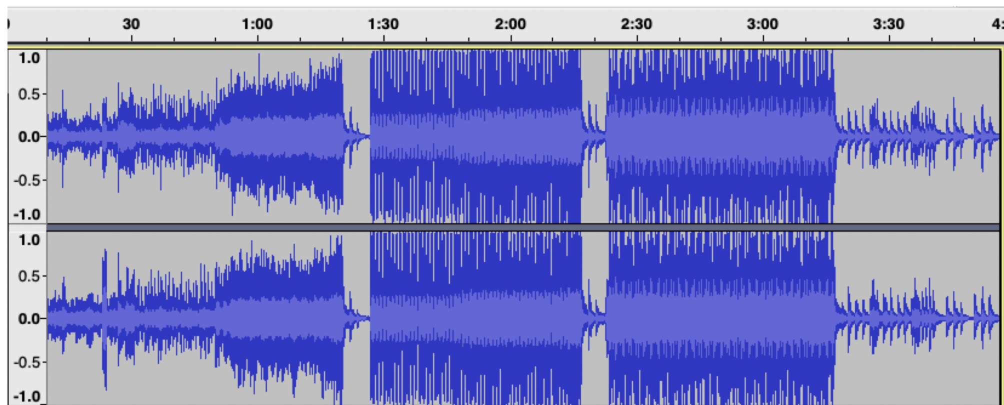


Figure 4.7: Drivers License Original Instrumental Audio (`https://bit.ly/3sgCG3Q`)

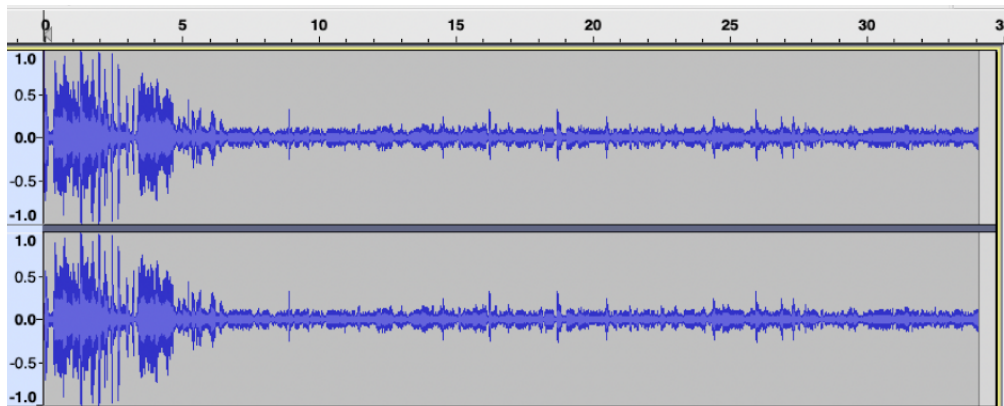


Figure 4.8: Drivers License Improvisation (`https://bit.ly/37Fapud`), starting state is 10, q=0.4, length=400, hop size=4096, feature=centroid

repeated subsequence a lot of times, similarly as in Figure 4.8, but the repetitions are not the same.

To finish the improvisations with Marsyas, in Figure 4.11, the extracted feature was the centroid, in this improvisation we can see that regarding the audio, even though, the same inputs were used, the feature changed the outcome of the improvisation. The outcome shows that as q is 0.4, the improvisation tries to loop through its lrs, we hear that it tries to stay in the same states for longer periods of time, this can be analyzed by seeing the waveform and hearing the improvisation.

Finally, after finishing the implementation using Marsyas, we will show the process of the AO implementation using Gist in Section 4.2.3

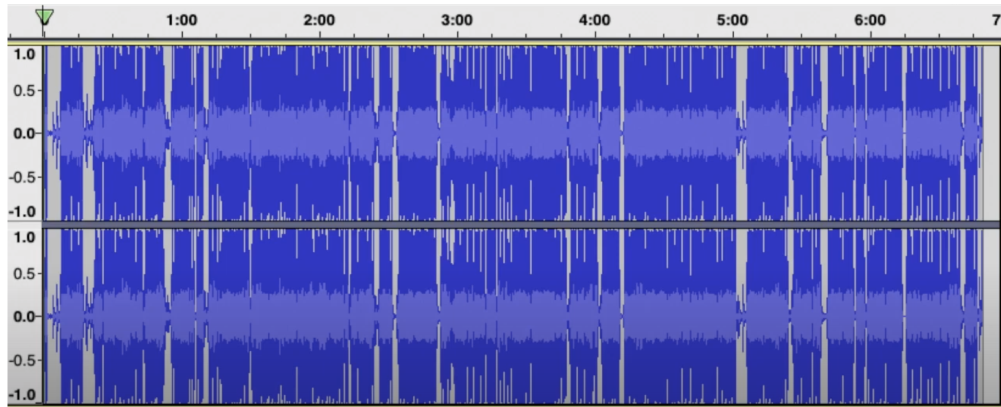


Figure 4.9: Drivers License Improvisation (`https://bit.ly/3fXgwi4`), starting state is 500, q=0.4, length=1200, hop size=4096, feature=power spectrum

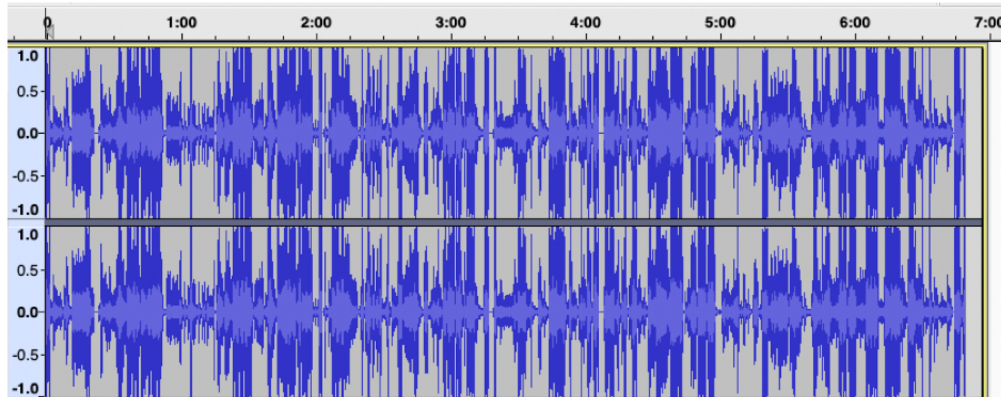


Figure 4.10: Drivers License Improvisation (`https://bit.ly/37HVLlU`), starting state is 500, q=0.4, length=1200, hop size=16384, feature=spectrum

4.2.3 Implementation with Gist

After doing the implementation and some audio improvisations with Marsyas, we started creating improvisations using Gist, as we mentioned before, Gist is an audio analysis library and its main purpose is to process audio frames and extract their audio features. It has mainly six audio feature groups that can be extracted, in these audio improvisations shown below we extract features from the groups: *Core Time Domain Features*, *Core Frequency Domain Features*, *Pitch* and *Mel-frequency Representations*.

In addition, one thing that we implemented when we started using Gist, was normalization. As we defined it before, normalization is a data-shifting and rescaling technique in which data points are shifted and rescaled until they're in the 0 to 1 range. We used normalization to rescale the features extracted from each audio factor. The benefit of doing this was that the feature threshold would

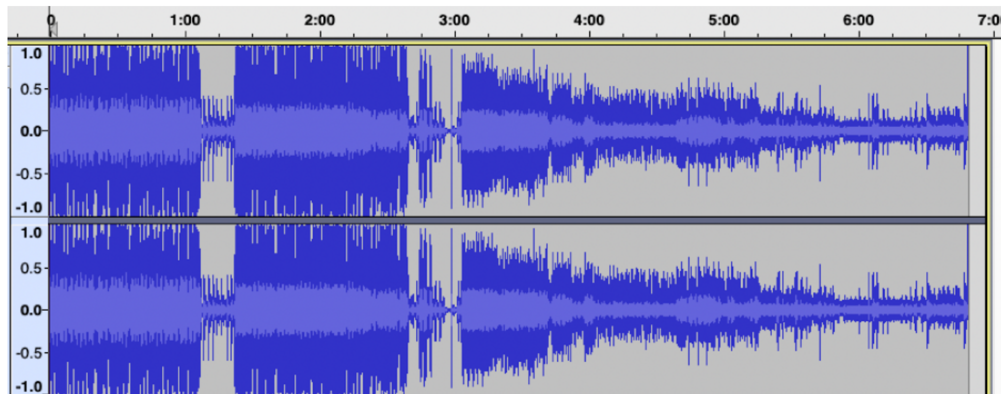


Figure 4.11: Drivers License Improvisation (https://bit.ly/3iLhG2a), starting state is 500, q=0.4, length=1200, hop size=16384, feature=centroid

be easier to find and analyze. Before the normalization process, the optimal threshold could be 50, while after the normalization process it could be 0.04. Also, implementing the normalization, the range of the threshold could only range from 0 to 1.

In Figure 4.12, we see the original Drivers License Instrumental Audio. Next, we will see the audio improvisations derived from this original audio.

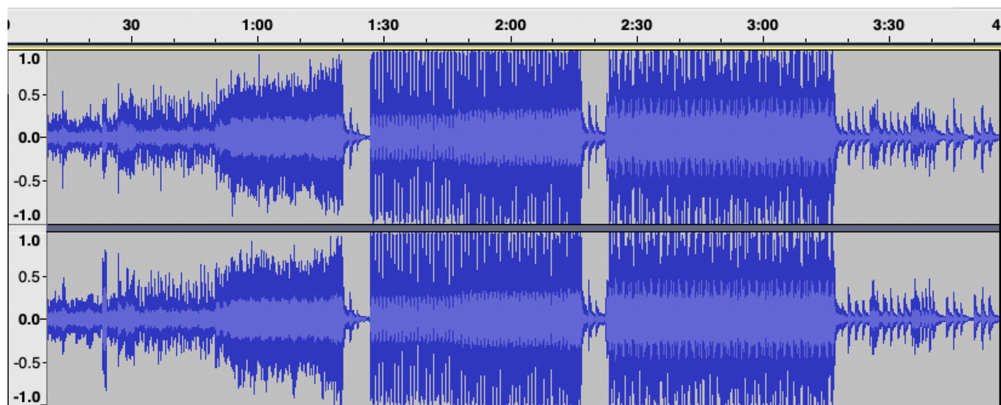


Figure 4.12: Drivers License Original Instrumental Audio (https://bit.ly/3sgCG3Q)

In Figure 4.13 we see the improvisation starts in state 400, what we call the *feature threshold*, is the threshold that is used to determine the distance between frames and their similarity when extracting the features. In this example, the feature threshold is 0.08, this means that if two frames have a distance smaller than 0.08 they will be considered similar. Also, as the q is 0.8, the improvisation will try to be similar to the original audio, going less times to the longest repeated subsequence. In particular, this improvisation using the Spectral Rolloff as the extraction feature, tries to go between states 250 and 500 of the original audio's automaton, meanwhile taking into account the

original audio.

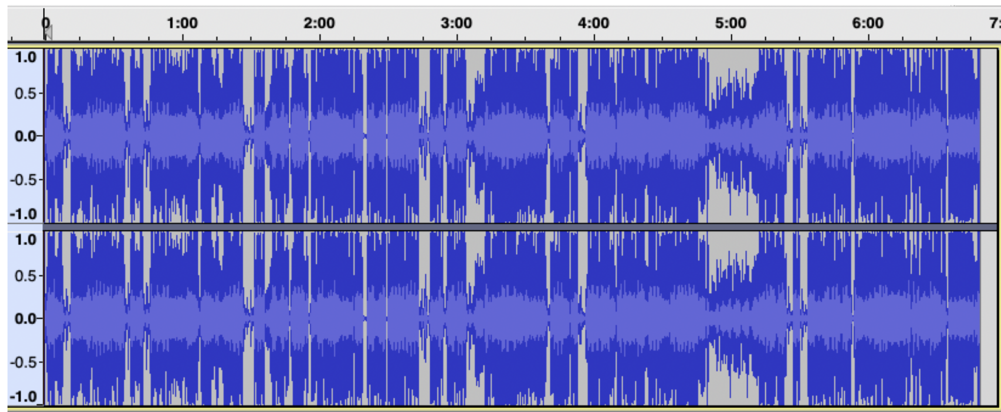


Figure 4.13: Drivers License Improvisation (`https://bit.ly/37HcbLy`), starting state is 400, feature threshold=0.08, q=0.8, length=1200, hop size=16384, feature=Spectral Rolloff

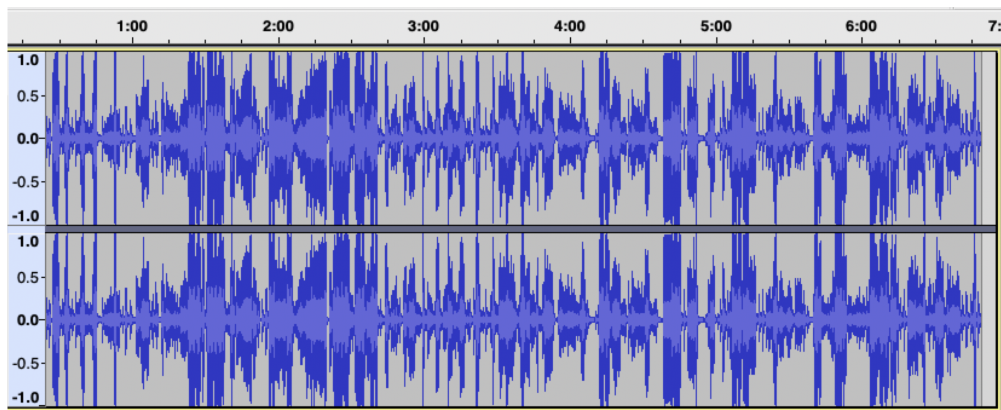


Figure 4.14: Drivers License Improvisation (`https://bit.ly/3yN7Z8G`), starting state is 400, feature threshold=0.1, q=0.8, length=1200, hop size=16384, feature=Spectral Centroid

Furthermore, in Figure 4.14 we see that the feature extracted was the Spectral Centroid, here the improvisation starts in state 400, the feature threshold is 0.1 and, as the q is 0.8, the improvisation will try to be similar to the original audio and even though it will go to the longest repeated subsequence, it will not stay in a loop there.

On the other hand, in Figure 4.15, the main two differences are the feature threshold and q, in this particular example, the most important difference is q, because it defines the changes in the improvisation. In the figure shown below, q is 0.2 and the feature threshold is 0.08, as q is 0.2, this means that the improvisation will try to repeatedly continue going to the longest repeated subsequence based on the feature, in this case, the differences are extreme between Figure 4.14 and Figure 4.15, because of the threshold that was defined for the improvisation.

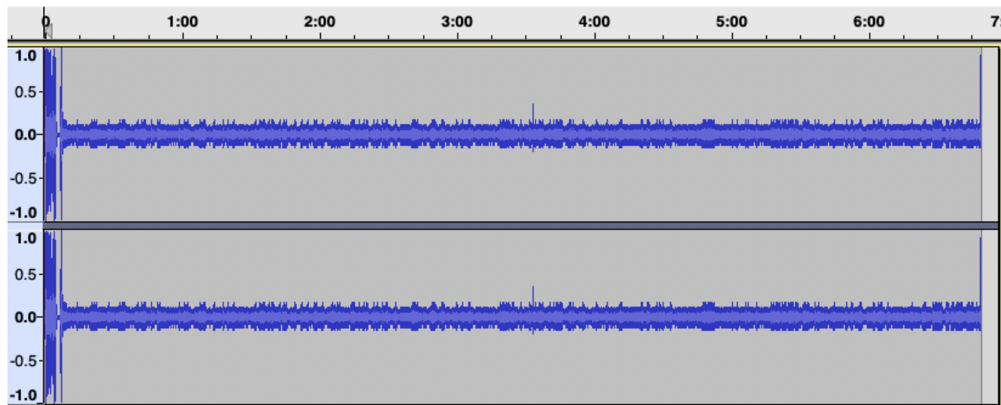


Figure 4.15: Drivers License Improvisation (`https://bit.ly/3iFb4Cd`), starting state is 400, feature threshold=0.08, q=0.2, length=1200, hop size=16384, feature=Spectral Centroid

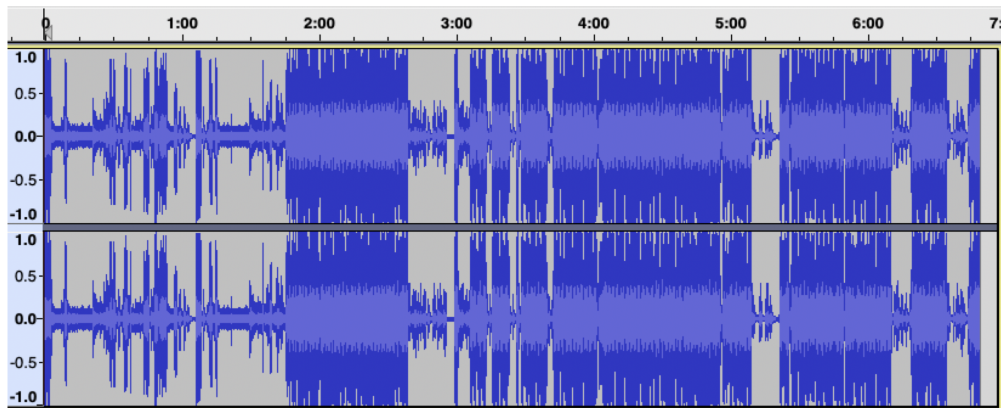


Figure 4.16: Drivers License Improvisation (`https://bit.ly/3seJq28`), starting state is 400, feature threshold=0.08, q=0.8, length=1200, hop size=16384, feature=Zero Crossings

Moreover, in Figure 4.16, the feature extracted was Zero Crossings. As we mentioned before, Zero Crossings is the rate at which a signal's sign shifts from positive to negative or vice versa in a given amount of time. Here, we have $q = 0.8$, thus it tries to be more similar to the original audio.

In Figure 4.17, we are extracting the Root Mean Square, and even though the starting state, the feature threshold and q are the same as in Figure 4.16, the improvisations are really different, this is why the feature extracted is really important when creating an audio improvisation.

Meanwhile, in Figure 4.18 we are extracting the MFCC, with the same improvisation threshold and q as the improvisation in Figure 4.17. It can be seen that the improvisation usually goes to the states that have a larger magnitude, in this case, the part of the song that is more repeated in the improvisation is the second verse of the song and the chorus.

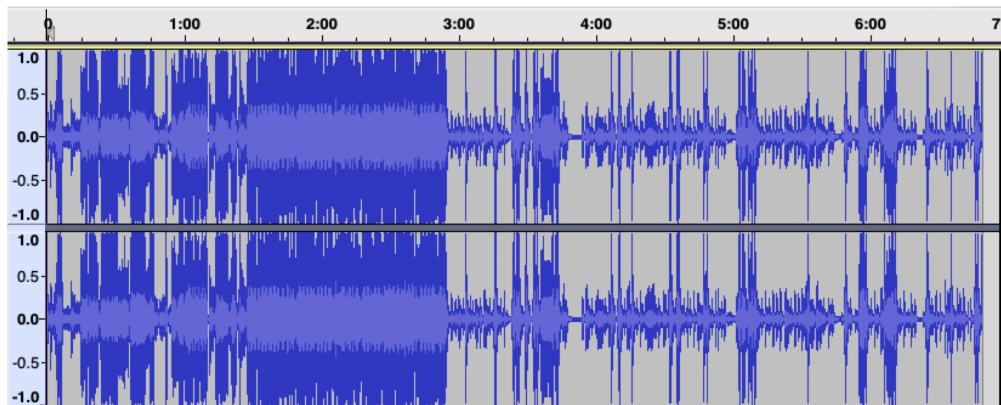


Figure 4.17: Drivers License Improvisation (`https://bit.ly/3yXauoY`), starting state is 400, feature threshold=0.08, q=0.8, length=1200, hop size=16384, feature=Root Mean Square

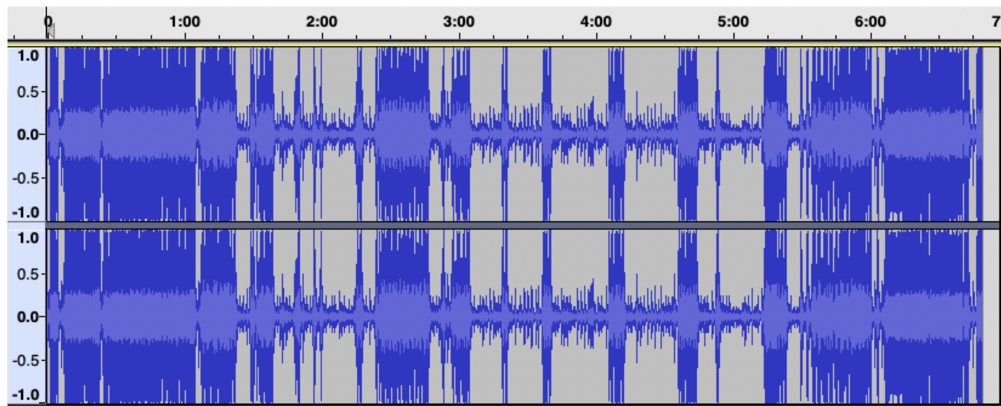


Figure 4.18: Drivers License Improvisation (`https://bit.ly/3fZuJuS`), starting state is 400, feature threshold=0.08, q=0.8, length=1200, hop size=16384, feature=MFCC

Finally, the improvisation that is shown in Figure 4.19 is completely different than the other ones, this improvisation is done extracting the Pitch feature, the Pitch extraction is an estimation of the pitch. Particularly, what we can identify is that this improvisation stays in the last parts of the song and never leaves these states, this is because of the feature extracted and naturally, the Audio Oracle created is different from the other ones.

4.3 Audio Oracle in `ossia-score`

Following the Audio Oracle implementation, validating that everything was working as expected and that the Audio Oracle algorithm was creating the improvisations, we started to work on the Audio Oracle integration with `ossia-score`.

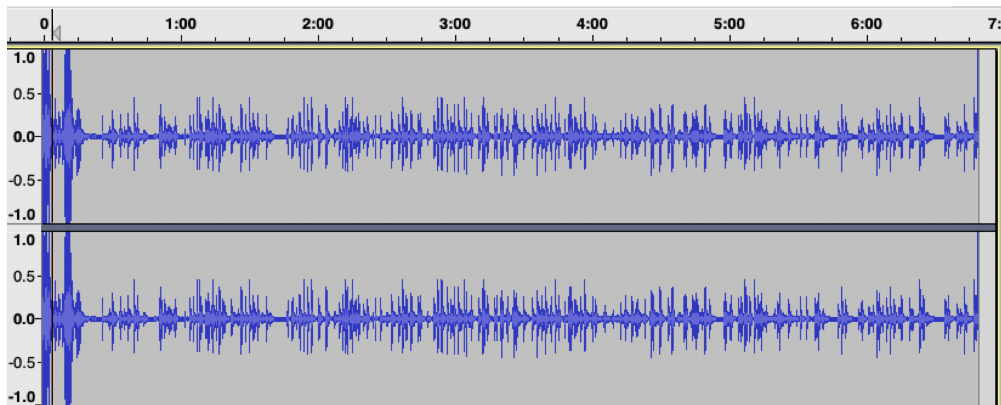


Figure 4.19: Drivers License Improvisation (`https://bit.ly/3sgXqso`), starting state is 400, feature threshold=0.08, q=0.8, length=1200, hop size=16384, feature=Pitch

The integration process had to be defined with Dr. Jean-Michaël Celerier, the creator and main developer of `ossia-score`. We had to schedule meetings to see what would be the best course of action for the Audio Oracle. The most important reason was that, the Factor Oracle and Audio Oracle worked in a completely different way, the Factor Oracle works as a plug-in, while the Audio Oracle needs to modify an audio file.

One thing to keep in mind is that `ossia-score` is an application that works really well with audio files, MIDI inputs and most musical inputs. This means that an user can upload an audio file just by drag and dropping it inside the application.

After a deep analysis, Dr. Celerier and the `ossia-score` team came up with the idea of using the Audio Oracle as an *addon* for transforming sound files (i.e. in `ossia-score` a `.wav` file is named as a sound file). The most important reason for this decision is that it gives the `ossia-score` users a new important feature that can be very useful for them. Especially, because there were none audio improvisation features inside `ossia-score`, giving the users more value integrating the Audio Oracle and also the Variable Markov Oracle as an *addon*, rather than a plug-in.

Furthermore, the final changes of the Audio Oracle are here (`https://bit.ly/3iGIcJZ`), we created a pull request to integrate it with `ossia-score`, this pull request is being reviewed by Dr. Celerier.

In Figure 4.20, we see the original Drivers License audio already in the `ossia-score` application, in the application we have the audio selected and at the top right we see the name of the file two times, the second one is the reference to the sound file. In Figure 4.21, we right click on the sound file name and then on `Offline processing`, here we see the Audio Oracle option, afterwards, in Figure 4.22 we hover over the Audio Oracle option.

Moreover, in Figure 4.23, we can see that in the new sound file, which is the improvisation created after running the Audio Oracle addon, the feature threshold, the hop size, q and the feature extracted are currently static. One difference from these and the Audio Improvisations from the last section

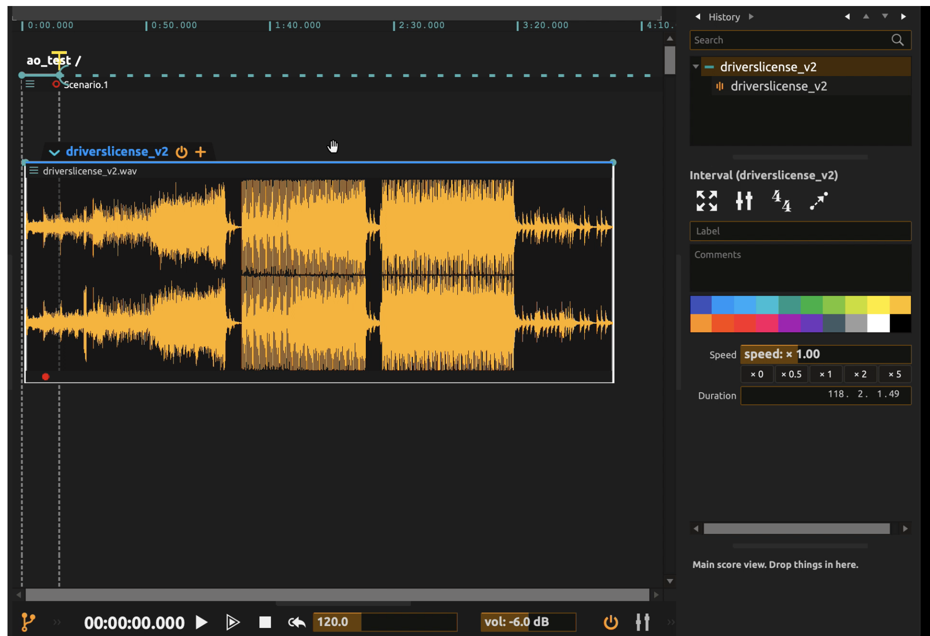


Figure 4.20: Drivers License Audio (`https://bit.ly/3m0YmQh`) in `ossia-score`

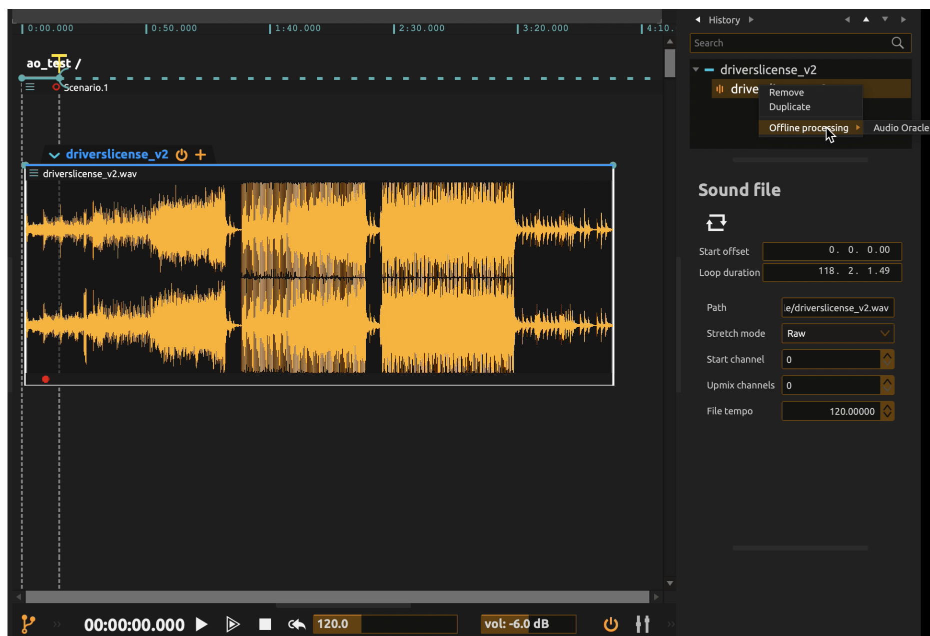


Figure 4.21: Drivers License Audio (`https://bit.ly/3m0YmQh`) in `ossia-score`, Offline process

is that before, the length of the improvisation could be larger or lesser than the original audio's length, but in `ossia-score`, the improvisation is exactly the length of the original audio.

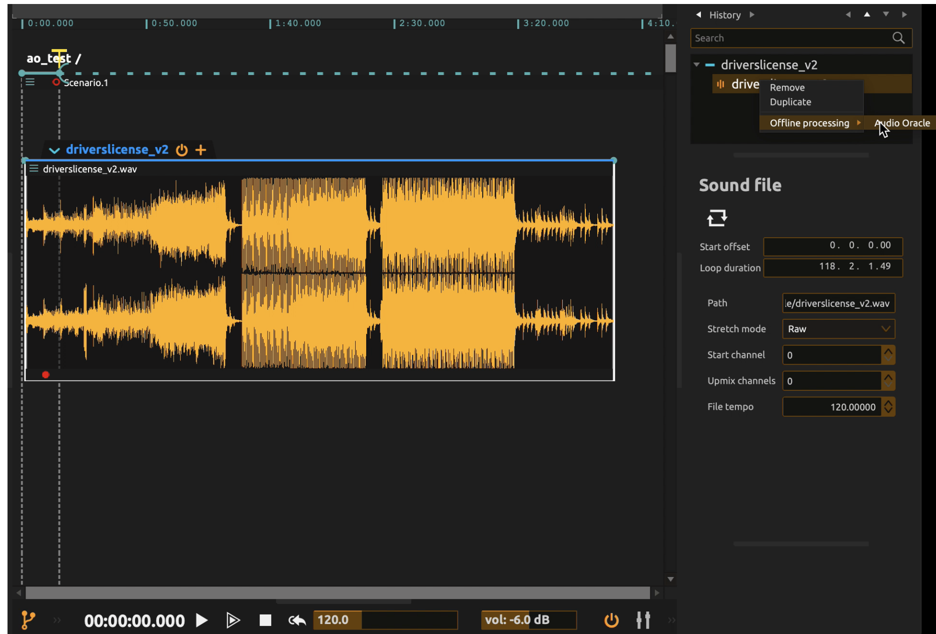


Figure 4.22: Drivers License Audio (`https://bit.ly/3m0YmQh`) in `ossia-score`, hover over Audio Oracle option

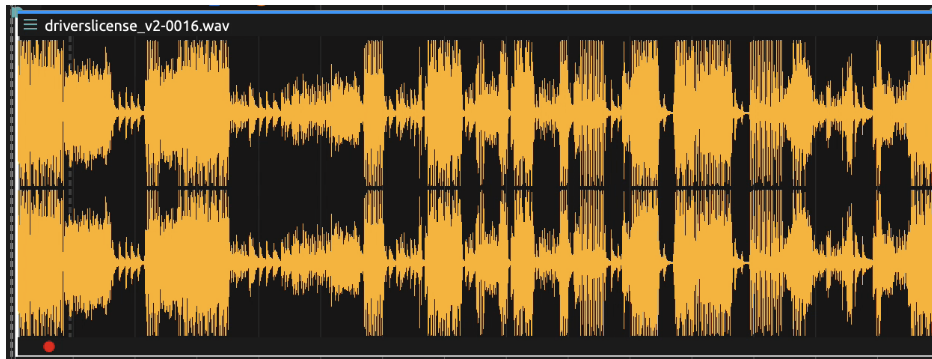


Figure 4.23: Drivers License Improvisation (`https://bit.ly/3m0YmQh`) in `ossia-score`

To finish this process, in Figure 4.24, we see the new Drivers License improvisation, which has a hop size of 32,768, feature threshold of 0.08, q of 0.8 and the feature extracted is Spectral Rolloff.

After doing some analysis, we realized that the users could have more benefits if they were able to choose exactly the feature they wanted to extracted. This is why, we added all the Audio Oracle features and Variable Markov Oracle features as *addons* in `ossia-score`, as shown in Figure 4.25. In this chapter we are only talking about the Audio Oracle, but in Chapter 5 we will show all the process of creating the Variable Markov Oracle *addon* and improvisations.

Figure 4.24: Drivers License Improvisation (https://bit.ly/3m0YmQh) in ossia-score $q = 0.8$, length equal to the original audio, hop size=32768, feature=Spectral Rolloff

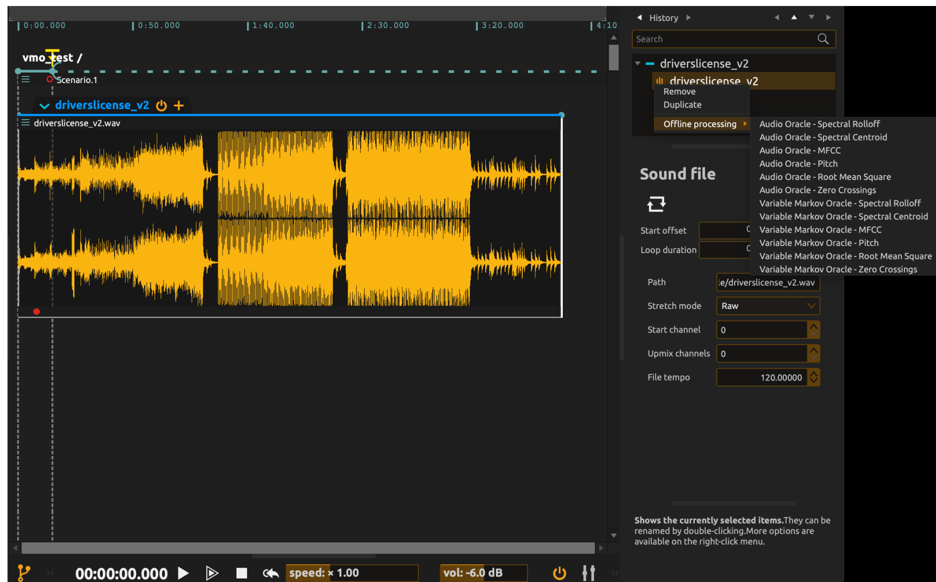


Figure 4.25: All Audio Oracle and Variable Markov Oracle *addons* in ossia-score

As a result of creating all the other *addons*, in Figures 4.26 to 4.30 we see the audio improvisations after using the implemented *addons*.

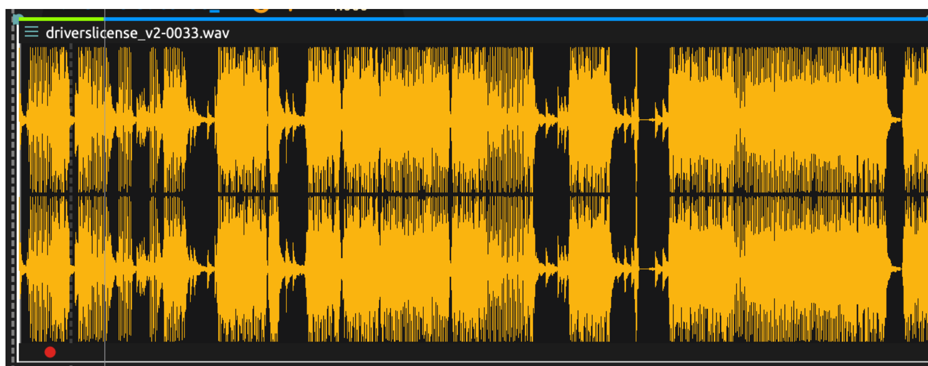


Figure 4.26: Drivers License Improvisation (`https://bit.ly/3AMkDpn`) in `ossia-score`, q=0.8, length equal to the original audio, hop size=32768, feature=Spectral Centroid

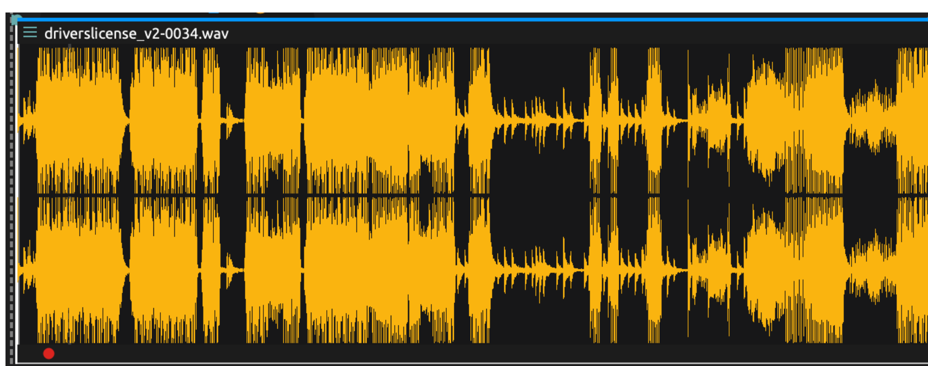


Figure 4.27: Drivers License Improvisation (`https://bit.ly/3m3DTu3`) in `ossia-score`, q=0.8, length equal to the original audio, hop size=32768, feature=MFCC

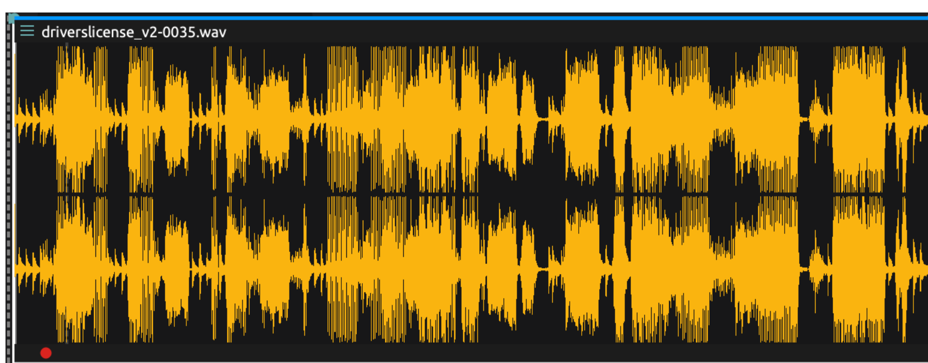


Figure 4.28: Drivers License Improvisation (`https://bit.ly/3yOzgYt`) in `ossia-score`, q=0.8, length equal to the original audio, hop size=32768, feature=Pitch

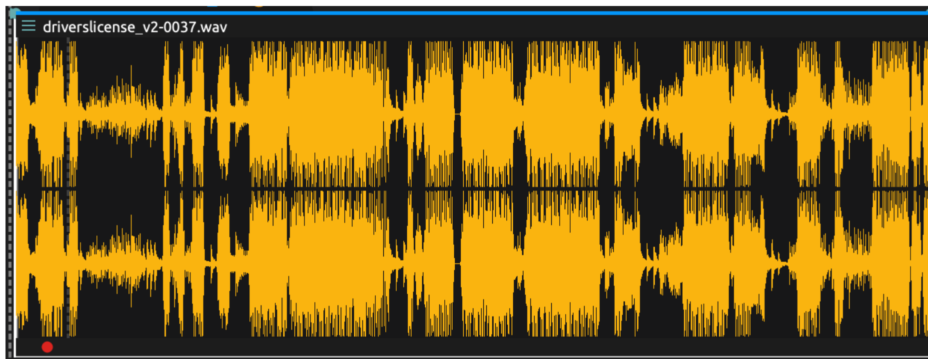


Figure 4.29: Drivers License Improvisation (`https://bit.ly/3yQTSiQ`) in `ossia-score`, q=0.8, length equal to the original audio, hop size=32768, feature=Root Mean Square

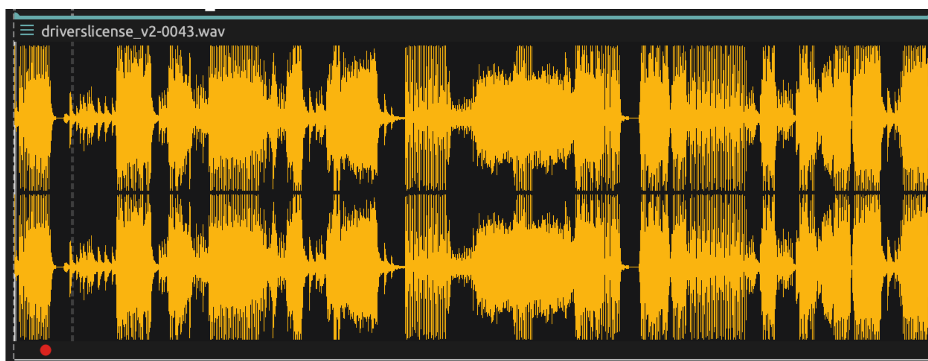


Figure 4.30: Drivers License Improvisation (`https://bit.ly/37DqII6`) in `ossia-score`, q=0.8, length equal to the original audio, hop size=32768, feature=Zero Crossings

4.4 Automaton Creation

The automaton creation of the Audio Oracle is different from the Factor Oracle automaton creation. In the Audio Oracle automaton there is not a symbol in the *internal* forward transitions, there is a number that indicates the feature extracted from the audio frame, which is symbolized in each state, in all of the features except the MFCC, there is only one number extracted from each audio frame, meanwhile, in the MFCC it is a vector, but in the automaton only the first value of the vector will be shown. And in the *external* transitions, we use the state from where the feature value is shown.

Moreover, one important difference is that in the Audio Oracle, the extracted feature changes the automaton extremely, because each feature is different, the automatons will have notable changes and the longest repeated subsequences will not be the same. In addition, as in the Factor Oracle the figures that will be shown below were generated with the implemented tool, in this case, the Audio Oracle.

Similarly as the Factor Oracle, what we are looking for when creating the automaton is finding the longest repeated subsequence or subsequences, in this case, we will find it using the features that were extracted. The input of the Audio Oracle are the audio file, the feature threshold, hop size and the feature to be extracted. In the following examples, the hop size used was an incredibly large one for the automaton to fit the document, the hop size was 262,144.

All of the automata shown below are created with the base audio “Drivers License” instrumental by Olivia Rodrigo. In Figure 4.31, we extracted the Spectral Centroid feature to create the Audio Oracle automaton. The threshold used to determine the similarity between audio frames was 0.08 and as mentioned before, the hop size was 262,144. This hop size is only used for the automata but it is not used to create any improvisation.

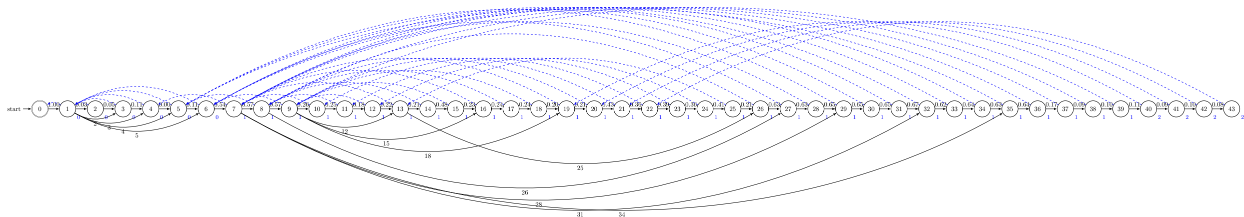


Figure 4.31: Audio Oracle automaton, feature extracted is Spectral Centroid, feature threshold = 0.08

For instance, in Figure 4.32 and Figure 4.33 the automata were created extracting the Spectral Rolloff, but it is notable that the automata are different. The automaton in Figure 4.32 has a feature threshold of 0.08 while the automaton in Figure 4.33 has a feature threshold of 0.05, this small difference of 0.03 can seem insignificant but it is indeed what can make the longest repeated subsequence and the forward and suffix transitions change radically. In this case, the lrs of Figure 4.32 is 3, while in Figure 4.33 the lrs is 2.

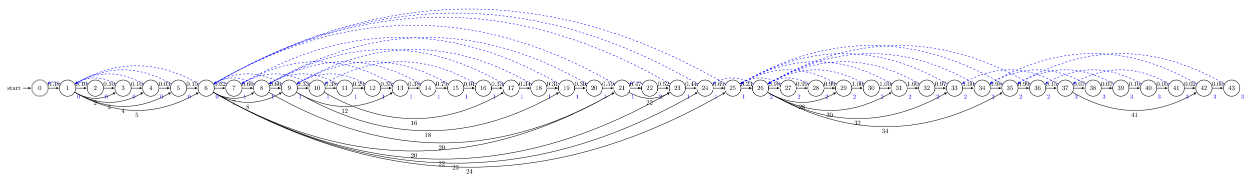


Figure 4.32: Audio Oracle automaton, feature extracted is Spectral Rolloff, feature threshold = 0.08

Continuing with a similar example, in Figure 4.34 and Figure 4.35, the automata were created

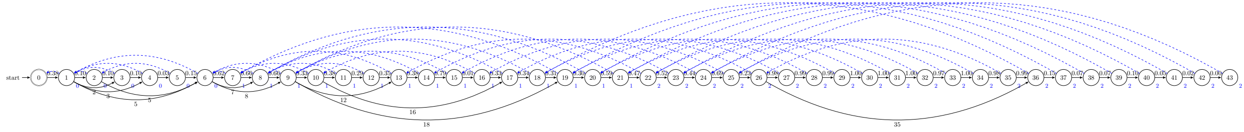


Figure 4.33: Audio Oracle automaton, feature extracted is Spectral Rolloff, feature threshold = 0.05

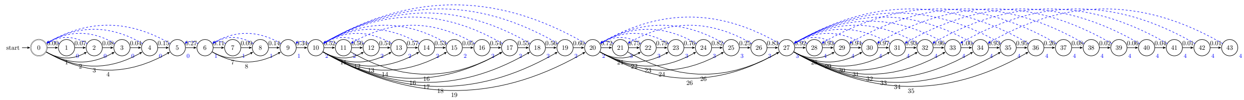


Figure 4.34: Audio Oracle automaton, feature extracted is Root Mean Square, feature threshold = 0.08

extracting the Root Mean Square, as seen in the previous example, the automata created have significant differences, the suffix links and forward transitions are not the same in both figures, but what is more important is that the longest repeated subsequence of Figure 4.34 is 4 and in Figure 4.34 it is 3, which means that with feature thresholds, the lrs can change.

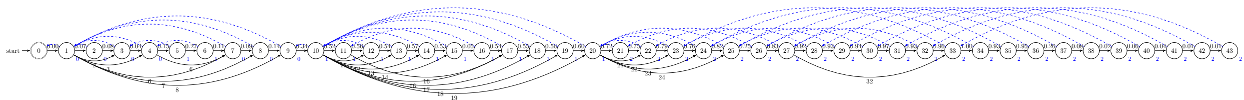


Figure 4.35: Audio Oracle automaton, feature extracted is Root Mean Square, feature threshold = 0.05

In addition, the last two automata, which are shown in Figures 4.36 and 4.37, were created extracting the Zero Crossings and Mel-frequency cepstral coefficients respectively.

Finally, after creating the Audio Oracle automata, we introduce the next chapter, which explains the Variable Markov Oracle implementation, its definition, integration with `ossia-score` and graph creation.

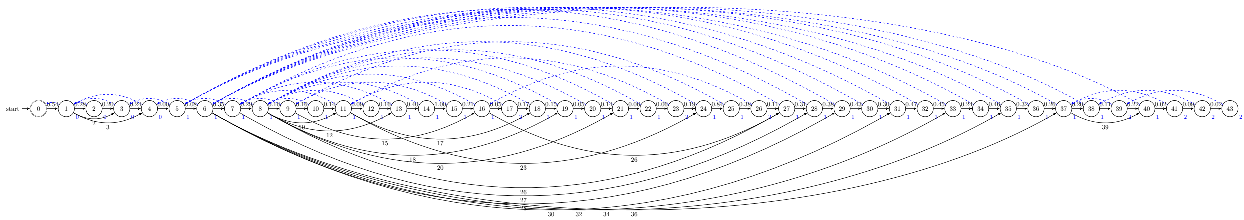


Figure 4.36: Audio Oracle automaton, feature extracted is Zero Crossings, feature threshold = 0.08

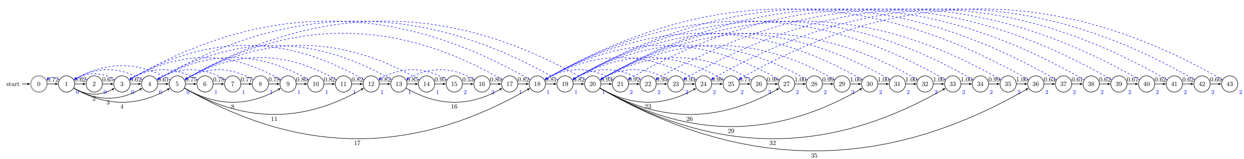


Figure 4.37: Audio Oracle automaton, feature extracted is MFCC, feature threshold = 0.08

Chapter 5

Variable Markov Oracle

The Variable Markov Oracle (VMO) algorithm is the last algorithm after the Factor Oracle and Audio Oracle. It is similar to both, but the main difference is that it uses a structure called Clustering, while the FO and AO use states to create an automaton.

Moreover, the implementation of the VMO is done in the programming language C++, for the same reasons as the AO, the VMO would be integrated with `ossia-score`, and `ossia-score` was developed in C++. Similarly, VMO was integrated as an *addon* as the Audio Oracle.

In this chapter we describe the process of the Variable Markov Oracle development. Firstly, we start by defining the Variable Markov Oracle. Secondly, we implement the VMO in C++, using the bases of the AO but adding the Clustering implementation. Thirdly, we integrate the VMO with `ossia-score`. Finally, we create the graphs representing the Variable Markov Oracle.

5.1 Definition

The Variable Markov Oracle [WD15], also known as VMO, is the successor of Factor Oracle and Audio Oracle. This algorithm resembles the suffix structure used in FO, the difference is that it does not have symbols attached to states. The VMO improves Audio Oracle by especially assigning labels to frames connected by suffix links during Audio Oracle construction.

Algorithm 10 is similar to the Factor Oracle and Audio Oracle construction algorithms, the difference is the way it keeps track of the connected states and how it preserves the on-line nature of the algorithm construction. The main difference is the new element Σ, which is a list of lists of the pointers to $\boldsymbol{O}$, another way we call it is Clustering. Roughly, $\boldsymbol{O}$ is an incoming signal and the input accepted by VMO. Furthermore, the mathematical definition of Σ is: $\Sigma = [\sigma_1, \cdots, \sigma_m, \cdots, \sigma_M]$, where M is the number of labels, which are classifying names corresponding to the forward links and σ_m is a list that contains the pointers for the mth label. Finally $\boldsymbol{P}$ is the label sequence for labels of observations $\boldsymbol{O}$, defined as $\boldsymbol{P}{=}p_1, \cdots, p_t$.

In addition, the Algorithm 11 provides the incremental algorithm for an incoming signal (i.e. $\boldsymbol{O}$). For each incoming sample $\boldsymbol{O}[t]$, a state is constructed with its connections to the forward link. The new defined cluster label p_t and the state's suffix link are initialized as null or zero. Moreover, it has a similar structure as the AO NewAddFrame function (see Algorithm 7), the main difference are the lines 16 to 22, which modify the cluster label and assigns it to q_{t+1}, then appends the pointer

Algorithm 10 VMOConstruction

Require: Time series as $O = O[1]; O[2] \cdots O[\text{T}]$
1: Create an oracle P with initial state p_0
2: sfx_p $[0] \leftarrow$ -1, $\Sigma \leftarrow \emptyset$, M $\leftarrow$ 1
3: **for** $t = 1$ to T **do**
4: *Oracle* $(P = p_1 \cdots p_t)$, M $\leftarrow$ *Add-Frame (Oracle* $(P = p_1 \cdots p_{t-1})$, O[t], M)
5: **return** Oracle$(P = p_1 \cdots p_t)$

of $\boldsymbol{O}[t]$ to σ_{q_t}. Furthermore, another difference is that in these algorithms, the lrs is not used nor defined. On the contrary, the FO and AO do define and use the lrs.

Algorithm 11 AddFrame function: Incremental algorithm for an incoming signal

Require: Oracle $P = p_1 \cdots p_t$, time-series instance O[t+1] and number of cluster M
1: Create a new state $t + 1$
2: $q_{t+1} \leftarrow 0$, $\mathbf{sfx}_p[t + 1] \leftarrow 0$
3: Create a new transition from t to $t + 1$, $\delta(t, q_{t+1}) = t + 1$
4: $k \leftarrow \mathbf{sfx}_p[t]$
5: **while** $k >$ - 1 **do**
6: $\boldsymbol{D} \leftarrow$ distances between O[t+1] and $\boldsymbol{O}[\delta(k, :)]$
7: **if** all distances in $\boldsymbol{D}$ are greater than θ **then**
8: $\delta(k, q_{t+1}) \leftarrow t + 1$,
9: $k \leftarrow \mathbf{sfx}_p[t]$
10: **else**
11: Find the forward link from k that minimizes $\boldsymbol{D}$
12: $k' \leftarrow (k, :)[\text{argmin}(D)]$
13: $\mathbf{sfx}_p[t + 1] \leftarrow$ k'
14: break
15: **if** k = -1 **then** ▷ No suffix exists
16: $\mathbf{sfx}_p[t + 1] = 0$
17: Initialize a new cluster with current frame index $\sigma_{M+1} \leftarrow t + 1$
18: $\Sigma \leftarrow [\Sigma; \sigma_{M+1}]$
19: Assign a label to the new cluster, $q_{t+1} \leftarrow M + 1$
20: Update number of clusters, $M \leftarrow M + 1$
21: **else**
22: Assign cluster label based on assigned suffix link $q_{t+1} \leftarrow q_k$, $\sigma_{q_{k'}} \leftarrow [\sigma_{q_{k'}}; t + 1]$
23: **return** Oracle $P = p_1 \cdots p_{t+1}$, M

Moreover, after the VMO is created, the following step is the improvisation, in this case, it is called Incremental − Improvisation and is shown in the Algorithm 12. This improvisation is different from the FO and AO ones that have been shown, because the VMO works with *Clustering* and

Algorithm 12 Incremental – Improvisation function

Require: A VMO $P = p_1 \cdots p_t \cdots p_T$, in active state i_c.

1: Retrieve all the following states of the states having the same label q_{i_c} as p_{i_c} in a list L
2: After a specific number of states, select a random state j and assign it to i_n, $i_n \leftarrow j$;
3: **if** L *is empty* **then then**
4: $\quad i_n \leftarrow i_c + 1$;
5: **else**
6: $\quad i_n \leftarrow$ Choose a random symbol from L either uniformly or by some specified weights
7: **return** i_n

$Clusters$. Clustering is a vector of clusters, and a Cluster is a structure that has a label and a vector of following states, this means that inside cluster vector, there are pointers to the audio factors with more similarity.

One thing that we noticed, was that the Incremental Improvisation we saw in the papers had a missing step. The previous algorithm had the **if** and **else** statement, and as the list L retrieved all the $following$ states, the algorithm would always go forward and the improvisation would end suddenly. Evidently, we created the step 2 of the algorithm, which after a specific number of states, depending on the length of the Clustering, would randomly choose a state j and assign it as i_n. This addition made possible a longer improvisation time and a more fluid one.

In this particular algorithm, given a starting state i_c, a pointer goes through the oracle structure (VMO) by navigating suffix links to reorganize the audio stream. Furthermore, the smoothness of the new audio is guaranteed because the suffix links point to the occurrences of the longest repeated subsequences of the active state.

In the next section, we will describe the implementation process of the Variable Markov Oracle.

5.2 Implementation

In the previous chapter, we created the audio improvisations with the Audio Oracle algorithm using the Gist C++ feature extraction library. Moreover, for the Variable Markov Oracle we extracted the same features that were extracted in the Audio Oracle, these features are: Spectral Centroid, Spectral Rolloff, Zero Crossings, Root Mean Square, MFCC and Pitch.

In Figure 5.1, we see the original Drivers License Instrumental audio, this is the audio used for most of the improvisations but we started using other audios to create different improvisations. Furthermore, we will see different improvisations created with the Variable Markov Oracle Algorithm.

The first improvisation we created was extracting the Spectral Centroid, this improvisation can be seen in Figure 5.2, one of the first details we see is the feature threshold, the feature threshold used in the Variable Markov Oracle has a widely different value than the one used in the Audio Oracle, this happens because the VMO and the AO are represented in different structures. The AO uses an

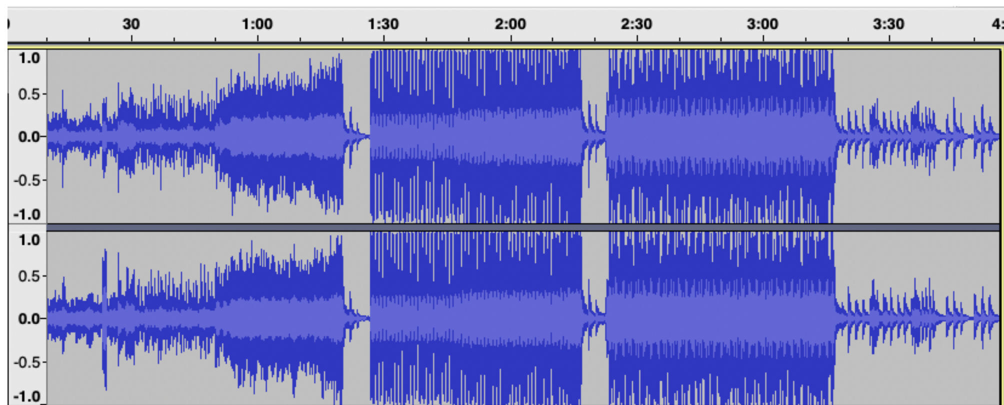


Figure 5.1: Drivers License Original Instrumental Audio (`https://bit.ly/3sgCG3Q`)

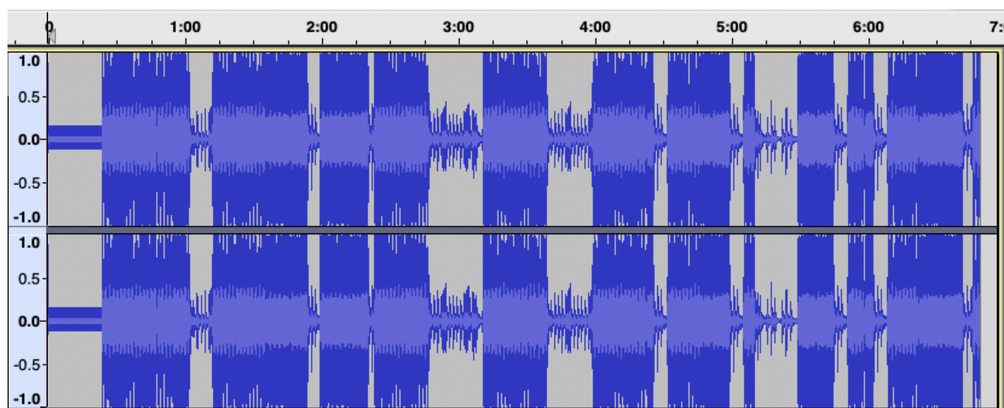


Figure 5.2: Drivers License Improvisation (`https://bit.ly/3yOBzuB`), starting state is 1, feature threshold=0.005, length=1200, hop size=16384, feature=Spectral Centroid

algorithm to find the Longest repeated subsequence, while the VMO uses clusters and a different validation to create the improvisation. For instance, in the Audio Oracle we can see thresholds like 0.08 and 0.1, meanwhile in the Variable Markov Oracle we can see thresholds, such as, 0.0005 and 0.005.

Moreover, if we compare the improvisations of Figure 5.2 and Figure 5.3, there are two main differences: the feature threshold, which are 0.005 and 0.0005 respectively, and the first improvisation starts in state 1, unlike the other which starts in state 2. These two differences can make these improvisations change a lot, we see that if it starts in state 1, as in Figure 5.2, it stays in a loop for a while at the beginning. While, in Figure 5.3, there is no loop at the beginning.

Similarly, in Figure 5.4 and Figure 5.5, the only difference is the feature threshold, one of them is 0.0005 and the other is 0.005, this means that even if they start in the same state, their improvisations are not very similar. The fact that they start in different states, can change flow of the

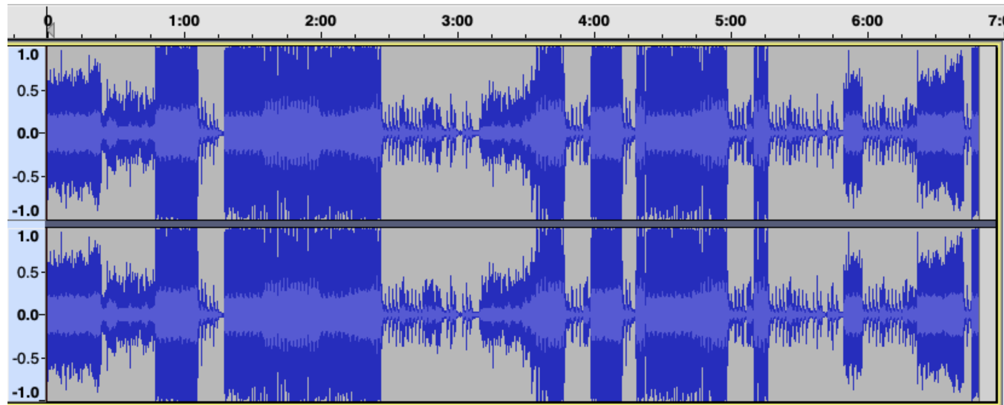


Figure 5.3: Drivers License Improvisation (`https://bit.ly/3xPPBL3`), starting state is 2, feature threshold=0.0005, length=1200, hop size=16384, feature=Spectral Centroid

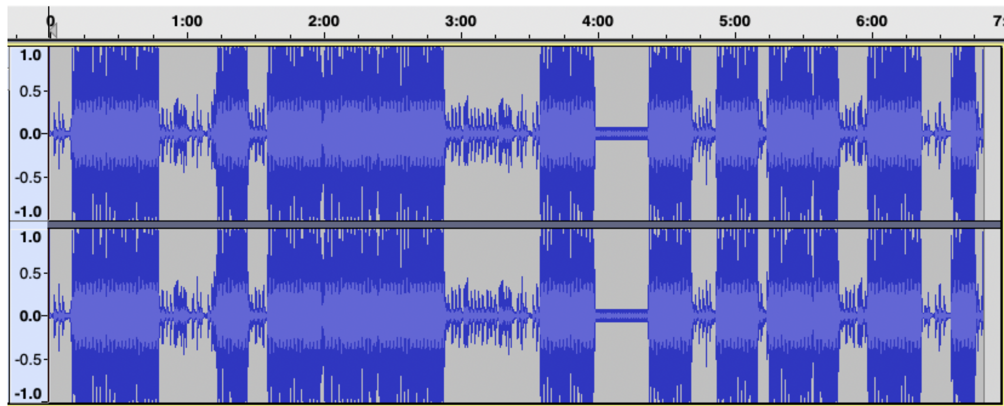


Figure 5.4: Drivers License Improvisation (`https://bit.ly/3AG9Sof`), starting state is 1, feature threshold=0.0005, length=1200, hop size=16384, feature=Root Mean Square

improvisation algorithm and lead to a very different audio improvisation.

For instance, in Figure 5.6 the feature we are extracting is the MFCC. A difference from the other improvisations, is that the feature threshold is bigger, in this case, the feature threshold is 0.02, while for the others examples we have showed is 0.0005 or 0.005. If we used a smaller threshold for the MFCC, the VMO would think most of the audio frames were not similar and the VMO would not work as well.

Additionally, in Figure 5.7 and Figure 5.8, the feature threshold changes from 0.0005 to 0.005. In addition, the first one starts in state 1 and the second one starts in state 8. We can see that these improvisations have similarities because the extracted feature is the same, but they are not exactly the same.

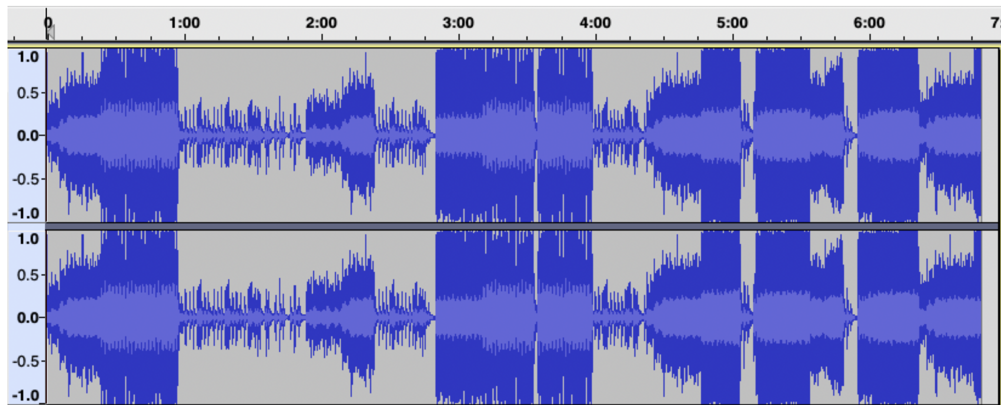


Figure 5.5: Drivers License Improvisation (`https://bit.ly/2VSpLZC`), starting state is 1, feature threshold=0.005, length=1200, hop size=16384, feature=Root Mean Square

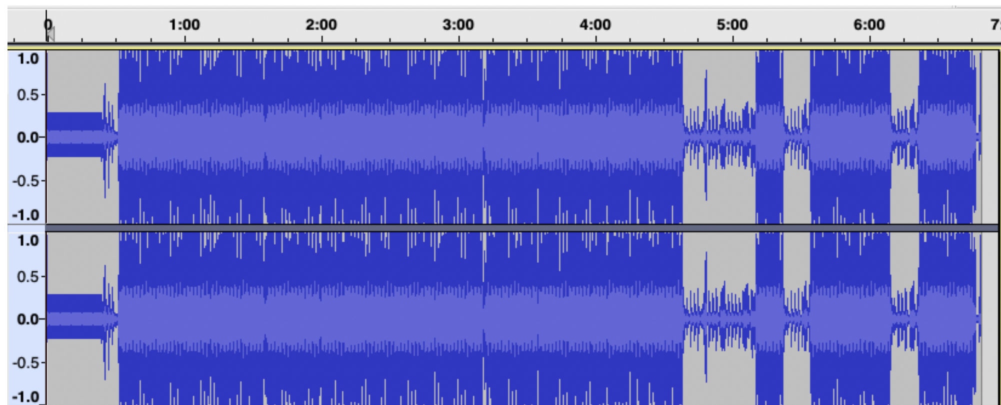


Figure 5.6: Drivers License Improvisation (`https://bit.ly/3yZpFOR`), starting state is 1, feature threshold=0.02, length=1200, hop size=16384, feature=MFCC

After showing the improvisations created with the VMO, we shall describe the integration of the VMO with `ossia-score`.

5.3 Variable Markov Oracle in `ossia-score`

Now, after doing the Variable Markov Oracle implementation and creating the audio improvisations, we needed to integrate the Variable Markov Oracle with `ossia-score`.

In the same way as the Audio Oracle, the Variable Markov Oracle was decided to be integrated as an *addon* because of the same reasons, the users were going to value more the creation of an audio improvisation from an existing audio file or in this case, a sound file. Equally, the final changes of the Variable Markov Oracle are here (`https://bit.ly/3iGIcJZ`), we created a pull request to

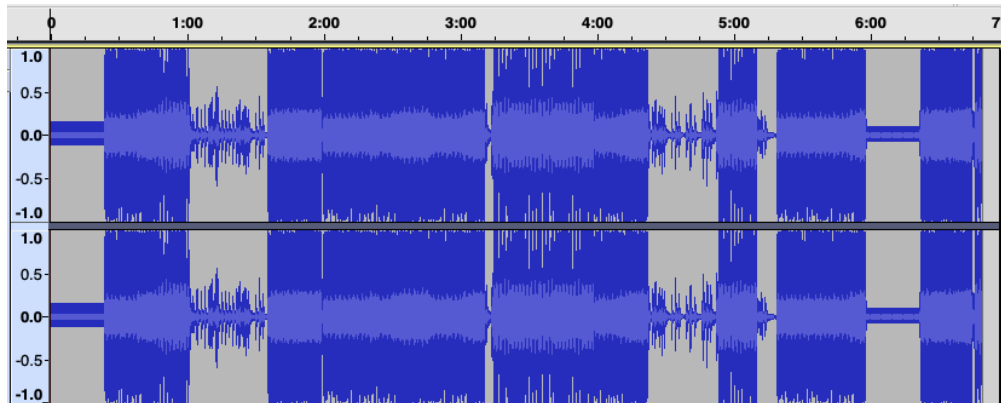


Figure 5.7: Drivers License Improvisation (`https://bit.ly/37Grggd`), starting state is 1, feature threshold=0.0005, length=1200, hop size=16384, feature=Zero Crossings

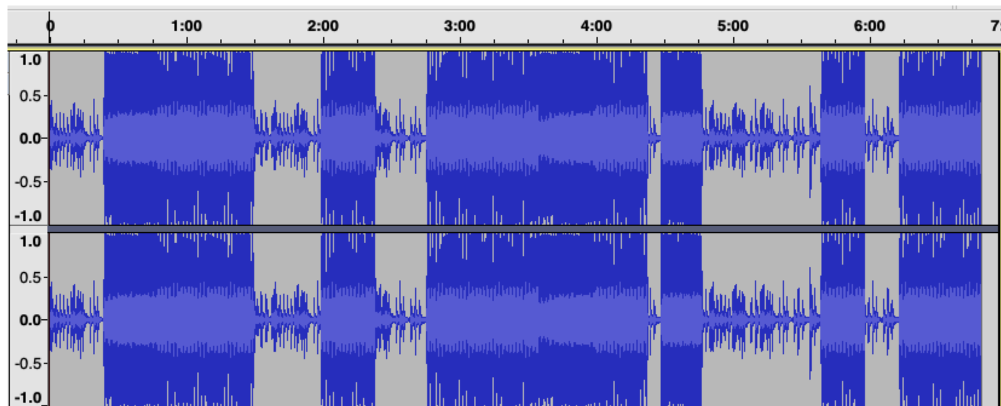


Figure 5.8: Drivers License Improvisation (`https://bit.ly/2XhUuQA`), starting state is 8, feature threshold=0.005, length=1200, hop size=16384, feature=Zero Crossings

integrate it with `ossia-score`.

The process to create an improvisation in `ossia-score` using the VMO *addon* starts by clicking on the sound file and then going to the top right side of `ossia-score` to see the sound file, name and attribute. This can be seen in Figure 5.9. Then, we need to right click the sound file name at the top right, go down to `Offline processing` and select `Variable Markov Oracle`, as shown in Figure 5.10.

Then, as seen in Figure 5.11, if we click on the Variable Markov Oracle option, we will start seeing a new pop up that says `Processing... In Progress`. This lets the user know the Variable Markov Oracle process is in progress and when the pop up disappears the processing will be done, this pop up was also implemented for the Audio Oracle.

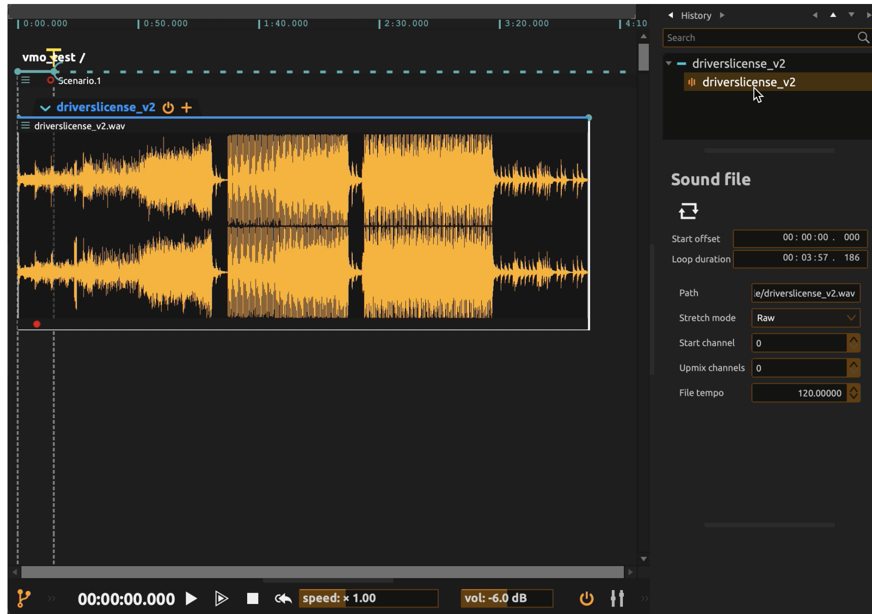


Figure 5.9: Drivers License Audio (https://bit.ly/3x0vjSe) in ossia-score

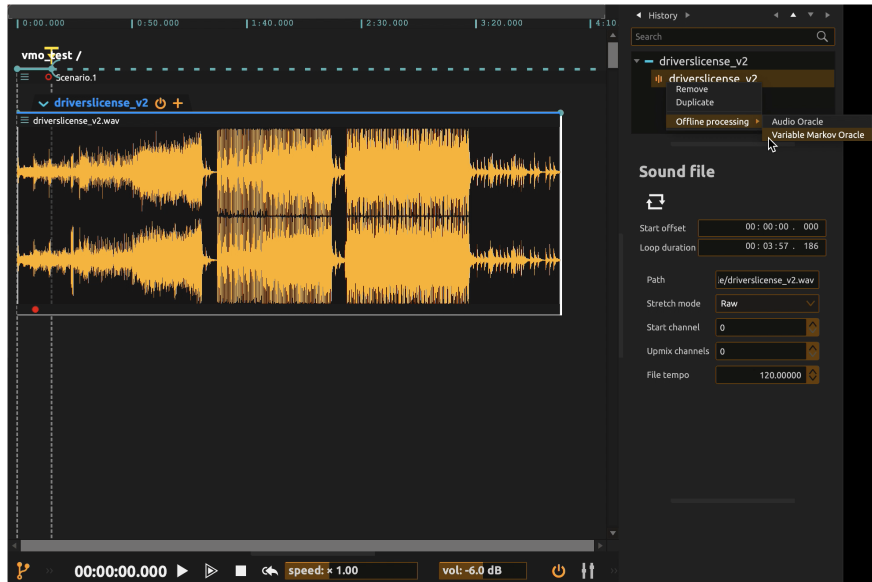


Figure 5.10: Drivers License Audio (https://bit.ly/3x0vjSe) in ossia-score, hover over Variable Markov Oracle option

Afterwards, in Figure 5.12, we see the audio improvisation after using the Variable Markov Oracle *addon* in ossia-score, and in Figure 5.13, we see the audio improvisation without the whole ossia-score application.

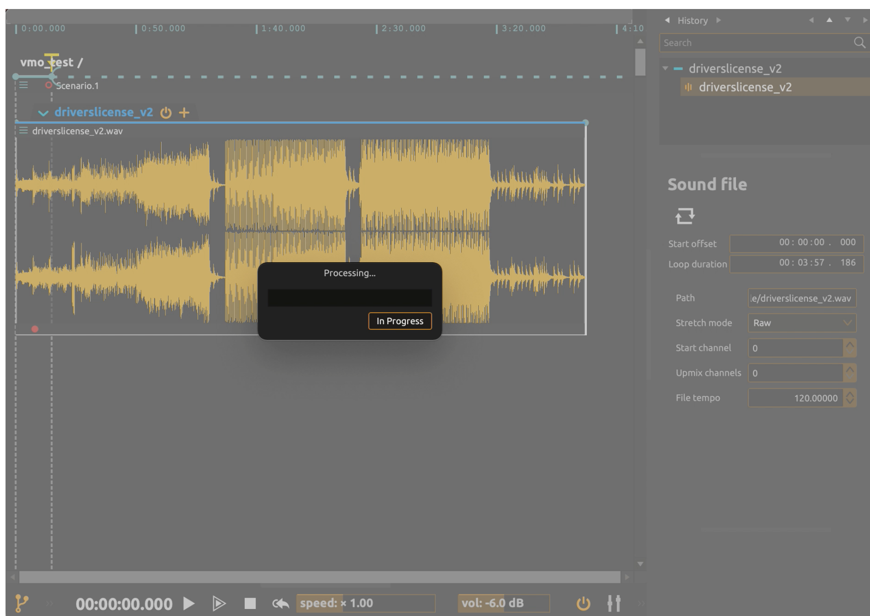


Figure 5.11: Drivers License Audio (https://bit.ly/3x0vjSe) in ossia-score, Processing pop up

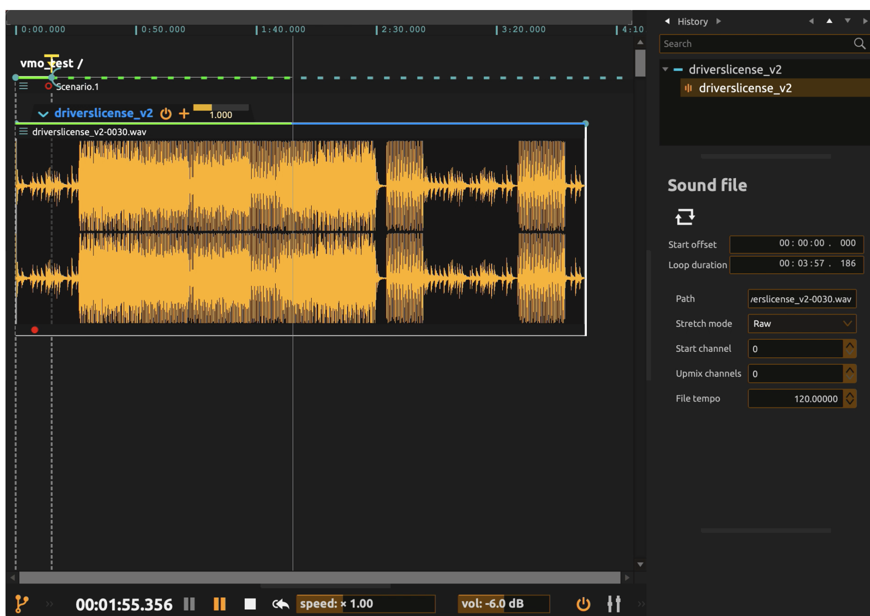


Figure 5.12: Drivers License Improvisation (https://bit.ly/3x0vjSe) in ossia-score using VMO

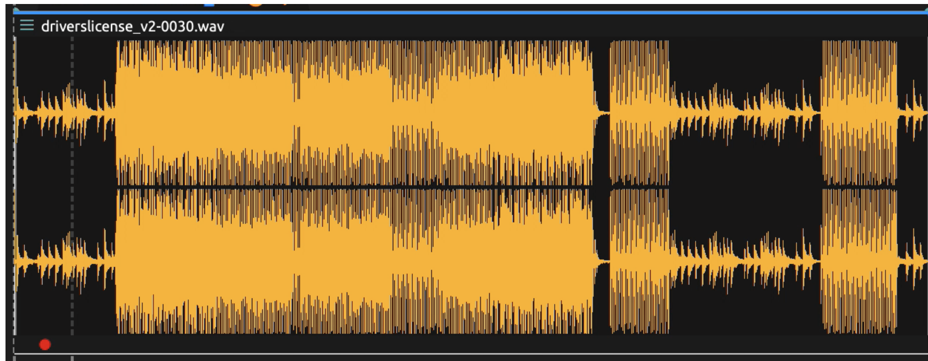


Figure 5.13: Drivers License Improvisation (`https://bit.ly/3x0vjSe`) in `ossia-score`

Like we showed in the previous chapter, after analyzing what would be better for the users, we decided to implement one *addon* for each extracted feature. The names would have the format of 'Audio Oracle - feature extracted' or 'Variable Markov Oracle - feature extracted', such as, 'Audio Oracle - Spectral Rolloff'. This was implemented for the Audio Oracle and Variable Markov Oracle, as shown in Figure 5.14.

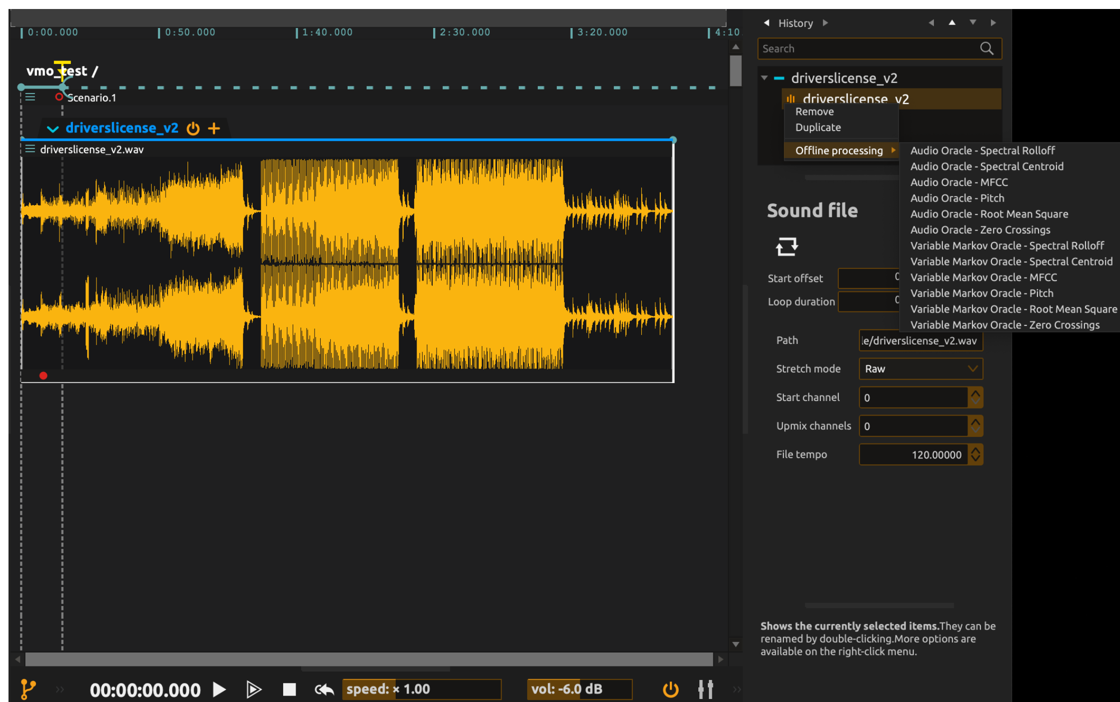


Figure 5.14: All Audio Oracle and Variable Markov Oracle *addons* in `ossia-score`

Finally, after creating all the other *addons* for the Variable Markov Oracle, we show from Figures 5.15 to 5.19, all the audio improvisations after using these implemented *addons*.

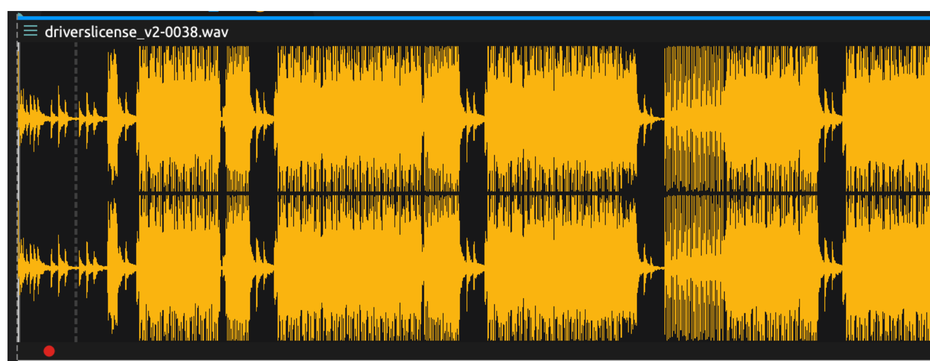


Figure 5.15: Drivers License Improvisation in ossia-score (https://bit.ly/3yL5iVl), feature threshold = 0.08, length equal to the original audio, hop size=32768, feature=Spectral Centroid

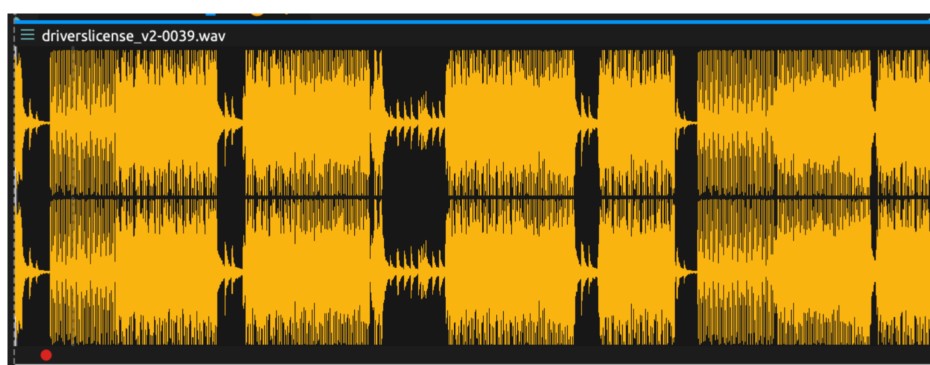


Figure 5.16: Drivers License Improvisation in ossia-score (https://bit.ly/3fXv1m7), feature threshold = 0.08, length equal to the original audio, hop size is 32768, feature is MFCC

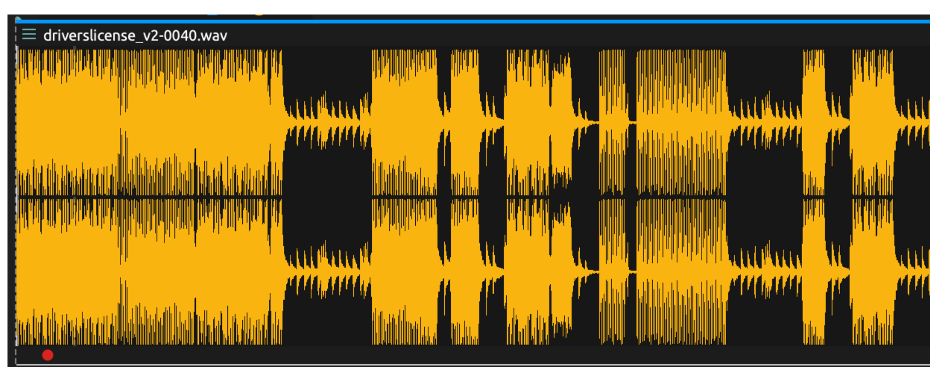


Figure 5.17: Drivers License Improvisation in ossia-score (https://bit.ly/2UhMgab), feature threshold = 0.08, length equal to the original audio, hop size is 32768, feature is Pitch

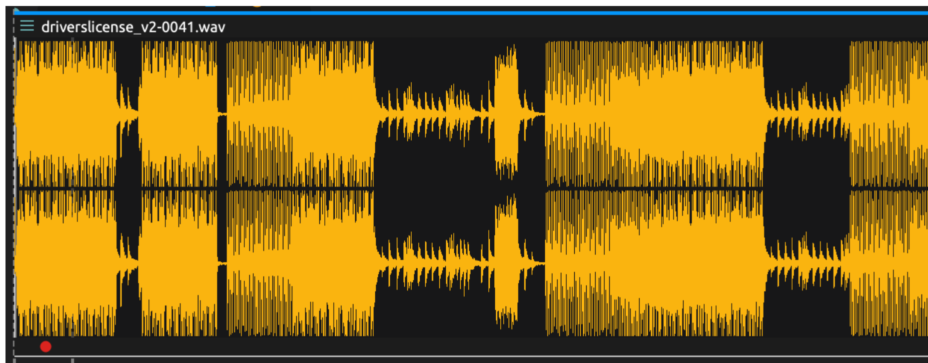


Figure 5.18: Drivers License Improvisation in ossia-score (https://bit.ly/3xMf746), feature threshold = 0.08, length equal to the original audio, hop size is 32768, feature is Root Mean Square

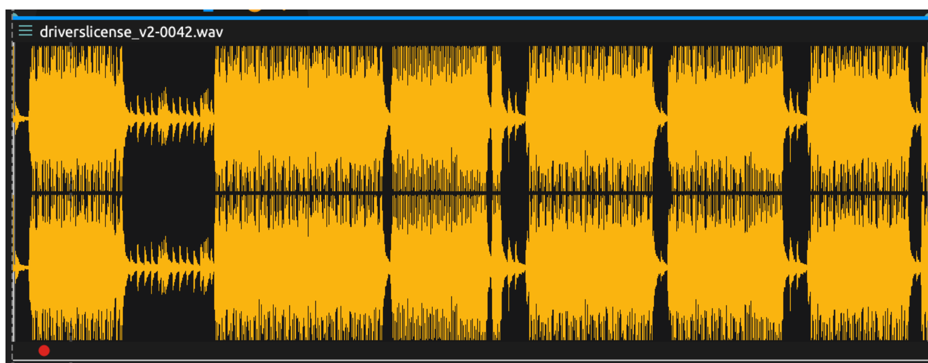


Figure 5.19: Drivers License Improvisation in ossia-score (https://bit.ly/3CJR2OZ), feature threshold = 0.08, length equal to the original audio, hop size is 32768, feature is Zero Crossings

After this section, in Section 5.4, we will show the graphs that represent the structure used to create the VMO. As we mentioned, this structure is a Clustering.

5.4 Graph Creation

At last, following the Variable Markov Oracle implementation and the integration with ossia-score, the final part of the Variable Markov Oracle was creating the graphs to illustrate how the Clustering works, because it is different than the automaton creation done in the previous chapter, since they are two different structures.

The graph creation of the Variable Markov Oracle can seem extremely similar to the Audio Oracle automaton creation, but there are key differences, these differences are: in the Audio Oracle automaton there is the feature value in the forward transitions from state i to state $i + 1$, which is a number that indicates the feature extracted from the audio frame, meanwhile, in the Variable Markov Oracle graph the number indicating the featured extracted value is in each state.

Another difference, which is bigger, is that we are not using the suffix links nor the forward transitions that we used in the Audio Oracle, the transitions created in these graphs are generated using the Clustering that was introduced in this algorithm. The Clustering has all the states, each state has a *label* and a Cluster, which is a vector pointing to its similar following states, these following states are represented in the graph as the following transitions and the label represents the suffix links. This is why, we can see in some states that there is an arrow pointing upwards, this means that the Clustering's label in that state is itself.

Moreover, one thing to point out is that these graphs are smaller than the automata in the Audio Oracle chapter, this occurs because these new graphs were created using the hop size of 524,288. The graph creation is generated using the `TikZ` latex package [1] and a dimension too large is a problem sometimes, so the solution to the problem was making the graph smaller.

Furthermore, in Figures 5.20 and 5.21, we see that the two graphs refer to the Variable Markov Oracle, extracting the Spectral Centroid feature, the difference is that in the former the feature threshold is 0.02, and in the latter the feature threshold is 0.05. The purpose of pointing out this one difference is that the Clusterings created, which are represented in these figures, changes a lot because of it, meaning that the feature threshold is really important.

We see that in Figure 5.20, there are a lot of forward transitions from state 1. While, in Figure 5.21, there are more forward transitions from state 6.

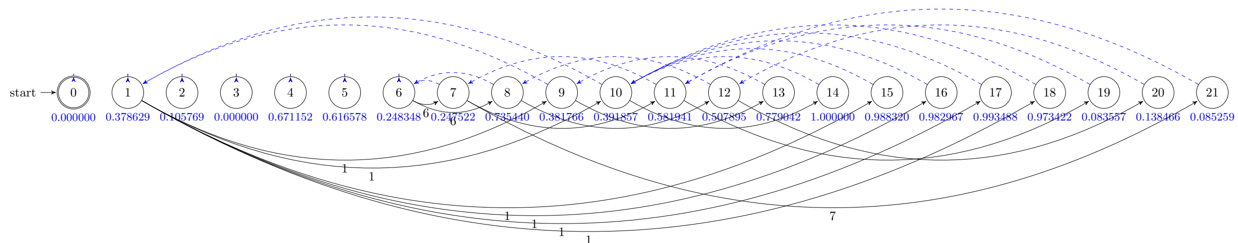


Figure 5.20: Variable Markov Oracle graph, feature extracted is Spectral Centroid, feature threshold = 0.02

Additionally, we give a similar example in Figures 5.22 and 5.23. In these two graphs the feature extracted was the Root Mean Square. In the former, the feature threshold is 0.02, and in the latter, it is 0.05. The graph in the first figure until the sixth cluster, has only two following states from cluster number 1, which are the cluster number 20 and 21, and two backward transitions go to cluster number 1 from the cluster number 20 and 21. Meanwhile, in the second figure, even though, these following and backward transitions are the same in the cluster number 1, there are way more forward and backward transitions from the clusters one to six than in the previous figure. This

[1] Overleaf. TikZ package. https://www.overleaf.com/learn/latex/TikZ_package

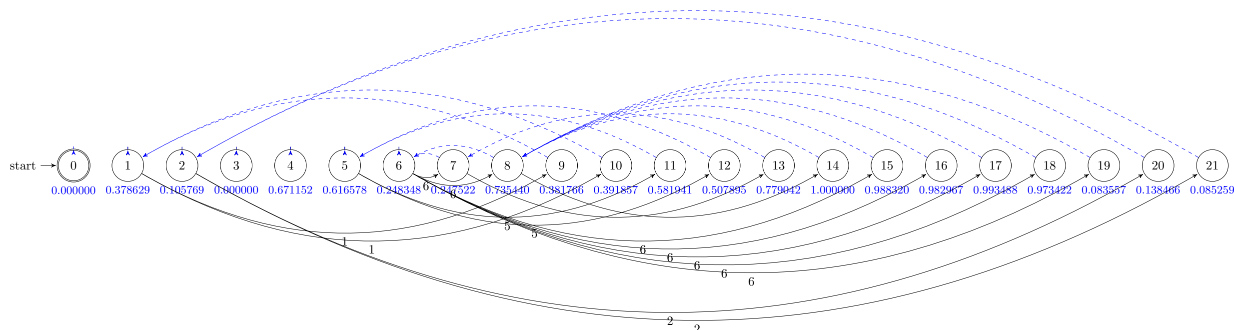


Figure 5.21: Variable Markov Oracle graph, feature extracted is Spectral Centroid, feature threshold = 0.05

means, that changing slightly the feature threshold, can change the Clustering transitions.

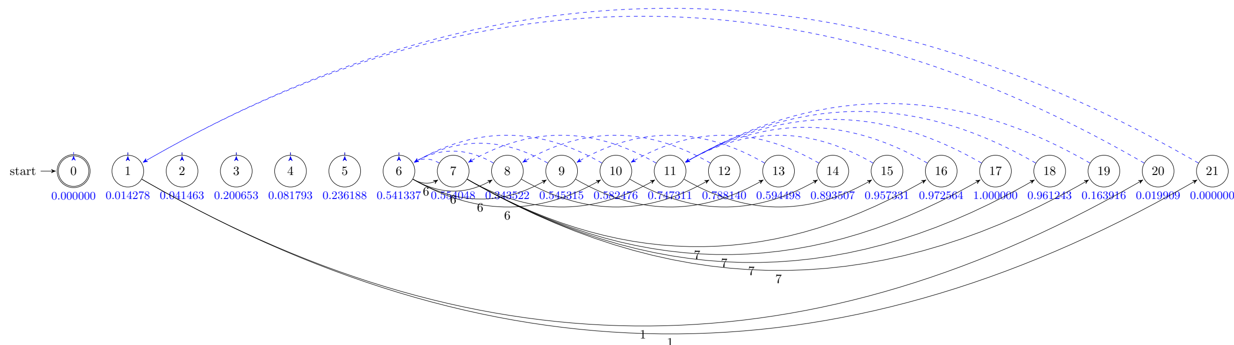


Figure 5.22: Variable Markov Oracle graph, feature extracted is Root Mean Square, feature threshold = 0.02

Similarly, in Figures 5.24 to 5.27, we have the same comparison, our point in doing these comparisons is showing the graph differences when changing the feature threshold. Sometimes, depending on the feature that is being extracted, the graph changes a lot, but in other cases, the graph does not change much.

For example, Figures 5.26 and 5.27 have a more noticeable change, than in Figures 5.24 and 5.25. In the last two figures, we need to look closely to see that some forward and backward transitions are not the same in both figures, even though, in plain sight they can look exactly the same.

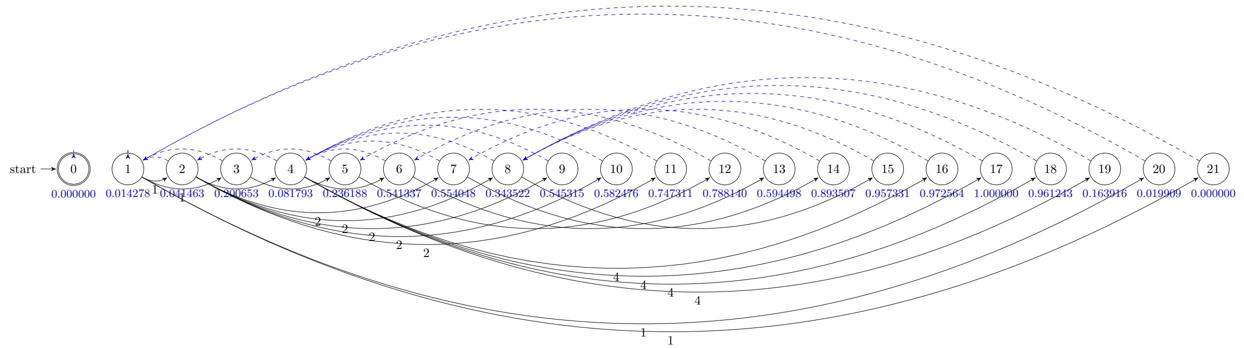


Figure 5.23: Variable Markov Oracle graph, feature extracted is Root Mean Square, feature threshold = 0.05

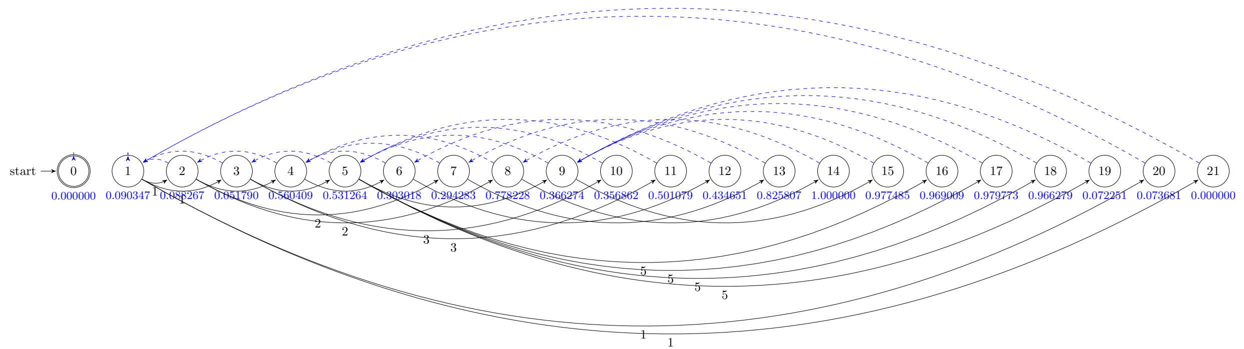


Figure 5.24: Variable Markov Oracle graph, feature extracted is Spectral Rolloff, feature threshold = 0.02

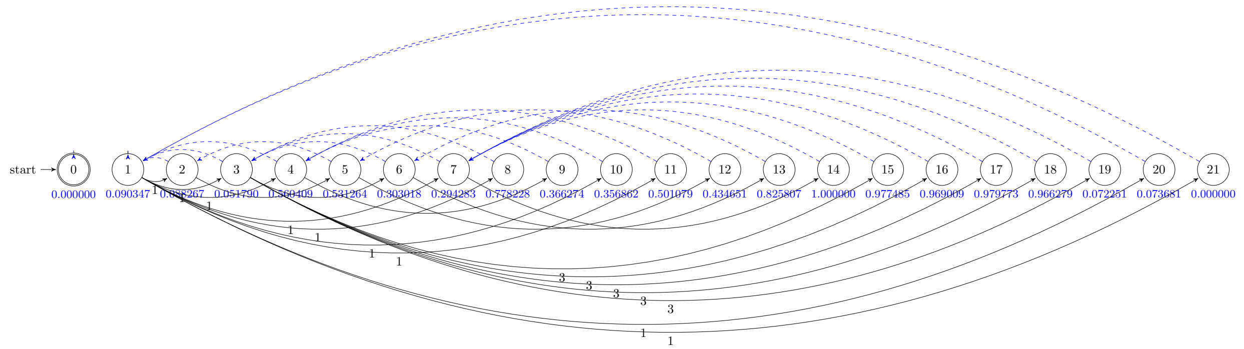


Figure 5.25: Variable Markov Oracle graph, feature extracted is Spectral Rolloff, feature threshold = 0.05

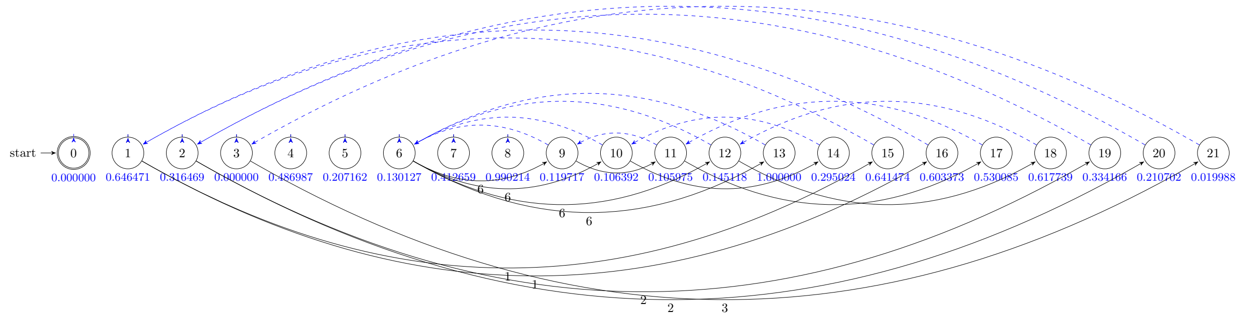


Figure 5.26: Variable Markov Oracle graph, feature extracted is Zero Crossings, feature threshold = 0.02

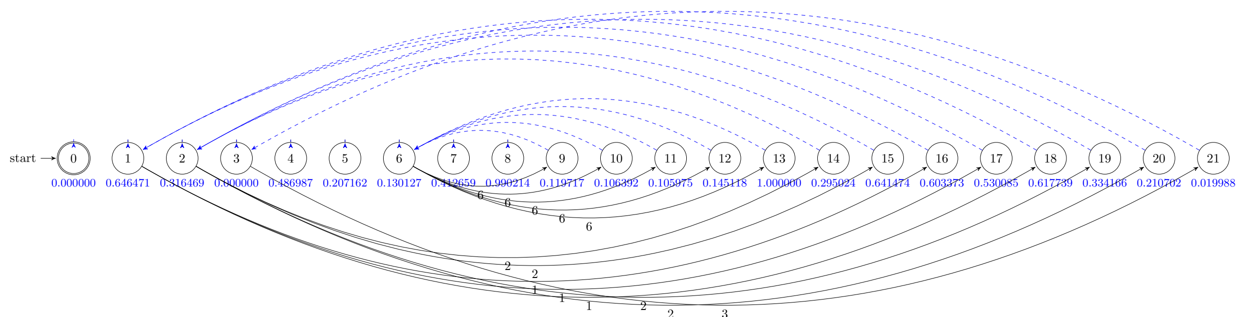


Figure 5.27: Variable Markov Oracle graph, feature extracted is Zero Crossings, feature threshold = 0.05

Chapter 6

Concluding Remarks

We conclude this document by stating the main results derived from this degree project and we also identify some directions for future work.

6.1 Overview

In this project we studied the Factor Oracle, Audio Oracle and Variable Markov Oracle algorithm. We implemented these three algorithms in C++, and integrated them with the `ossia-score` application.

We first started with the implementation of the Factor Oracle algorithm in C++, because this algorithm is the base of the other two, meaning that this implementation had to be done carefully. After it was being implemented in C++, the next step was to integrate it with the `ossia-score` as a plug-in. We created two plug-ins, one for strings and another one for MIDI input. The first one was called `New Factor Oracle` and the second one was called `New Factor Oracle MIDI`. We tested that the implementation and integration were done correctly using different inputs.

We then implemented the Audio Oracle algorithm in C++. This implementation consisted in finding the C++ feature extraction library and integrating it with the Audio Oracle algorithm to extract different audio features. Moreover, after it was completed, we did the integration with `ossia-score` as an *addon*, which differed from the Factor Oracle integration in `ossia-score`, the main reason for the Audio Oracle to be an *addon* was to give the users a new feature to create improvisations with sound files.

Afterwards, we implemented the Variable Markov Oracle algorithm in C++, as we had already chosen the C++ feature extraction library, we needed to create the Clustering structure, which is an extension from the previous algorithm. Furthermore, we completed the implementation in C++ and proceeded to integrate with `ossia-score` as an *addon*.

Finally, we created one *addon* for each feature extraction for the Audio Oracle and Variable Markov Oracle. This means that we created exactly ten *addons*.

6.2 Future Work

The following are some interesting directions for future work:

User input flexibility In Chapters 4 and 5, we showed how the Audio Oracle and Variable Markov Oracle *addons* were implemented. One aspect that could be missing to make the user experience better, is the ability for the user to input exactly what feature threshold value they want from 0 to 1, what hop size they want and for the Audio Oracle, which q is better for their desired improvisation. This could be implemented by talking to the `ossia-score` developers to create a pop up with the inputs or find a way to integrate it more easily.

Live audio improvisation Like we saw in chapter Chapters 4 and 5, the audio improvisations were going to be implemented for offline audio, which means the user needs to input a sound file into `ossia-score` to create an audio improvisation. An interesting approach for a wider audience, would be to implement the Audio Oracle and Variable Markov Oracle for live improvisation, so the musician can start recording live and the algorithm would start creating the automaton (for Audio Oracle) or clustering (for Variable Markov Oracle).

Find the optimal feature threshold automatically For the Audio Oracle and Variable Markov Oracle, it would be an improvement for the users to not need to input the feature threshold manually. The optimal feature threshold would be found using Information Rate (IR) algorithms, this way, the feature threshold would be found automatically when the audio file is input.

Improvisation algorithm improvement using PyOracle heuristics Moreover, PyOracle heuristics could be used to improve the improvisation algorithm, which would help to create better improvisations. This would refine the transitions between audio factors and consequently, it would improve the complete audio improvisations.

Bibliography

[AD04] Gérard Assayag and Shlomo Dubnov. Using factor oracles for machine improvisation. *Soft Computing*, 8:604–610, 2004.

[ADCD17] Jaime Arias, Myriam Desainte-Catherine, and Shlomo Dubnov. Interactive machine improvisation scenarios, 2017. [Online]. Available: https://vmo-score.github.io/.

[CC21] Ching-Hua Chuan and Elaine Chew. Audio onset detection using machine learning techniques: the effect and applicability of key and tempo information. 08 2021.

[Cel] Jean-Michaël Celerier. Ossia open software system for interactive applications. [Online]. Available: https://ossia.io/about.html. Accessed 2021-01-10.

[CR00] Jean-Michaël Celerier and Julien Rabin. ossia score. [Online]. Available: https://github.com/ossia/score, 2000. Accessed 2021-02-03.

[DAC07] Shlomo Dubnov, Gérard Assayag, and Arshia Cont. Audio oracle: a new algorithm for fast learning of audio structures. In *Proceedings of the 2007 International Computer Music Conference, ICMC 2007, Copenhagen, Denmark, August 27-31, 2007*. Michigan Publishing, 2007.

[DAC11] Shlomo Dubnov, Gérard Assayag, and Arshia Cont. Audio oracle analysis of musical information rate. In *Proceedings of the 5th IEEE International Conference on Semantic Computing (ICSC 2011), Palo Alto, CA, USA, September 18-21, 2011*, pages 567–571. IEEE Computer Society, 2011.

[GP14] Theodoros Giannakopoulos and Aggelos Pikrakis. *Introduction to audio analysis: a MATLAB approach*. Academic Press, 2014.

[Leb17] Jakob Leben. Marsyas, music analysis, retrieval and synthesis for audio signals, 2017. [Online]. Available: http://marsyas.info.

[Ler12] Alexander Lerch. *An Introduction to Audio Content Analysis: Applications in Signal Processing and Music Informatics*. Wiley-IEEE Press, 1st edition, 2012.

[LLA03] Arnaud Lefebvre, Thierry Lecroq, and Joël Alexandre. An improved algorithm for finding longest repeats with a modified factor oracle. *Journal of Automata, Languages and Combinatorics*, 8:647–657, 01 2003.

[Man11] V. J. Manzo. *Max/MSP/Jitter for music: a practical guide to developing interactive music systems for education and more*. Oxford University Press, 2011.

[MLA+16] Steve McLean, Luko, Alexis, McIntosh, and Andrew. Pitch, Sep 2016.

[NCA17] Jérôme Nika, Marc Chemillier, and Gérard Assayag. Improtek: Introducing scenarios into human-computer music improvisation. *Comput. Entertain.*, 14(2), January 2017.

[SD13] Greg Surges and Shlomo Dubnov. Feature selection and composition using pyoracle. In Philippe Pasquier, Arne Eigenfeldt, Oliver Bown, and Graeme McCaig, editors, *Musical Metacreation, Papers from the 2013 AIIDE Workshop, October 14-15, 2013, Boston, Massachusetts, USA*, volume WS-13-22 of *AAAI Workshops*. AAAI Press, 2013.

[Sta17a] Adam Stark. Audiofile - a c++ library for reading and writing audio files, 2017. [Online]. Available: `https://github.com/adamstark/AudioFile`.

[Sta17b] Adam Stark. Gist - an audio analysis library, 2017. [Online]. Available: `https://github.com/adamstark/Gist`.

[Sur20] Priy Surya. Clustering in machine learning, Feb 2020.

[vV13] Merlijn van Veen. How window size determines frequency resolution, Jul 2013. [Online]. Available: `https://www.merlijnvanveen.nl/en/study-hall/21-how-window-size-determines-frequency-resolution`.

[Wal20] Leigh Walker. Audio spectrum, Apr 2020. [Online]. Available: `https://www.teachmeaudio.com/mixing/techniques/audio-spectrum`.

[WD15] C. Wang and S. Dubnov. The variable markov oracle: Algorithms for human gesture applications. *IEEE MultiMedia*, 22(4):52–67, 2015.

[WHD15] Cheng-i Wang, Jennifer Hsu, and Shlomo Dubnov. Music pattern discovery with variable markov oracle: A unified approach to symbolic and audio representations. In Meinard Müller and Frans Wiering, editors, *Proceedings of the 16th International Society for Music Information Retrieval Conference, ISMIR 2015, Málaga, Spain, October 26-30, 2015*, pages 176–182, 2015.

[WHD16] Cheng-I Wang, Jennifer Hsu, and Shlomo Dubnov. Machine improvisation with variable markov oracle: Toward guided and structured improvisation. *Comput. Entertain.*, 14(3):4:1–4:18, 2016.