Alma Mater Studiorum · Università di Bologna

FACULTY OF SCIENCE
Master in Computer Science

Panson:
An interactive sonification framework for real-time applications

Supervisors:
Prof. Simone Martini
Dr. Thomas Hermann
Dr. David Johnson

Author:
Michele Nalli

Session I
Academic Year 2021/2022

To my family,
who have always believed in me

Abstract

Many sonification systems face a number of common design challenges. These are addressed in every project with different, specific-purpose solutions. We present Panson – an **interactive sonification framework** implemented in Python that can ease the development of sonification systems. Panson allows the user to implement sonifications using the sc3nb library [37] as interface to the SuperCollider [11] sound synthesis engine. The framework provides support for both offline and online (real-time) sonification through a set of composable classes; these classes are designed to natively support interaction in Jupyter Notebooks [20]. Using Panson, we will show an example of its application by implementing a facial expression sonification Jupyter Notebook based on OpenFace 2.0 [27].

Introduction

The term "sonification" refers to a group of techniques that allows to experience data through sounds [33]; by doing so, listeners who are able to interpret sounds can explore some aspects of the data in a way that is radically different from plotting methods and can provide important insights. These techniques make sound an important carrier of information, creating many possibilities in system design.

Over the years, many sonification systems with very different aims have been developed: from sonification of body movements to provide auditory biofeedback, to monitoring systems; from sonification of EEG data, to sensory substitution applications. Most of these applications face similar challenges of different nature: from the choice of tools to be used, to the structure of the code implementation. Much of the development effort could be avoided by adopting a common framework, rather than relying on specific-purpose solutions for each of the projects.

In this work, we discuss the requirements, design and implementation of such a framework and explore its possibilities. The main goal of it is to simplify the development of sonification applications while promoting a clean and reusable code-style. A success in development would mean not only having a new tool to speed-up the development of applications, but also making sonification a more accessible field of technical development.

- **Chapter 1** contains an introduction on the main theoretical concepts related to Sonification and Facial Expression Analysis. We describe a number of sonification systems and consider their common aspects to explain the need for a reusable sonification framework.

- **Chapter 2** refines the need for a sonification framework that emerged from chapter 1 into a concrete list of requirements and development goals. Next, it proceeds describing some tools that are relevant for the framework.

- **Chapter 3** dives into the structure of the framework, describing its API and discussing its main implementation choices.

- **Chapter 4** describes a facial feature sonification application implemented using Panson and the different sonifications implemented.
- **Chapter 5** discusses development results of both framework and facial feature sonification application and outlines the possible developments of both projects.

Contents

List of Figures

Chapter 1

Background

This chapter will provide some background regarding sonification and facial expression analysis and review some sonification systems to explain the need for a reusable sonification framework.

1.1 Sonification

Sonification is an interdisciplinary research field that explores the use of non-speech audio to convey information (definition by Kramer et al. [44]). Despite the apparent simplicity of the previous statement, it is not easy to clearly and unambiguously define what sonification is.

A more precise and articulated definition was conceived by Hermann [33]: "A technique that uses data as input, and generates sound signals (eventually in response to optional additional excitation or triggering) may be called sonification, if and only if

1. The sound reflects objective properties or relations in the input data.

2. The transformation is systematic. This means that there is a precise definition provided of how the data (and optional interactions) cause the sound to change.

3. The sonification is reproducible: given the same data and identical interactions (or triggers) the resulting sound has to be structurally identical.

4. The system can intentionally be used with different data, and also be used in repetition with the same data."

This definition specifies strict limits regarding which kind of process can be called sonification, putting emphasis on scientific aspects such as unambiguity of definition, reproducibility and independence from data.

Different kinds of sonification techniques have been conceived [34], but the only one that is relevant to this thesis is called Parameter Mapping Sonification.

1.1.1 Parameter Mapping Sonification

Parameter Mapping Sonification (PMSon) [35] is based on the idea of establishing a relation (or mapping) between data values and auditory parameters of the output sound. A metaphorical way to think about PMSon is seeing the data as a musical score to be played by the sonification (the element that interprets the score); the instruments played by the sonification are the synthesizers, intended as algorithms that output audio signals.

This approach is very general and flexible: depending on the mapping, it is possible to work with arbitrary data and synthesizers. Mappings from data features to synthesizers' parameters can be of all kinds: one-to-one, many-to-one, one-to-many and many-to-many. Mapping functions adopted can vary depending on the properties of the data, parameters of the synthesizers and desired effect on perception.

As synthesizers can have a number of parameters, PMSon is suitable for navigating multivariate data in a way that would not be possible by using visual plotting methods. For this reason, PMSon is sometimes metaphorically referred to as "auditory scatter plot", referring to the fact that often scatter plots convey information not only through coordinates, but also through other properties such as color and size of the points.

Sounds can be generated using different approaches:

- Continuous: each data point is used to modulate parameters of a number of running synthesizers.
- Discrete: each data point is translated into an independent and usually short-lived sound.
- Event-based: a sound is triggered when an event occurs (a data point satisfies certain conditions).

These different approaches can coexist in a same sonification.

The great flexibility in defining mappings and sound generation methods make PMSon suitable for all kinds of sonification tasks. For instance, it can be used to audibly navigate datasets in search of patterns or to produce

auditory "summaries" of the data; in real-time scenarios, it can be used for monitoring or for building sensory substitution or sensory augmentation systems.

1.2 Behavioural Cues and Social Signals

Communication between humans is made up of a multitude of **social signals**, which reflect people's attitude towards social interaction [60]. Some examples of social signals are attention, empathy, politeness, flirting, agreement, aggressivity and attractiveness. Social signals are expressed through a sequence of non-verbal **behavioural cues**, such as facial expressions, gestures, body posture, vocal behaviour (here we are not referring to the verbal part) as well as physical characteristics (e.g. height). Behavioural cues are simpler elements than social signals because, unlike social signals, they are objectively-measurable changes in physiological activity and do not have any interpretation associated. **Social signal processing** (SSP) is a research domain that aims at providing computers with the ability to perceive and understand social signals [60]. Despite being a very promising field of research, it is still relatively young, which implies that it is difficult to develop SSP systems.

Between behavioural cues related to the head, facial expressions require further explanation and categorization. The **facial action coding system** (FACS) [47] is a widely used method for describing facial expressions. FACS defines 44 **action units** (AUs) that represent independent movements of facial muscles, but also of the head, eyes and other miscellaneous actions. Complex facial expressions can be described with a set of AUs that can be interpreted to obtain information on basic emotions displayed, personality traits or social signals. The facial feature sonification application described in chapter 4 will consider and sonify facial AUs, leaving their eventual interpretation to listeners; non-facial AUs will not be considered: instead, the system will rely on head rotation information and gaze angles.

1.3 State of the Art

Many sonification systems with different aims have been designed. The analysis of some of them will make clear what are the common technical challenges faced during their design. Our main interest is in systems that are capable of real-time performance.

1.3.1 Sonification Systems

1.3.1.1 Physioson

Physioson is a system for interactive sonification of body movements designed to be an assistive tool for physiotherapy treatments [46]. It includes a wearable sensor system that can assess the body movements of the user and a software component written in python that processes sensor data and provides auditory feedback. Users can use their own auditory bio-feedback to adjust their form during exercise execution.

Physioson provides support for recording sensor measurements of the correct execution of the movement under the supervision of an expert, along with a video; recorded reference data will be needed to sonify exercise executions and guide the user to the correct execution of exercises. Sensor recordings (along with their videos) can be replayed using different sonifications.

When reference data for a certain exercise is available, so called "relative" sonifications can be used. These sonifications consider both reference data and data obtained in real-time from sensors and are meant to provide information on the difference between real-time execution and recorded correct execution of the exercise. "Absolute" sonifications are used when reference data is not available. Sonification definitions can be added and removed from the system, but the system is shipped with a library of implemented sonifications. Sonification have parameters that can be regulated through sliders displayed on the main application window.

1.3.1.2 GymSon

GymSon is a sonification system designed to provide realtime auditory feedback regarding a particular gymnastic movement: the switch leap performed on the balance beam. This system was used in the context of a study regarding the possibilities of improving the execution of complex movements using auditory feedback [31].

GymSon sonifies the angle between the thighs of gymnasts, extracted in realtime from data coming from sensors mounted on thighs and waist. The purpose of the sonification is to provide a feedback on how good the execution of the movement is, given that, in a good performance, the measured angle should be at least 180°.

1.3.1.3 Head Movements Sonification

Hermann et al. describe a sensory substitution system for realtime interactive sonification of head movements and gestures [36]. Such a system

can have many applications: for example it could be used to support visually impaired people in their interactions. Users could wear the sensor system to sonify their own head movements and gestures; this way they could learn how auditory feedback relates with their proprioception. If the sensor system is worn by an interaction partner, their head gestures could be perceived in realtime by a visually impaired user.

1.3.1.4 Swimming Sonification

Hermann et al. designed a system for swimming sonification with the aim of optimizing crawl stroke movement efficiency [38] [28]. Unlike previously described systems, this one sonifies pressure sensor data rather than movement data.

The mentioned studies experiment with various sonifications adopting different approaches: some of them sonify sensor data independently using one channel to control one synthesizer, while others consider data from multiple channels at the same time.

1.3.1.5 CardioSounds

CardioSounds is a wearable system for realtime electrocardiogram sonification [25]. The system can be controlled through a smartphone app. CardioSounds is meant to be used for monitoring heart behaviour in cases of cardiac pathologies using auditory feedback.

As described in [25], the system setup includes only one ECG lead sampling at 1000 Hz, but the authors deem important to add least another one. Moreover, plans for future development include the possibility of visualizing the data and twisting sonifications' parameters on the CardioSounds app, turning this system into an interactive sonification system. The app should also be capable of storing and retrieving ECG recordings and play them back using different playback rates.

1.3.1.6 MarketBuzz

MarketBuzz is a system for realtime sonification of financial data that can be used to monitor the movement of market indices [41]. Using auditory display can have significant advantages in contexts where there is a need for monitoring numerous data streams, and effective visual display would require multiple displays. Sonifications serve different purposes, e.g. tracking relative movement of an index (increasing or decreasing), the absolute position or the distance from a target.

1.3.1.7 Sonification of Facial Actions

SoFA (Sonifier of Facial Actions) is a realtime facial movement sonification system for musical expression [30]. A part for musical tasks, it could be applied to a variety of tasks, such as augmenting visual processing of facial gestures through sound.

Facial features are extracted from a webcam using optic flow and face detection algorithms. The optic flow step computes a grid of motion vectors for the image, while face detection associates motion vectors with face regions, excluding all motion vectors that are not relative to the face.

The mapping considers a pentatonic scale where lower parts of the face are mapped to low pitches and higher ones to high pitches. The amplitude (velocity) is computed from the motion vector magnitude. The system provides a GUI interface to twist a number of sonification parameters.

1.3.2 Common Aspects

From previously described systems, we can extract some relevant features shared by many of them:

- **Focus on real-time usage**: all described systems have real-time usage as their primary focus. Some of them support **recording and replaying data** using different sonifications.
- **Multiple sonifications**: support for adding, editing and loading sonifications.
- **Interactive sonification**: sonification parameters could be tweaked while the sonification is running.
- **Different kinds of stream**: streams can obtain data in different ways: we saw examples of data obtained from different kinds of sensors, through the internet (MarketBuzz 1.3.1.6) and extracted from a video stream (SoFA 1.3.1.7).
- **Multiple data streams**: multiple, independent data streams can be processed jointly.

These aspects are common to many other sonification applications. A framework that could address technical challenges behind these features would be a precious tool for developers, as it would allow them to focus exclusively on the peculiar characteristics of the application, highly reducing development effort. Moreover, it would give an initial structure to the application, reducing the time needed for designing and developing the system.

Chapter 2

Conceptual Design

In section 1.3.1, we reviewed some real-time sonification systems and discussed their common aspects. In this chapter, we will articulate those aspects into a concrete list of requirements for our framework and describe technologies that will be relevant to its development.

2.1 Requirement Analysis

Some of the systems that we discussed obtain data in **real-time** from **multiple sensor streams**. These sensor streams, in the general case, can have different frame rates, but nonetheless, they have to be processed jointly to compute useful sonifications; generally speaking, when we are dealing with streams of data coming from related sensors, the most important information is the one that we can derive from the relationships between the data coming from each of the streams, because it describes the overall behaviour of the whole system. Moreover, real-world systems are often subject to problems that could lead to irregular frame rates; e.g. noise or bad reception in case of data that is obtained through a wireless medium.

A useful operation to perform on real-time data is to record it persistently. In this context, as we are dealing with sonification systems, we also deal with auditory representations of the data; this representation is more interpretable than the raw, numerical representation, and certainly worth recording in many cases. It would be useful to be able to record the data in its row form or in its auditory representation. From now on, we will refer to the first case as **logging** and to the second as **recording**.

After the data has been logged, a mechanism is needed to replay the data using a sonification. In other words, we would like to have an effective **offline** mode of interaction with the data, that would allow to navigate its auditory

representation while interacting with the parameter of the sonification. Replaying data should generate identical sounds.

The framework should provide support for working with video and audio streams/files. More specifically, it should provide functionalities to record and replay video and audio along with the sonification of logged data. While video streams and files can be treated independently from the other components of the framework, audio streams and files require more thought, as it would be desirable that the framework could provide support for modifying audio streams based on a data stream.

The framework should be capable to work with different data and sensor systems and be reusable for applications with different aims. Reusability is best given by allowing the components of the framework to be recombined differently inside of an interactive environment. This is justified by the fact that we want the framework to be usable primarily in a research context, where tools such as Jupyter Notebooks (see section 2.3) widely used. Standalone applications for end-users should be derived only later on.

To sum up, the framework fulfill the following requirements:

- **R1.1**: support offline data sonification.
- **R1.2**: support online (real-time) data sonification, with support for multiple data streams.
- **R1.3**: support the recording of sonifications.
- **R1.4**: support the logging of data coming from real-time streams.
- **R1.5**: support playback of logged data with different sonifications.
- **R1.6**: support video and audio streams and files.
- **R1.7**: be independent from the kind of data that applications sonify.
- **R1.8**: support working in Jupyter Notebooks as main mode of interaction.
- **R1.9**: be cross-platform.

2.2 Python

Python is a high-level, general purpose programming language [45]. Nowadays it is one of the most widely spread programming languages and the

most popular choice for Scientific Computing, Data Science, Machine Learning and IOT [42]). Its popularity is partially due to the vast ecosystem of libraries Python developers can count on. Python supports extensions written in C and C++: an option when maximum computational efficiency is needed. Many popular libraries relevant for data processing implement their own extensions.

Given all of this, it is easy to see how Python would be a reasonable choice as programming language for our framework. By adopting Python, we could count on many mature libraries as foundation for our framework during development. Moreover, the framework would be based on a very popular language and published in a context where most of the libraries adopted would already be familiar to potential users, which would make it easier to learn.

2.2.1 Global Interpreter Lock

The default, reference Python interpreter, written in C, is called CPython [9]. It is also the most widely used implementation of Python. It is important to have a basic understanding of the architecture and functioning of CPython. In particular, there is one of its features that is relevant to our purposes: the Global Interpreter Lock (GIL).

The GIL is a mutex that protects access to Python objects by allowing only one thread at a time to execute Python bytecode [14]; this concretely means that multi-threaded programs written in Python cannot fully exploit modern multi-core CPU architecture by executing multiple thread on different cores. The purpose of the GIL is to avoid race conditions on shared interpreter data used for memory management [24]. The GIL allows to have minimum performance overhead for single-threaded programs, but strongly restricts the possibilities of multi-threaded programs. Different solutions designed not to impose these restrictions on multi-threaded programs were always found to have an unacceptable performance overhead for single-threaded programs, and for this reason were never adopted. A promising proof-of-concept of a Python interpreter without GIL and with some increase in performance has been implemented recently [6]. Nonetheless, we will have to design the framework by taking into account the present limitations.

The GIL imposes major limitations when working with CPU-bound, multi-threaded processes. In case of I/O-bound processes, every thread will release the GIL when starting to wait for I/O operations to complete, allowing another thread to be executed. In case of CPU-bound processes, threads will generally perform few I/O operation and run until they are preempted, blocking the execution of other threads. The Python interpreter switches

the running thread only in-between bytecode instructions. As we said, only a single thread runs in the context of a single program at every moment; for this reasons, each bytecode instruction is considered atomic. This means that most of intuitively atomic operations are actually atomic [53].

The typical workaround to this problem, is to **rely on multi-processing** rather than multi-threading. By doing so, we will have different processes executing their own interpreters in separate memory spaces; these processes run in parallel as they are scheduled by the operating system independently. The typical way to exchange data between multiple processes is to use Inter Process Communication (IPC), which is less efficient than relying on shared memory. Python provides in its standard library a module that allows to handle multi-processing by using a high-level API called **multiprocessing** [56]; this is the recommended library to be used for multi-processing in most of the contexts. multiprocessing also provides some support to allocate memory segments that are shared between different processes using shared memory maps.

An approach to bypass the GIL limitations could be developing a C or C++ extension to perform multi-threading tasks. Extensions are still subject to the same GIL limitations, but it is possible to manually release the GIL [52]. However, this solution would require a careful design and a much greater development effort.

This knowledge can come up as relevant when we design how to handle multiple streams. Ideally, we would want each data stream to run independently; using multiprocessing, it is possible to run each data stream on a different process, making it possible to spread them on multiple cores.

2.3 Jupyter Notebooks

The Jupyter notebook is an open-source interactive coding environment that allows to create documents (called "notebooks") capable of containing both executable code and rich-text [20] [39]. Originally, the project supported only the Python language, but nowadays it supports more than 40 languages; nonetheless, Python is still the most common language used in Jupyter notebooks. Jupyter notebooks can be used as literate programming tools to support workflow during research or to present analysis and results [43] [57]; they are also widely used as a support for teaching and collaborations.

When using Python in a Jupyter notebook, the actual interpreter running the code (referred to as "kernel") is called IPython [48]. Recent versions of IPython are capable of displaying different kind of media, such as images,

videos, audio, directly in the notebook. This powerful display system provides interactive GUI support through a library of widgets [50]. Ipywidgets are simply Python objects that have a graphical representation that can be displayed in the notebook; they provide a powerful mean of interaction between the user and the notebook. Users can also define custom widget representations for their classes.

2.4 SuperCollider

SuperCollider is an open-source platform for audio synthesis [11]. The sound synthesis engine of SuperCollider is a process called scsynth that behaves like a server process, listening for Open Sound Control (OSC) messages on the network and executing the respective commands to generate sounds. To communicate with the server, sclang, an interpreter for the SuperCollider language, is usually used [15]. In addition to scsynth and sclang, the SuperCollider project also includes an IDE, called scide.

The client-server architecture makes SuperCollider very flexible: client and server can run both on the same machine or on different machines, and multiple clients can hold connections with the same server at the same time. Moreover, we could substitute sclang with whatever program that is able to communicate with the server.

2.4.1 sc3nb

sc3nb (SuperCollider 3 Notebook) is a Python package that implements a SuperCollider client [37] [12]. Using sc3nb, the Python library ecosystem for data analysis can be interfaced with SuperCollider sound synthesis capabilities, using data to drive sound generation. All of this can be used interactively inside of Jupyter notebooks. sc3nb has a lean interface that mimics the one of sclang in many respects, making it easy to learn for users who are already familiar with SuperCollider.

sc3nb is able to interact with and control both sclang and scsynth. It can interpret the SuperCollider language by sending the code as an I/O stream to sclang for evaluation. Given that sc3nb is not natively capable of compiling synthesizer definitions, this task is also normally delegated to sclang, but it can also be performed by other Python libraries that support this functionality, such as supriya [8]. In case we do not need any of the functions that sclang is used for, we also have the option to avoid starting it, reducing significantly time required for the setup.

As stated in the code repository, sc3nb "is meant to grow into a backend

for a sonification package" [12]; this stated intention is important because it suggests that we could submit pull requests to the project.

Chapter 3

Panson

The requirements for the framework have been laid out in the previous chapter, along with the tools that we intend to use. It has to be mentioned that the requirements for the facial feature sonification application (discussed in section 4.1) also had an important influence on many design choices, driving the development of the framework. This chapter will present the framework design and explain the main concepts and implementation decisions.

Many sonification projects have their name ending with "son"; given that our framework attempts to be a general tool for implementing interactive sonification applications, we thought about a simple name that could capture this idea: Panson (from pan- "all" and son for sonification). A concrete advantage of this name in terms of usability is that users will be able to import the framework with a short name as "ps", similar to what is common to do with pandas.

```
import panson as ps
import pandas as pd
```

It is worth noticing that, despite the similarity, "ps" and "pd" remain easily distinguishable by sight by anyone working with the code.

The reference code implementation of Panson is available at [17], under the directory `panson/`.

3.1 Design

Panson was developed using an object-oriented approach, designing classes to encapsulate different features in such a way that the elements of the framework are easily composable. As specified, Panson's main mode of interaction

is meant to be an interactive usage in a Jupyter notebook environment (R1.8); this forced us to carefully consider peculiarities and limitations of this kind of context.

In approaching the design, we chose to adopt a bottom-up approach, starting from the implementation of core functionalities and building the rest of the framework on top of that. This kind of approach allowed us to quickly develop a prototype that we could use to build the facial feature sonification application on top of. The simultaneous development between Panson and the facial feature sonification application made sure that there was a mutual influence in the development between the two; for this reason, the requirements of the application described in section 4.1 are also relevant to the development of Panson, given that an effort was made to develop Panson in such a way that it could enable the application to satisfy those requirements easily, while trying to keep the framework general-purpose. Concretely, this meant starting the development implementing prototypes for the Sonification class (3.1.1) and data player classes (3.1.4), so as to have the basic features relative to data processing and sonification playback in place. To support this kind of development approach, continuous refactoring stages were necessary to restructure and simplify the code. Section 3.1.4 contains some diagrams that show the general structure of the elements of the framework in different contexts.

A general guideline behind the development was the idea that users of the framework should be able to implement applications without having to resort to the usage of any of the advanced features of Python, such as generators, decorators and metaclasses, unless these features would be really necessary. Moreover, Panson should encourage to write code in a concise and maintainable manner, using common Python mechanisms: in other words, it should encourages the writing of pythonic code [55].

3.1.1 Sonification

The Sonification class is meant to be used to **define the sonification behavior** independently from its context of use (online or offline). It is an abstract class that provides template methods for specifying how a sonification behaves, i.e. how data is mapped to sound, at different stages of its execution:

- Initialization: this phase takes place when the sonification object is created.

  Hereafter we show the signatures of the two abstract methods that Sonification requires to implement:

```
init_parameters(self, *args, **kwargs) -> None
init_server(self) -> Bundler
```

init_parameters deals with the initialization of sonification's parameter with their default values (see section 3.1.1.1). Its positional and keyword arguments are passed to the method by the `__init__` method.

init_server returns a bundle for allocating resources that will be needed for the whole life of the sonification, e.g. loads synthdefs needed or loads samples in buffers.

- Start of playback:

```
start(self) -> Bundler
```

This method could be used for tasks such as allocating continuous synths that run for the whole sonification or setting attributes that have to be set at each sonification start.

- Processing of a data sample:

```
_process(self, row: Series) -> Bundler
```

This method is where the mapping from data to sound is defined. The "row" argument contains the data in the form of a Pandas Series object, where every data value is accessible through its label; it is called "row" because it can be thought of as a row in a Pandas DataFrame. This method will be called once for every sonified data row.

The underscore preceding the method name indicates that the method is private. This is because the actual method that will be called on each data point is called "process", which is a wrapper around "_process". The wrapper ensures that parameters are not changing while the mapping is being computed.

- Stop playback:

```
stop(self) -> Bundler
```

This method is not required to be overridden. By default, it frees all the synths belonging to the default group associated to the SuperCollider client.

- Freeing of allocated resources:

  ```
  free(self) -> None
  ```

  This method should be overridden only if there is a need for releasing resources allocated during initialization, such as buffers. By default, it does nothing. This method has to be called explicitly.

In implementing these methods, users should be free to use whatever sc3nb programming style they prefer, i.e. low-level interaction based on OSC messages or high-level interaction based on sc3nb's node objects. Nonetheless, the primary focus will be supporting the highest-level node-based programming style, as it is the recommended one.

The independence of sonifications from their context of use is achieved by not allowing methods implementations to interact directly with the SuperCollider server, but rather using Panson's data players 3.1.4 as intermediaries. For this reason, methods - rather than sending messages to the server as side effect - returns the messages in a sc3nb structure called bundler; the responsibility of their actual usage is on the data player object used, which will also ensure the correct time of execution of the commands.

Panson provides first-class support to sc3nb high-level node-based programming style through the **"bundle" decorator**, which can be a applied to the implementation of Sonification's template methods to capture all messages sent in the body of the method in a bundler object to be returned; this mechanism is a clean and readable way to gather and return all messages generated by sc3nb when interacting with node objects. When introducing the design, we stated that advanced features of the Python language, such as decorators, should have not been exposed to users unless really necessary; nonetheless, we considered the use of a decorator as the simplest solution for this.

It is important to notice that resource allocation (nodes, buffers and buses) in SuperCollider is handled on the client-side, which means that sc3nb (the client) has data structures (allocators) for handling resource ID allocation. In a multi-client SuperCollider setup (the one in use with sc3nb) clients negotiate the ID range to use at registration time so that resources can be shared between different clients without interference [18] [22]. In sc3nb creating a Node object (Synth or Group), a Buffer object or a Bus object, changes the allocators' state on the client, even if the messages generated by those operations are not actually sent to the server. When rendering the full sonification using the "export" function of the DataPlayer it will be necessary

to instantiate resources such as buffers on the non-realtime server (see section 3.1.4.1); for this reason, Sonification stores the initialization message, so that `init_server` is not called multiple times, causing affecting the allocators' state.

3.1.1.1 Parameters

As Panson is meant to be a framework for *interactive* sonification, it must provide a way to interact with sonifications' parameters while they are running. This functionality is provided through a set of classes we refer to as "parameters".

When thinking about how to handle such a problem, the first solution that comes to mind is relying on object attributes that store the values and that are used to compute the mapping in the "_process" method. Assigning new values to these attributes would modify the behavior of the sonification. This approach has the problem that attributes could be changed in the middle of the computation of a mapping, potentially causing inaccurate or inconsistent results. Given that in Python only one thread at a time is allowed to run (see discussion in 2.2.1), this problem may be not so frequent, but can nonetheless happen when the thread is preempted while performing the sonification after the interpreter's thread switch interval has expired. The way to solve this problem is to make the data processing *atomic* with respect to assignments of values to sonification's attributes. To do so, some sort of locking mechanism is needed; the goal is to implement it in the most transparent way for the user.

The chosen solution relies on a feature of Python known as *descriptors*, which are a way to customize attribute lookup, assignment and deletion of classes [54]; descriptors are simply classes that redefine those operations. We utilize this feature to make the computation of the mapping ("_process" method) atomic with respect to the parameter modifications in a way that is almost completely *transparent*. **Parameter** is the simplest class leveraging this mechanism, with the only feature implemented being the one just mentioned. What it does is redefining the assignment operation in such a way that the actual assignment of the value only takes place after having acquired a lock; this lock is the one acquired by the "process" method before computing the mapping, implementing the atomicity requirement that is needed. Parameter instances can only work properly if declared as *class* attributes, rather than instance attributes; indeed, a Parameter attribute works by creating a private attribute in the owner object (a concrete Sonification instance) on the first time is assigned and managing access to it using the aforementioned lock. The method "init_parameters" is required to assign

an initial value to each of the parameters to make the whole mechanism work properly and there are no restrictions on the type of the value assignable. It should be emphasized that only attributes that are meant to be modified interactively have to be defined as Parameter instances to avoid unnecessary overhead.

The Parameter class only provides support for interacting with attributes through code. Given the requirement of providing support for interactive usage in Jupyter notebooks, we extended the idea behind the Parameter class by adding ipywidget-based representations to parameters. From now on, we will refer to these classes as "widget parameters".

When a sonification is displayed in a notebook, its representation is simply a label containing the name of the class followed by all the ipywidgets-based representations corresponding to widget parameters: this makes widget parameters an effective way to define widget-based GUIs using a declarative approach, with a method that can be easily extended to be used with GUI toolkits such as Qt. Interacting with ipywidgets changes the parameter value. Widget parameters can be defined subclassing the abstract class WidgetParameter. Panson includes a library of widget parameters that allow to use many ipywidgets useful in this context [51], such as sliders, range sliders, dropdowns, checkboxes and more. The library also contains some useful custom classes:

- DbSliderParameter: a slider of decibel values from -90 to 0.
- MidiSliderParameter: a slider of MIDI values.
- FreqSliderParameter: a non-linear frequency slider from 20 Hz to 20 kHz.

In file `panson/widget_parameters.py` at [17] we can find the implementations of all widgets parameters. It is possible to see some examples of widget renderings of sonification objects in section 4.4.

3.1.1.2 GroupSonification

The GroupSonification class allows to group multiple sonifications together, so that they can be controlled simultaneously by a data player; these sonifications have to be passed to the constructor when instantiating the class as a sequence. Having such a tool available enables designing sonifications in a more modular way.

GroupSonification implements the same interface of the Sonification class, where each method simply calls the respective methods on the sub-components.

For instance, any data series processed by the GroupSonification object will be fed to all its sub-components' processing method and the resulting bundle will contain all OSC messages generated.

The widget representation that will be rendered in a Jupyter notebook when a GroupSonification object is displayed is nothing more than a vertical stack of the widget displayed by its sub-components.

3.1.2 Streams

Real-time data streams can be defined using the Stream class. Stream objects are the **input data backend** of applications built with Panson and make the rest of the application independent from a specific data source. To specify how to obtain real-time data, we have to implement a **generator function** that obtains and returns the data and provide the Stream class with it. Generator functions are functions that return an iterator; Python provides a user-friendly construct to define generator functions through the "yield" statement [59]. This statement controls the flow of the generator function, pausing the execution of the function and returning the value; when a new value will be requested to the generator, the execution will be resumed from where it was paused. Even if this construct may seem confusing, it captures exactly what is needed in this context and, being a standard Python mechanism, plenty of documentation and tutorials are available online.

The design of the Stream class is inspired by the Thread class from Python's threading module. The data generator function can be specified in two ways:

- It can be passed to the "datagen" argument of the constructor when instantiating the Stream class.
- It can be specified by overriding the "datagen" method when subclassing the Stream class.

The data generator function should respect some constraints.

- The first element yielded should be a list containing the labels of the data.
- All subsequent elements yielded should be numpy arrays of the same length of the list of labels. Each array returned will be assembled with the list of labels into a pandas Series object, so that, users of the framework will have to interface with Series objects rather than raw arrays.

In case what the user want is to yield single-value data, e.g. sensor stream, Panson requires anyway to return a one-sized numpy array.

- Each array must have same type of elements.
- The generator is required to block and sleep when waiting for new the data, to avoid waste of CPU.

In case the user needs to carry on operations when the stream is opened or closed, the Stream class provides a mechanism to specify function to be executed, referred to as "hooks". For instance, hooks could be used to switch on/off the data flow from the sensor system. If we specify a Preprocessor class (see 3.1.3), the preprocessing will be computed on each of the data points before yielding the value out of the stream.

The Stream class provides a "test" method to help users with debugging of their custom defined streams, as well as testing if everything is working as expected at a certain point. This method tries to open the stream and get a number of data samples from it. It can also be used to inspect and set information regarding the stream; more specifically we are talking about:

- The size of the arrays returned.
- The type of the arrays returned.
- The estimated frame rate.

These pieces of information may be also required by other components of the framework (see section 3.1.4.3).

Panson contains a number of predefined streams that can be used for testing. This collection is meant to grow in a library of Stream classes that could be used for most common situations.

3.1.3 Preprocessors

The Preprocessor class is used to define preprocessing methods for the data. This class is an abstract class that requires the implementation of only one method named "preprocess", of which we can see the signature below.

```
preprocess(self, row: pd.Series) -> None
```

As we can see, this method expects a Series object as an argument and it does not return any value; this means that the implementation of preprocess will only be useful for its side effects on the Series argument.

The user can also define attributes for the class in the initialization method, but, for now, it is not possible to pass arguments to it. This happens because the preprocessors are not to be instantiated directly by the user: the user is required to pass the preprocessor type to the framework components that require it. The type plays the role of a builder for preprocessor instances, so that a new preprocessor will be instantiated every time that it is needed, e.g. every time that a stream is opened.

This mechanism is very simple for the user, but it fails to be flexible because of the impossibility to specify parameters for the constructor. A possible solution to this could be the use of an actual builder object in place of the processor type. This would require some thought, as there may be more elegant solutions.

3.1.4 Data Players

Data players are used to apply sonifications to data. Different data player classes were implemented for different purposes. Each one of them processes data using the Sonification object and sends the resulting bundler to the server with the correct timing. When starting up the SuperCollider server through sc3nb, it is important to set an appropriate **server latency** value for the system configuration, so that bundlers will be executed on time.

All data players can be displayed in Jupyter notebooks using ipywidget-based views; these views allow to graphically control most of the data player functionalities. Output of the data player will be printed in the notebook cell, after the widget view; to clear the output the only supported way is to re-evaluate the cell. For this reason, it is important to display the data player in a separate cell from the one in which it is created. To support a more user-friendly mode of interaction, we thought initially of providing widget views with an output widget [49] capable of capturing the output printed by the data player and cleaning it up at the user's command. Unfortunately, output widgets are not designed to work with multi-threading or multi-processing, and could not be adopted without causing major problems.

3.1.4.1 DataPlayer

DataPlayer is the class to be used when doing *offline* processing: it supports loading data from CSV files or from pandas DataFrame objects. Loaded data can contain timestamps, or a static frame rate can be specified. If timestamps are used, the first timestamp value does not necessarily have to be 0: that value will be used as an offset for all timestamps. Internally, the data player loops over rows of the DataFrame with the correct timing and com-

putes the sonification on each data row by applying the specified sonification. It is possible to interactively jump around the data and to specify a playback rate value to speed-up or slow-down the playback; if the specified value is negative, the playback will be reversed. All these operations can be carried on while the data player is running. This is implemented by stopping the playback, executing the operation and resuming the playback.

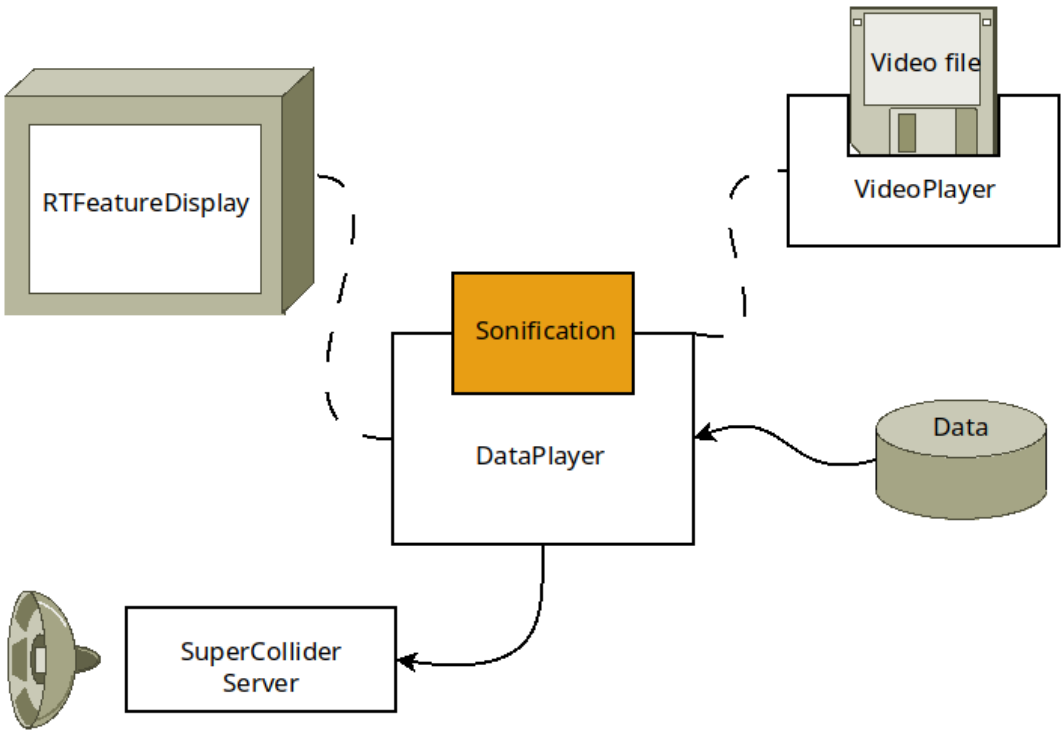

Figure 3.1: Sonification of offline data. Dashed lines indicate optional elements. For a description RTFeatureDisplay and VideoPlayer see 3.1.5 and 3.1.6.1.

This class is capable of **recording** the audio output generated throughout its playback by leveraging sc3nb's Recorder class. Users have to explicitly start and stop recordings, which can be done from the widget view. Beyond this, DataPlayer offers an "export" feature capable of rendering the full playback of the data in an audio file using SuperCollider's **Non-Realtime Synthesis** feature [19]. To perform this operation, the sonification object is cloned, and the clone is used to compute the sonification of all the data, storing the results (server commands) in a "score" object. The cloning operation is necessary because we do not want the original sonification object to be affected by side effects that may take place during the computation of the

sonification. In computing the sonification of the data, the data player will not call the `init_server` method on the Sonification object not to cause side effects on the allocators (see section 3.1.1), but it will access an initialization bundle memorized during the instantiation of the sonification object.

When creating an DataPlayer object, it is possible to specify a VideoPlayer object (see 3.1.6.1) for the constructor. The DataPlayer will use the VideoPlayer object to display the video synchronously with the data.

Figure 3.2: DataPlayer widget.

3.1.4.2 RTDataPlayer

RTDataPlayer is designed for *online* (real-time) processing using a **single stream**. When started, the data player will open the stream and start continuously consuming its data. Each data sample obtained will be converted into a pandas Series and passed to the user-defined sonification processing code. The resulting bundler object is immediately sent to the server and executed (after server latency).

RTDataPlayer, similarly to DataPlayer, is capable of recording generated sounds in an audio file. Additionally, it is also capable of logging data in a .csv file; logged data can be replayed using a DataPlayer.

When creating an RTDataPlayer object, it is possible to specify a RTVideoPlayer object (see 3.1.6.2) for the constructor. When recording or logging data, if an RTVideoPlayer object was specified, the video stream will be recorded.

3.1.4.3 RTDataPlayerMT and RTDataPlayerMP

RTDataPlayerMT and RTDataPlayerMP are both **multi-stream data players**: they are able to jointly process multiple independent data streams with different frame rates. Their implementations are based on multi-threading and multi-processing approaches respectively, but their interface is the same; in the future these two classes could be merged into one.

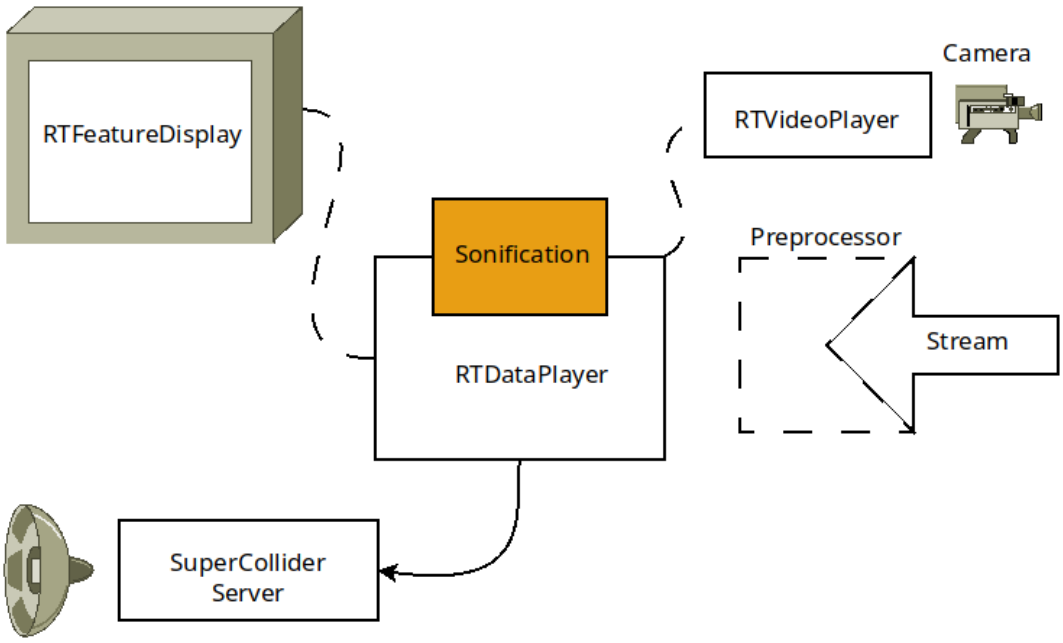

Figure 3.3: Real-time sonification of data. Dashed lines indicate optional elements. For a description RTFeatureDisplay and RTVideoPlayer see 3.1.5 and 3.1.6.2

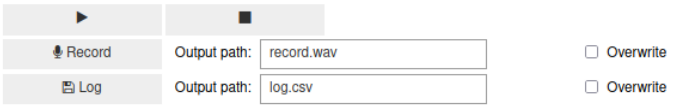

Figure 3.4: RTDataPlayer widget.

Multi-stream data players have to compute sonifications on data coming from multiple streams with potentially different frame rates. To achieve this, they have to compute the sonification at regular intervals, taking into account the most recent data sample from each of the streams. The most recent data from every stream is joined into a pandas Series; to make this possible, labels associated to data from different streams must be specified in such a way that no name collision will occur. The frequency at which the sonification is computed can be specified as a constructor argument; if it is not specified, the highest stream frame rate is chosen. This is only possible if streams' frame rates are known: users can set this piece of information explicitly or can estimate it using Stream's "test" method (3.1.2).

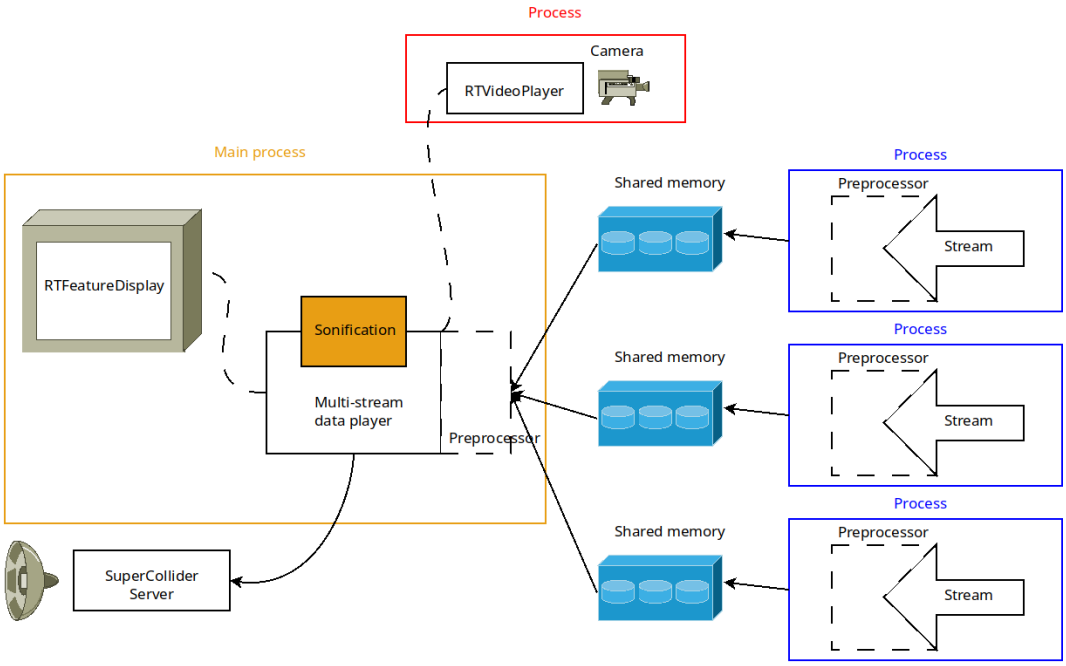

Figure 3.5: Real-time multi-stream sonification based on multi-processing. Dashed lines indicate optional elements.

Multi-stream data players can perform logging in different ways:

- **Stream-level** logging logs every data sample generated by a single stream.

  Currently this is implemented only in the RTDataPlayerMT.

- **Global-level** logging logs the multi-stream data that is sonified. Logged data can be replayed using a DataPlayer.

Data can optionally be timestamped by the data players both when it is obtained from the streams and when it is sonified.

As we discussed in section 2.2.1, Python's multi-threading is not suitable for all situations because of the limitation imposed by the GIL. RTDataPlayerMT was initially developed as a prototype, given the easier implementation of multi-threading programs with respect to multi-processing programs. Indeed, when relying on multi-processing, we need some sort of inter-process communication (IPC) mechanism. Nonetheless, a multi-threading solution can be useful if we have to deal with few streams with light or no preprocessing: here multi-threading would be preferable to multi-processing as it does

not have to deal with the overhead due to IPC. Moreover, RTDataPlayerMT could be used instead of RTDataPlayer in case we do not want to sonify all the data produced by the stream, but rather we would like to run the sonification at a specified frequency.

RTDataPlayerMT starts a thread for each stream and sonifies the last data available from each stream at a regular frequency in another thread. RTDataPlayerMP works in a similar way, but using processes instead of threads. These processes continuously fetch data generated by streams into slots of memory that is shared between different processes. In Python, this is possible by using the Array class from the multiprocessing module [56], which provides synchronized access to a shared memory map; this allows ti move data between processes using a mechanism that is more efficient than IPC. Before the sonification is computed, local copies of all the shared-memory arrays are performed and joined into a Series object.

To allocate shared memory maps, it is necessary to know in advance the amount of memory to allocate. For this reason, it is necessary to know the size of the arrays returned by each of the streams and the type of their elements. This piece of information can be obtained and set for each of the streams calling their "test" method.

Multi-stream data players support preprocessing of joined multi-stream data through the Preprocessing class. The only preprocessing operations that should be executed here are the ones that need to access data from multiple streams simultaneously: all the preprocessing operations that could be handled at a stream-level should be delegated to the Stream class. This is especially true when using RTDataPlayerMP: as streams are executed in other processes, executing the preprocessing in the streams will distribute the CPU load.

Figure 3.6: Widget for multi-stream data players (with two streams).

3.1.5 Feature Displays

Feature displays allow to display in real-time values of the sonified data. Currently, only one class of this family is implemented: the RTFeatureDis-

play.

RTFeatureDisplay displays the history of most recent values sonified using a first-in first-out (FIFO) policy. At creation, it requires to specify the labels of the data features to track and the size of the queue to be used (infinite by default). This class is meant to be led by a data player and can work with all data player classes (also offline). Data players feed the RTFeatureDisplay object with data in the form of Series objects and the RTFeatureDisplay object updates the dynamic plot with the features corresponding to the specified labels.

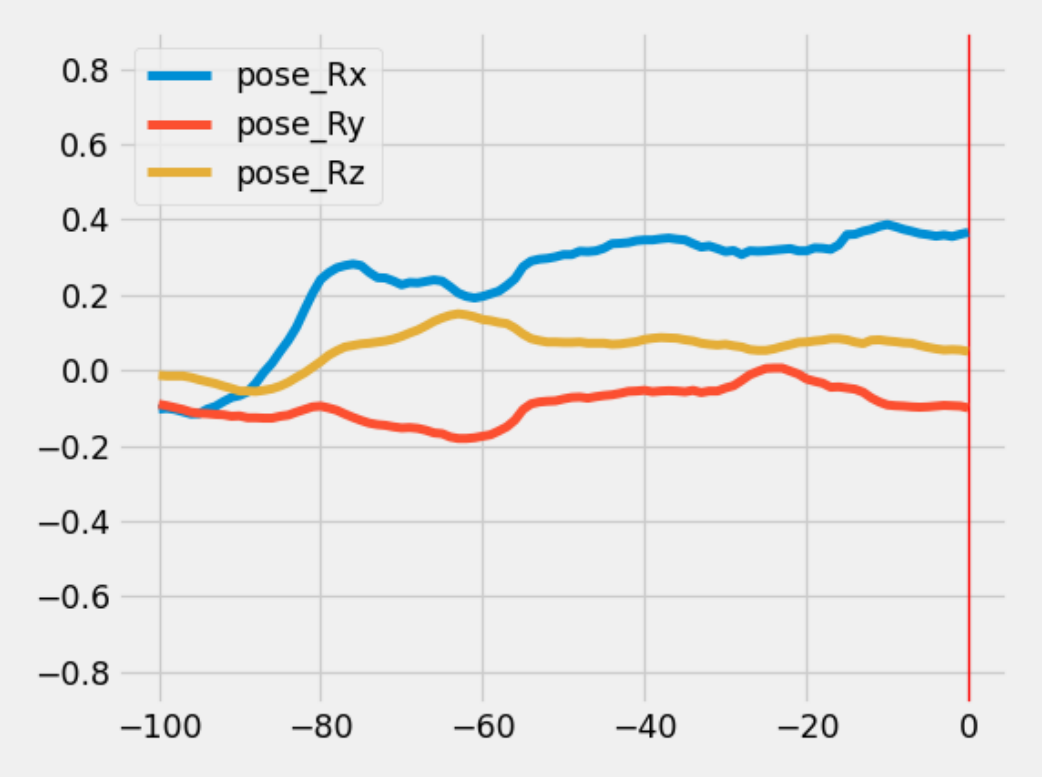

Figure 3.7: RTFeatureDisplay displayed inline in a Jupyter Notebook.

A RTFeatureDisplay object can be fed with data in the form of a pandas Series and in result it will update the dynamic plot containing the data values stored in the queue. The only values that will be taken into account by the data are the ones specified in the label list passed to the constructor.

The "show" method displays the plot and starts updating it; this is done using matplotlib. If we are using a Jupyter notebook, the appropriate mat-

plotlib backend ("notebook") is set by default, allowing the user to display dynamic plots directly in the notebook. By displaying a feature display in the same cell of another component, we can compose different representations. Below, we demonstrate how to display a feature display on top of a data player.

```
feature_display.show()
display(data_player)
```

When sonifying offline data using a DataPlayer object, the FIFO behaviour of the RTFeatureDisplay is not always desirable. In this case, it would be useful to have a feature display that could show a window around the current data point being sonified, rather than the history of most recently sonified data points. Such a functionality is not currently implemented, but will be implemented as **FeatureDisplay** class.

3.1.6 Video Players

Video players provide support for handling video files and streams through two separate classes: VideoPlayer and RTVideoPlayer. These classes are both proxies that communicate with separate processes, responsible of carrying on the actual operations; this is done to bypass Python's GIL (see 2.2.1).

Video players display a window for the video playback even when they are used in Jupyter notebooks, as Jupyter notebooks provide support for displaying image data, but it is not appropriate for displaying videos.

3.1.6.1 VideoPlayer

The VideoPlayer class provides functionalities to handle the **display of video files**. When instantiated, it displays a window where the frames of the loaded video file are displayed. It offers the possibility to specify a constant frame rate or a timestamp file containing the precise timestamps for each frame of the video.

The current implementation of the VideoPlayer is not designed to be a stand-alone component, but rather to work together with a DataPlayer object. The DataPlayer perform seek operations on the VideoPlayer, which will update the displayed video frame.

In the first prototype implemented, the DataPlayer was seeking video frame using their indexes while running. However, this approach is only applicable if to each data sample corresponds one video frame; in general, this

assumption cannot be made as the data and the video stream can be independent. A trivial solution would be seeking time instead of indexes; VideoPlayer supports this operations, but it implements it using binary search, which has a logarithmic complexity in time, rather than the constant complexity necessary to perform an index seek. Having a non-constant complexity for an operation that is executed for each data point sonified is not a satisfactory solution as it may not be efficient enough to be used with long video files. This means that the current implementation is not complete. The final VideoPlayer should be able to playback video files without relying on any external component, roughly like the DataPlayer does with data. A standalone VideoPlayer component designed to bypass the GIL would be useful in many different projects. It is possible that this component will grow into an independent project.

To handle loading of video files, VideoPlayer relies on the PIMS library [21] [58]. This library provides means for performing random access on video frames without having to load the whole video into memory, enabling the possibility of working with large video files.

3.1.6.2 RTVideoPlayer

RTVideoPlayer provides support for **displaying and recording a video stream** coming from a device. When the object is created, a process that grabs frames from the device and displays them in a window is started. If recording is started, grabbed video frames will be written to a video file and their precise timestamps will be written in a CSV file, in a format that could be directly used by the VideoPlayer class. If an RTVideoPlayer object is passed to a real-time data player (single-stream or multi-stream), it will start recording whenever the data player starts recording or logging; with multi-stream data players stream-level logging does not start any recording.

3.1.7 %%editor

Sonifications can be coded in Python source code files, to be imported before their use, or directly in Jupyter notebooks. In the second case, the default Jupyter notebook environment, with its basic text editing functionalities, is not the best choice in case of complex sonifications. Nowadays, only few text editors and IDEs provide support for working with Jupyter notebooks.

Panson provides a prototype of a cell magic [40] called “editor” to support efficient development and editing of sonifications. This magic is inspired by the built-in IPython magic “edit”, which does not work properly in Jupyter

notebooks; "editor" is its adaptation to work in Jupyter notebooks. The "editor" magic opens the cell where it is contained in the editor specified by the $EDITOR environment variable. When the editor is closed, a cell containing the edited text will be created below the current cell (it is not allowed to substitute the content of the current cell). Using this magic, users can perform sophisticate text editing operations using their favorite text editor. Being a cell magic, it has to be placed at the beginning of the cell and called as the following:

```
%%editor
# cell content
...
```

This magic is general-purpose and not bound in any way to the sonification context. For this reason, it would be a good idea to move this component into an independent project. At the time of writing, no other similar project was found.

3.2 Evaluation

In this section, we will describe the results of a simple technical evaluation related to various real-time setups of Panson. We decided to take into account different real-time setups because it is where performance matters the most. This evaluation does not aim to be complete or accurate, but only to give an overview of the performance of the framework. The evaluation was carried out on a laptop MSI Modern 15 A11M-217XIT using an **Intel Core I5-1135G7**.

3.2.1 Latency

An important metrics to consider is **processing latency**. In real-time scenarios, processing latency is the amount of time from when the data is obtained from the stream to when the bundler that updates the sonification is sent to the SuperCollider server; this measure does not include server latency, which is dependent on the server's settings specified.

To measure this, we used a dummy sonification that simply returns an empty bundler, so that no time is consumed by what would normally be user-implemented processing code. This kind of evaluation is only possible using the RTDataPlayer as in multi-stream data players data acquisition phase and processing phase are independent, i.e. the processing is run at regular intervals.

The mean processing latency has been proved to be of around **0.49 ms**, independently from the size of the data arrays returned by the generators. This value has been obtained using sizes varying from 1 to 1 million of elements per array. Of course, the measured processing latency does not include the time necessary for the allocation of the memory of the arrays, which takes place inside of the generator. The reason why the processing latency is independent from the data sample size is that RTDataPlayer simply packs the data header and the data sample in a Series; this operation is performed by passing the references of the arrays, without any copy of data.

0.49 ms is a small value if compared with typical server latency values, where 50 ms is considered a small value. Most of the total processing value will always be due to user-implemented code in streams and sonifications; this is a desirable result because it means that the framework does not add a big computational overhead to user-defined operations.

3.2.2 CPU Usage

Here we measure the approximate CPU usage of real-time data players using different stream frequencies and sonification frequencies (i.e. how often the sonification is computed); this will give some insights on the scaling abilities of the framework. In the following results, 100% indicates the full use of a single core, meaning that it is theoretically possible for a multi-threaded process to have values that are higher than 100%. However, this is not possible for a Python program because of the limitations due to the GIL (see section 2.2.1).

To perform these evaluations, we used a sonification that does nothing at any of its stages, so that its impact on CPU consumption is negligible. The stream adopted generates zeroed arrays of 1000 elements at the specified frame rate.

Image 3.8 shows the results of the evaluation of the RTDataPlayer class. CPU consumption appears to maintain linear growth as frequency increases. This is both expected and desirable. Regarding the values of CPU consumption, we must notice that, in a typical situation, arrays returned from the stream will be smaller in size than 1000 elements.

In image 3.9, we can see a comparison between the multi-threading approach to multiple streams and the multi-processing one. To test this, we used two of the aforementioned streams; this means that there are a total of three threads/processes running: two of those associated to streams and the other one performing the sonification. In this evaluation, we used a same frequency value for both the streams and the sonification; in general, different values could be used for each of the stream and for the sonification frequency.

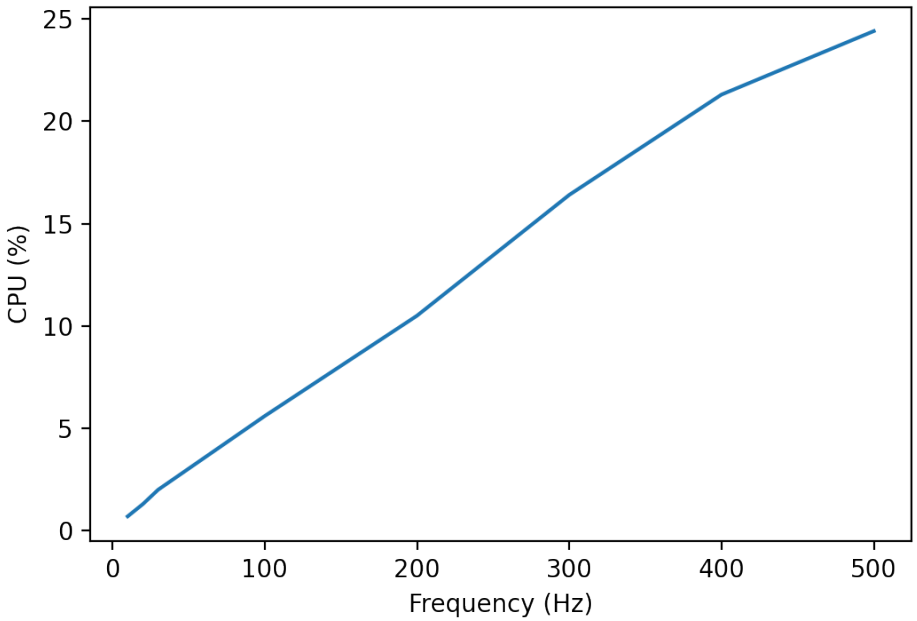

Figure 3.8: CPU consumption of a RTDataPlayer at different frequencies.

The values obtained from the multi-processing evaluation only take into account the CPU usage of the main process (the one that computes the sonification). This will give an idea of how much CPU usage can be spread on multiple cores. The total CPU load of multi-processing would be superior to multi-threading, as the multi-processing approach has to deal with the overhead of copying the data to the shared memory space, whereas multi-threading can rely on actual shared memory.

With this setup the multi-threading data player seems to perform well for frequency values inferior to 100 Hz. Indeed, we must remember that for the multi-processing data player only the CPU consumption of the main process is shown, which means that the total CPU consumption would be higher. For higher frequency values, multi-processing is able to significantly reduce the main process' CPU usage, spreading the stream processes on other cores. The advantage of this approach would become even clearer if we used a much greater number of streams or if we used independent frame rates for the streams and the sonification processing.

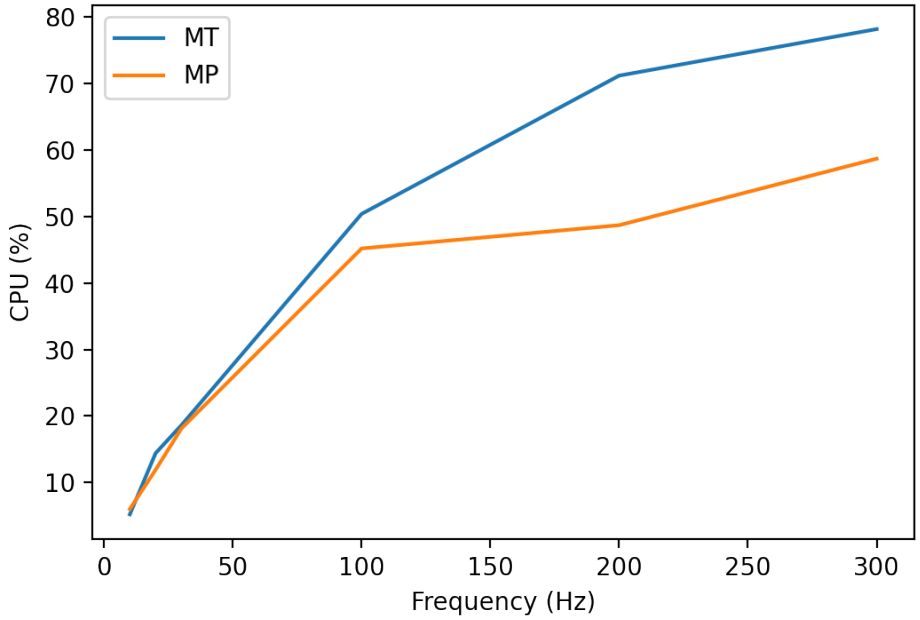

Figure 3.9: Comparison between CPU consumption of multi-stream data players.

Chapter 4

Facial Feature Sonification

This chapter discusses the design and implementation of an **interactive facial features sonification application** based on Panson. As we already mentioned, it is not correct to think that the application was unidirectionally influenced by Panson; in fact, the application's requirements traced the path for Panson's development during early stages of the project. Indeed, the overall goal of this thesis was to implement a facial feature sonification application using components that could be reused for other similar applications. We can say that Panson and the application had reciprocal influence in their development.

4.1 Requirement Analysis

The general aim of the application is to provide an interactive system for facial expression sonification that could adequately support a number of use cases that we describe in this section.

4.1.1 Use Case 1 - Support for Visually Impaired People

Visually impaired people could use the application for sonifying their own facial expressions in real-time (using a webcam) to learn how generated sounds relate to their proprioception. After the learning phase, they could use it during a conversation with another person to sonify their interlocutor's facial expressions.

Sonifications should be informative and convey through sound as much as the communicatively relevant information as possible while being easy to perceive and interpret correctly. Ideally, the users should quickly get

used to the sounds generated, so that their primary focus can shift on the conversation.

Sonifications designed for this use case should be:

- **R2.1**: easy to learn
- **R2.2**: unambiguous
- **R2.3**: unobtrusive

4.1.2 Use Case 2 - Support for Facial Expression Analysis

Sonification can be a powerful tool for highlighting qualities of the data by drawing attention on them. The application should ideally be a useful tool for facial expression analysis, allowing to leverage hearing as a way to guide attention to some meaningful details that otherwise would be easily ignored. Facial Expression Analysis can infer higher order information (such as basic emotions, pain, personality traits, etc.) more accurately than subjective judgments based on intuition. For instance, using FACS (see section 1.2), it is possible to distinguish truth from lying much more effectively than using arbitrary judgements [29].

The application should provide functions to support users in facial expression analysis; this means providing ways to record the video and data streams, play them back using different sonifications and navigate them. It should also provide ways to perform offline processing of video files that were not recorded within the application.

The application should be capable of:

- **R2.4**: recording generated audio (and eventual video)
- **R2.5**: recording the data
- **R2.6**: playback the data using different sonifications

4.1.3 Use Case 3 - Facial Expression Practice Tool

The application should be a useful tool for practicing facial expressiveness. For example, it could be used by **actors**; using an **hybrid online/offline mode of interaction**, they could compare the sounds generated by their own facial expressions recorded in real-time through a webcam to the sounds generated from the offline processing of a recording and practice matching the two as much as possible. Another possible usage could be as a support for

facial physiotherapy exercises. Nowadays, many forms of brain-damage that compromise facial expressiveness are often treated with facial exercises; these could be augmented using facial feature sonification.

Ideally, it would be possible to apply this mechanism to faces of different people. This would be useful to practice imitating facial expressions. An appropriate sonification for this use case would not be overly influenced by the topology of the face, so that it would be possible to compare facial expressions of different people.

The application should be capable of:

- R2.7: sonifying real-time data along with prerecorded data
- R2.8: sonifying facial expressions from different faces in a comparable manner

4.2 OpenFace 2.0

OpenFace 2.0 is an open-source and cross-platform computer vision tool for facial behavior analysis, capable of performing facial landmark detection, facial action unit recognition, head pose estimation and eye-gaze estimation [26]. In these tasks, OpenFace is capable of state-of-the-art results as well as real-time performance. OpenFace can run without the need for any sort of specialized hardware, such as GPUs. Finally, it is freely available for research purposes in all its functionalities. For all this reasons, OpenFace is a good choice for performing feature extractions on video files or streams in the context of our feature expression sonification application.

4.2.1 Installation

In addition to the default installation method, the project provides a pre-built **docker container** with OpenFace executables. Using this option is a convenient way for users to install OpenFace without having to deal with potential dependency issues.

4.2.2 Usage

In this section we will mention some relevant details about OpenFace command line interface; the full documentation is available in the Wiki of the Github repository [7]. OpenFace also provides a Windows-only GUI interface, which is not relevant in our case since the application will interface to OpenFace by calling its command line executables from code.

OpenFace includes multiple executables. but the only one that is relevant to our purpose is called `FeatureExtraction`: this executable can be used for sequence analysis (on video files or sequences of images) that contains only one face. Below we will describe the relevant command line arguments of `FeatureExtraction` and the associated functions.

4.2.2.1 Offline Processing

The `-f` option followed by a video file path is used to specify a video file to process. Multiple video files can be specified using multiple `-f` options.

If we have a directory containing a sequence of images representing frames, we can process the whole sequence of files as if it was a video file using the `-fdir` option followed by the path of the directory. Formats of images supported are JPEG, PNG and BMP and files must be named in the correct order, e.g. encoding the frame number in their names. The output generated will be associated to the full sequence of images rather than to single images.

4.2.2.2 Online Processing

Using the `-device` option, OpenFace allows to specify a device id of a webcam to perform feature extraction from in real-time. If this option is specified, `-cam_width` and `-cam_height` followed by a number can be used to specify the resolution, which by default is 640×480.

4.2.2.3 Output

By default, `FeatureExtraction` puts its output files into a directory named "processed", under the working directory; if this directory does not exist, it is created. We can specify another name for the directory using the `-out_dir` option, followed by the path of the directory.

With the `-of` option, we can specify a path for the output files generated by the executable. If this option is not used, the original filename is considered. If `-fdir` or `-device` is used, the directory name or the timestamped webcam ID are used respectively.

`FeatureExtraction` accepts a number of options to specify which pieces of information will be computed and will appear in the output; by default (when no option is specified) all features are computed and included in the output; unless this is what we want, we should specify only the option that we need to avoid unnecessary computational overhead. The following are the ones we are interested in:

- `-aus`: output Action Units recognized by OpenFace.

- `-pose`: output head pose, including location and rotation angles.
- `-gaze`: output features relative to the gaze.
- `-tracked`: output processed video with detected landmarks.

4.2.3 Output Format

A part from the tracked video file, all information relevant to our application will be written in a CSV file.

4.2.3.1 Facial Action Units

OpenFace is able to recognize a total of 18 of Facial Action Units (see section 1.2 for a general explanation), specifically: 1, 2, 4, 5, 6, 7, 9, 10, 12, 14, 15, 17, 20, 23, 25, 26, 28 and 45. An illustration of every of those AUs can be found at [26]. The detected information is the presence and intensity of the AUs. Intensity is encoded with a number in the real interval between 0 and 5, where 0 indicates absence, 1 presence at minimum intensity and 5 presence at maximum intensity. Presence is encoded with 0 for absence and 1 for presence. Between the mentioned AUs, presence and intensity are detected for all of them, except for AU 28, for which only presence is available.

Consistency between intensity and presence information is not guaranteed, as models for detecting them are trained on different datasets [1] and, for this reason, we should rely only on one of the two. The intensity value should be always preferable to the presence when working with sequences, as the model should be able to learn what a neutral expression would look like and calibrate the predictions based on that [2]. For this reason, our application will rely on intensity information.

In the output CSV file, the AU intensity and presence are to be found respectively under the labels `AU*_r` and `AU*_c` (* is a placeholder for the AU number), where r and c stand for regression and classification.

4.2.3.2 Head Pose

OpenFace is able to recognize the location of the head with respect to camera as well as head rotations. In our application we will only consider head rotations as they are more significant from a communicative perspective. In the CSV file, these are labeled as `pose_Rx`, `pose_Ry` and `pose_Rz`; they are expressed as radians of the rotation angle around the respective axis.

4.2.3.3 Gaze

OpenFace provides information on the gaze direction. More precisely, it provides the normalized eye gaze direction vector for both eyes independently, as well as a more easy to use format that averages the information of both eyes, which captures the overall behaviour of both eyes. This last piece of information is found under the labels `gaze_angle_x`, `gaze_angle_y`, representing the eye gaze direction in radians in world coordinates averaged for both eyes.

4.2.4 Technical Considerations

Integrating OpenFace in our project is not as trivial as it may seem. Hereafter we are going to describe some technical problems that were encountered when trying to interface OpenFace with our application.

4.2.4.1 Real-time Feature Streaming

When dealing with offline processing, it is easy to get features from OpenFace: it is simply necessary to parse the CSV output of OpenFace. Getting features in real-time is not as trivial, as OpenFace does not support any mechanism to do that. Some of the issues on Github discuss precisely this topic.

The most complete solution between the ones suggested is to make use of a messaging library to stream the features in real-time to other processes. Such a solution is extremely flexible and would be capable of streaming features over the local network. This approach has been implemented for Windows, using the ZeroMQ messaging library [4]. Despite the plans of including a messaging server in the project [5] [13], these were never carried out, leaving the project without an official solution for streaming features in real-time.

As it is out of the scope of this thesis to implement real-time feature streaming in OpenFace, we eventually decided to leverage Unix **named pipes** (also knows as FIFOs) as IPC mechanism to be able to get features in real-time [10]. In this case, `FeatureExtraction` will be instructed, using the `-of` option, to write its real-time features to a named pipe rather than to a regular file. Since we are going to use the docker container, we have to put the named pipe in a shared directory, also accessible by the program that has to consume the features.

One thing we must be aware of is that `FeatureExtraction` will attempt to carry out a post-processing on the AUs values before quitting. If `FeatureExtraction` is writing to a named pipe, it will be blocked at the

beginning of the preprocessing stage in the attempt of reading from the pipe to retrieve the data to preprocess. At this point, the only way to stop the process is to terminate it. This is clearly not an elegant solution, but it allows us to reach our goals while minimizing the development effort.

Unfortunately, Windows named pipes are semantically very different from Unix ones; for this reason, this method will not work on Windows. As we will see later in section 4.3, this is not a big problem, as the rest of the application is completely independent from the way data is obtained; this concretely means that, to port the application to Windows, it is enough to use a stream that would work on Windows; one option would be getting the features streamed using the approach based on the ZeroMQ library implemented for Windows [4].

4.2.4.2 Frame Drop

During the application development we encountered some issues with OpenFace that is worth mentioning. Normally, OpenFace processes every video frame it is fed with in both offline and online processing; however, there are some situations where OpenFace ends up skipping frames.

When doing online processing, this is common in case the processing power is not sufficient to process every frame. The required computational power seems to vary depending on the video frame properties; for instance, a video with low brightness requires more computational power, hence, it is likely to lower the processing frame rate.

When doing offline processing, this problem can appear anyway, for different reasons [3]. This behaviour also seems to be associated with poor time precision of the data and wrong FPS values for the output tracked videos. It is not clear what causes this bug and in which context OpenFace will present this kind of behaviour. Most of the times we can accept the inaccuracies, but, in contexts where we need maximum precision, we can resort to a workaround: split frames of the video into separate files (e.g. JPEG files) and process all of them using the `-fdir` option. In this case, we will also need to populate timestamp data for every frame, given that OpenFace has no way of doing it, not knowing the frame rate of the video. This method has significant downsides: the video needs time to be split into its frames and the disk space occupied is much larger; moreover `FeatureExtraction` processing using the `-fdir` option seems to be slower than when using the `-f` option. For these reasons, in our application we decided not to adopt this workaround and accept potential inaccuracies.

4.3 Application

This section describes the structure of the application and how Panson components were used in its context.

Using the term "application" can be misleading, as it is not structured as a stand-alone application, but rather as a Jupyter notebook making use of Panson. The notebook can be found at [17], with the name `openface.ipynb`. Panson shapes the application; indeed, we can say that most of the application is made up of an interaction of Panson objects.

The initial notebook prototype was chronologically antecedent to Panson and therefore not based on it. The prototype was not meant to grow directly into the final, Panson-based application, but it was rather meant for experimenting with OpenFace and getting a clear idea of the technical challenges that Panson would have had to solve, as well as the limitations due to Jupyter notebooks.

The current version of the notebook is meant to explore the possibilities of facial feature sonification while providing an overview of what Panson is capable of. As such, it also contains brief tutorials about different Panson components; however, these only give an overview of Panson and are not meant to be complete documentation.

The notebook mainly focuses on real-time sonification as a way to explore different sonifications, but also provides an example of offline usage, as well as functions to convert video files to different formats and merge audio and video files using ffmpeg; these functions could be used to convert video files to MP4 so that they could be displayed inside of the notebook or to merge sonification renderings with the videos from which the facial features were extracted. This way, users can adapt the existing code for their tasks.

After an explanation of the required setup and a tutorial on how to define sonification classes in Panson, the notebook goes on with showing the usage of a really simple, test sonification in both use cases: offline and online. In the offline use case, a VideoPlayer object can optionally be used to play the video of the face along with the features previously extracted from that video. In the online use case, it is not always possible to use a RTVideoPlayer, as the webcam device cannot normally be shared between two different processes (the notebook and `FeatureExtraction`). This implies that we will not be able to record the video stream. We must notice that this kind of situation is not typical: usually each source of data is independent, while here the facial features are extracted directly from the video frame source. RTVideoPlayer is intended for contexts in which the video stream accompanies the data, but it is nonetheless independent. When used in online mode, `FeatureExtraction` will display a window showing the real-time tracked video with landmarks,

which would provide the user with visual feedback.

To specify how to obtain data in real-time from OpenFace, the application defines two Panson Stream objects:

- `openface_stream`: this object simply reads the CSV-formatted data from a named pipe and returns it. When the stream is opened, FeatureExtraction is executed in online mode as an opening hook, and terminated when the stream is closed as a closing hook. It is necessary to terminate the process, because, as we explained in 4.2.4.1, the process will block in the attempt of executing a post-processing, reading from the named pipe.

- `avg_openface_stream`: this object is like the previous one, but with an additional preprocessing step: a moving average with a window size of 10 on the AUs' intensity values. This can be useful to make intensity values less oscillating, but we have to be aware of the fact that it will slightly reduce the responsiveness of the system.

As discussed in section 4.2.4.1, this method of getting data would work only on Unix-like systems (Linux, macOS). If we want to run this application on Windows we would have first to define another stream object that would obtain the data in some other way than through a Windows named pipe. Given that a messaging server seems to be available for Windows [13], we could try to use it to stream the real-time features to our application. The stream object to implement in this case would listen for feature traffic over the network instead of reading from a named pipe. We could use this object on a Windows host to get features from the messaging server running on the same host or on a generic host to get features streamed through the local network by a messaging server running on a Windows host. Stream objects are the **input data backend** of the application and the rest of the application is independent from them.

Using DataPlayer objects, we can play the data using different sonifications at different playback rates, change the parameters of the sonification during the playback, navigate the data, record the sound produced and rendering the sonification of the data from the beginning to the end in an audio file.

4.4 Sonifications

The notebook implements a number of sonifications as Panson Sonification subclasses. These sonifications use different approaches and take into

account different features, therefore they can be a good starting point for further development of more articulate sonifications. At [17], **live demos** of interactive sonifications can be found under the directory `demos/`.

For each sonification described, we will discuss the idea behind it, its implementation, its parameters and some related ideas. Custom synthesizer definitions will be reported and discussed. The best way to learn about these sonifications is to try them out in the notebook and experiment changing their code, and the reader is encouraged to do so. Many of these sonifications can be effectively used together: this can be done using a GroupSonification object.

4.4.1 AU04Continuous

This sonification is just a simple test meant to demonstrate the implementation of a very simple Sonification object using facial features extracted with OpenFace. It sonifies the intensity of AU 4 (Brow Lowerer), using its value to regulate both amplitude and frequency of a continuous synthesizer; as synthesizer, we adopted s2, the default continuous synth shipped with sc3nb. Even if default synths are loaded automatically at server's startup, the sonification is required to send anyway the instructions to load them; if this is not done, the export method of the DataPlayer class will not work properly. This is because default synths are not loaded automatically on non-realtime mode servers.

The full AU intensity range [0,5] is mapped into the MIDI range [69,81] (one octave). This means that if the tracked face will lower its brows, the generated sound will increase in pitch. To make the synthesizer sound smooth, we used a lag time coherent with the expected frame rate: as we usually expects a frame rate of 30 FPS, we used an hard-coded value of 0.03 seconds as lag time. This means that the pitch will smoothly slide to its value during a 30 ms lapse. Even if this introduces additional latency, a 30 ms difference is not enough to be noticed by the user and affect the interpretability of the sonification.

The intensity range [0,1] is mapped to the amplitude range [0,0.3]. This means that the maximum amplitude value is 0.3 and is used for all intensity values that are greater than 1, which is the AU presence threshold; this approach allows to fade the sound in and out and provides some information on OpenFace intensity prediction in the range [0,1]. The sonification has also a parameter "amp" that can be used to scale down the amplitude value obtained from the intensity mapping. This parameter is in the intensity range [0,1] and is rendered in the notebook as a slider.

While processing every data sample, the intensity value of AU 4 is ac-

Figure 4.1: Widget rendering of AU04Continuous Sonification.

cessed by looking up the key "AU04_r" on the series containing the data passed to the processing method.

4.4.1.1 Discussion

This sonification can effectively be used as a first example to familiarize with the notebook. It shows how to access OpenFace features and how to implement a simple mapping to amplitude and frequency. The user can get an intuitive idea of how precise OpenFace predictions are by trying the sonification out in real-time and listening to the auditory feedback generated. This sonification does not serve any other practical purpose and is not relevant for any of the use case considered.

4.4.2 Upper Face Musical Sonification

The main idea behind this sonification approach is to allow to control a musical instrument using facial expressiveness. In this specific case, the user will be able to control the execution of notes on a minor pentatonic scale using the upper part of the face. More precisely, we are considering the following AUs:

- AU 1: Inner Brow Raiser
- AU 2: Outer Brow Raiser
- AU 4: Brow Lowerer
- AU 5: Upper Lid Raiser
- AU 6: Cheek Raiser
- AU 7: Lid Tightener

An illustration of these AUs can be found at [26].

AUs that are located lower in the face are mapped to lower pitches and AUs that are located higher to higher pitches. When AUs are relative to the same area, an arbitrary order is established. AUs are mapped from the

fundamental of the minor pentatonic scale up in the following order: 6, 7, 5, 4, 2, 1. AU 1 is mapped into the octave of the fundamental.

This kind of sonifications have two parameters: one for regulating the amplitude and the other for changing the fundamental note, hence the key of all the scale. While the first one is rendered as a slider that takes its values in the interval [0,1], the second one is rendered as an integer slider that can take only MIDI note numbers as its values.

4.4.2.1 PentatonicContinuous

The first sonification makes use of sc3nb's s2 continuous synth. At start time one synth is started for every tone of the scale with zero amplitude. For each data sample, the amplitude of all synths is modulated according to the intensities of the corresponding AUs.

An AU intensity value lower than 1 is mapped into an amplitude of zero. Intensity values greater or equal to 1 are mapped on the decibel interval [-20, -5] (corresponding to the amplitude range [0.1, 0.56]). The decibel scale is used respect the logarithmic perception of amplitude of the human ear and give instead an impression of linear variation.

Figure 4.2: Widget rendering of PentatonicContinuous Sonification.

4.4.2.2 PentatonicMda

The second sonification is an attempt to experiment with an event-based approach using a piano synth implemented with the MdaPiano UGen [16]. This UGen is not part of SuperCollider and it has to be installed as an extension.

The synth simply triggers the execution of the piano note and release the synth when its volume is under a certain amplitude threshold (hard-coded to 0.05). The release of the synth is triggered abruptly by the DetectSilence UGen: it is a very simple method, but it may not be the best solution in case the quality of the sound is really important, as the sudden release of the synth may introduce artifacts.

```
SynthDef("mdapiano", { | freq=440, vel=100, amp=1 |
```

```
    var piano = MdaPiano.ar(
        freq,
        1,
        vel,
        decay: 0,
        release: 0,
        hard: 0,
        stereo: 0,
        mul: amp
    );
    DetectSilence.ar(piano, 0.05, doneAction:2);
    Out.ar(0, piano);
})
```

Sonifying continuous intensity values using an event-based approach presents some challenges. First of all, we have to decide under which conditions to trigger events. Secondly, we would like the sequence of triggered sounds to provide useful information regarding the dynamic behavior of intensity values, which is something that would be more easily accomplished with continuous sonification.

We established four thresholds that divide the intensity range in five levels. These thresholds are 1, 2, 3 and 4. Every time that the intensity of an AU changes level (increasing or decreasing), an event is triggered with its parameters set according to the new level. The intensity values can oscillate quite a lot. When they oscillate continuously around the specified thresholds, they will continuously retrigger the same event over and over, causing a noticeable amount of unnecessary noise. One way to alleviate this problem in the online use case is to use `avg_openface_stream` as stream object; the moving average performed on the data as preprocessing will make it significantly less oscillating.

4.4.2.3 Discussion

The two sonifications just presented show how multiple synths can be handled with both continuous and event-based approaches. None of them was specifically designed to address any of the use cases discussed, but the idea behind could be applied to support the practice of facial expressiveness. Nonetheless, by quickly testing the sonification in real-time, it is easy to notice how different sounds correspond to different expressions. Users could try to generate the same musical sounds of someone else and match the sounds. The continuous approach is more suitable than the event-based one

for this kind of task, as it can directly map the intensity of AUs to the amplitude of different tones. An event-based approach discretizes the AUs information, which is something undesirable in a context in which we want to put emphasis on intensity changes.

A direct mapping of intensity to amplitude is no warranty of a good interpretability, as amplitude is difficult to compare for the ear. This is not a problem if the goal is to capture the overall behaviour of the sonified part of the face, where an "imprecise" amplitude mapping could switch the focus from the exact values of AUs' intensities to a more global view. In this sense, we could say that this would allow to compare different faces using sonification (R2.8).

Using the same concept of continuous, musical mapping, we could think of sonifying other areas of the face or sonifying together AUs that are related; in this last case, an idea would be to design an "harmonic" mapping that would associate related AUs as notes of a same chord.

This kind of sonification is surely not appropriate as a support for visually impaired people in a conversation, as it is very invasive. It also makes it difficult to sonify information on a big number of AUs, as the result would be too chaotic.

4.4.3 DropBlink

This sonification sonifies blinking (AU 45) using a synthesized drop sound. As blinking is intuitively conceivable as a binary event on/off, it could seem a good idea to use presence value rather than intensity, but in practice using the intensity has proven to be much more accurate, for the reasons we mentioned in 4.2.3.1. OpenFace considers the blinking active when the relative intensity value is greater or equal to 1, so this sonification will trigger the drop sound when this threshold is exceeded.

Oscillating intensity values can cause problems also in this context and in general in all contexts where an event-based approach is adopted. To mitigate the effects of oscillating features, instead of using a single threshold set to 1, we establish two different thresholds for activation and deactivation of blinking. The drop sound will be triggered when blinking is activated. The default values of these thresholds are 1 and 1.6: this means that blinking will be switched on only when 1.6 of intensity is exceeded while the blinking was off and it will be switched off only when the intensity gets lower than 1 while the blinking was on.

These thresholds are parameters of the sonification, hence they can be changed while the sonification is running. The most appropriate way to render them in the notebook is using a FloatRangeSlider ipywidget. This

widget allows to position threshold values on the same slider; this means that the widget will enforce the limitations that the lower threshold cannot be greater than the higher one.

Figure 4.3: Widget rendering of the DropBlink sonification.

Sometimes, intensity predictions can get quite inaccurate when performed in real-time. It is interesting to notice that, if the user keeps their eyes closed for some time, OpenFace will perform some sort of calibration and improve prediction precision.

4.4.3.1 DoubleDropBlink

Instead of sonifying only the closing of the eyes (blinking activation), we can also think of sonifying the opening (blinking deactivation) using a different sound. In this case we use a drop sound with a pitch that is a perfect fifth (seven semitones) higher than the other. A higher pitch naturally relates to the idea of opening of the eyes, making the sonification easily interpretable.

In the notebook, this sonification is implemented as a subclass of the previous one. A more appropriate way to implement it could be using a unique class endowed with a checkbox parameter to specify if the opening of the eyes has to be sonified or not.

4.4.3.2 Discussion

These sonifications seems to work in a in a satisfactory way. The adjustable AU activation/deactivation thresholds are an effective way to regulate the sensitivity of the sonification. If activation and deactivation thresholds are very distant, the sonification can minimize false positives, but it will also have trouble in recognising quick blinking sequences.

Both sonification could be useful to support visually impaired people in their interaction if used along with other sonifications (R2.1, R2.2, R2.3). By their nature, these sonifications are not thought for supporting the practice of facial expressions, because blinking is not an interesting element to practice. The only feature of blinking that could be practiced is its timing, but its accuracy is totally dependent on OpenFace.

Testing these sonifications in real-time, we found out that it is immediate to associate blinking to the drop sound and get used to it. From time to time, the sonification will make users aware of their own unconscious blinking.

Sonifying both closing and opening of the eyes, the user will also be able to understand if the eyes are being kept closed for a prolonged amount of time. However, this is not necessarily the best way to convey this information, as sonifying both events can be quite intrusive and distracting. If we are also sonifying gaze, another possible way of conveying this information would be muting the gaze sonification when eyes are closed.

4.4.4 Multi-percussion

In this section, we will describe two sonifications that have been implemented using an event-based approach based on samples. The event-based approach used is similar to the one we used in the event-based sonification based on MdaPiano (4.4.2.2). The intensity range is divided into five levels and samples are played when these levels are crossed, using different parameters.

These sonifications consider all AUs except for AU 45 (Blinking) and AU 28 (Lip Suck). The reason for this is that AU 45 has already been considered in the previous section and OpenFace does not provide intensity information for AU 28. We could think of using its presence information, but, as we have seen, it is supposed not to be very reliable when processing videos; for this reason, we decided not to include it in this sonification. Later on, we could design a separate sonification only for AU 28 and merge it with other sonifications using the GroupSonification class.

In designing the sonifications, we considered some guidelines to ensure a degree of clarity and interpretability:

- Samples that typically play together should be easy to distinguish.
- AUs that appear more often should be associated to samples that are not very intrusive.
- AUs of a certain area of the face should be associated with samples that are somehow related.
- High pitches of samples are high in the face.

All samples used were obtained from freesound.org and modified using Audacity. Samples were modified to be shorter and mono. In general, we want samples not to be very rich and complex, but to be just enough complex

to be clearly distinguishable. All buffers used to contain samples are allocated in the initialization phase and they can be released calling the "free" method.

The notebook ([17]) contains a description of all the associations between AUs and samples, with a widget to reproduce the samples directly in the notebook. In general, we associated high-pitched bell samples to the eyebrows, snares to the eye area, different metallic sounds to the lip area and low percussive sounds to the lower part of the face. AU 9 (Nose Wrinkler) and AU 14 (Dimpler) are associated to very peculiar samples that do not seem related to the other AUs.

4.4.4.1 Percussive

The first sonification plays samples with an amplitude that is derived from the level of the intensity of the respective AU. Intensity values are mapped on a decibel scale so that the variation is perceived linearly. This method is not very precise, as it is difficult for the human ear to compare amplitudes. Nonetheless, it would hopefully be easy for a user to intuitively perceive the general direction of the intensity of the AUs (increasing or decreasing).

This sonification seems to have practically many problems. First of all, no countermeasures were adopted against the oscillation of AUs' intensity values. This implies that a lot of samples are replayed unnecessarily, causing a lot of noise and making sometimes hard for the user to distinguish different samples; this happens also when the user seems to maintain a stationary expression. Secondly, this kind of approach results kind of hard to interpret as intuitively a user would associate the sample with the activation of the respective AU, which is not always what happens in this case. For example if an AUs stays stationary in a certain level for a certain amount of time, it will not generate any sound. If then its intensity starts decreasing, when crossing the level threshold, the associated sample will be played. This can be quite confusing as the user may think that the triggered sound could be due to an activation of the AU. To solve this problem, we can somehow encode the direction information in the generated sounds.

4.4.4.2 DirectionalPercussive

The second sonification improves the previous one in different ways. The same samples of the previous sonification are played with different properties depending on whether the intensity of the associated AU is increasing or decreasing. Increasing intensities make samples' pitch bend up, while decreasing intensities make samples' pitch bend down. We can also optionally specify to use panning to play decreasing samples on the left speaker and

increasing samples on the right speaker. A checkbox parameter can be used to enable or disable this behaviour, as we can see in image 4.4. To counter feature intensity oscillation, we adopted a similar approach to the one used in the blinking sonification. This time, activation and deactivation boundaries have to be specified for every threshold that would trigger a sample playback. This sonification has a "bounds" parameter that allows to specify relative boundaries that are applied to all thresholds.

The synth used allows to play a sample with a constant rate until a certain point (breakTime) and, starting from there, linearly change the rate until the end, where the final rate (rateFinal) will be reached. This will create the bending effect on the sample pitch. Values for breakTime and rateFinal are set individually for each sample.

```
SynthDef("playbuf_bend",
    {| out=0, bufnum=0, rateInitial=1, amp=1, pan=0, breakTime, rateFinal |
    var sig, rate;

    var rateAvg = (rateInitial + rateFinal) / 2;
    var sampleRateAvg = rateAvg * BufSampleRate.kr(bufnum);
    var breakFrame = breakTime * BufSampleRate.kr(bufnum);
    // mono signal: frames = samples
    var remainingFrames = BufSamples.kr(bufnum) - breakFrame;
    // calculate remaining time with dynamic rate
    var remainingTime = remainingFrames / sampleRateAvg;

    rate = EnvGen.kr(
        Env(
            [rateInitial, rateInitial, rateFinal],
            [breakTime, remainingTime]
        )
    );
    sig = PlayBuf.ar(1, bufnum, rate*BufRateScale.kr(bufnum), doneAction:2);
    Out.ar(0, Pan2.ar(sig, pan, amp));
})
```

4.4.4.3 Discussion

The second sonification seems to work quite well in practice. The "bounds" parameter is an effective way to regulate the amount of excitability of the sonification; if it specifies a large range (e.g. $[-0.6, +0.6]$) the sonification will be reasonably quiet, but still responsive to movement (R2.3).

Figure 4.4: Widget rendering of the DirectionalPercussive Sonification.

This sonification encompass the whole face. With some training, the user should be able to correctly interpret the sonification. The training can be carried on interactively using the webcam, so that the user would hopefully be able to experiment using their own facial expressions and learn to correlate sounds with their own proprioception (R2.1). Practical evaluation is necessary to confirm this.

This kind of sonification could be used as a support verbal interaction with visually impaired people. It is likely that, after some time of usage, users would get accustomed to the sonification and that it will not be very distracting during the interaction; hopefully users will be able to interpret the sonification without consciously thinking about it.

With this kind of sonification, it is difficult to compare AU intensities, as intensities are mapped to amplitude, which is difficult for the human hear to precisely compare. Moreover, the fact that we are using an event-based approach, maps the layer level to the amplitude rather than the precise intensity value. This makes this sonification inappropriate for practicing facial expressiveness. Nonetheless, it could be anyway useful in case we are interested in comparing the timing of facial expressions.

Bending the pitch of samples does not work well with all kind of samples: percussive samples are not very suitable for this approach and the difference is hard to perceive. One possible solution to this could be using the same sample at different frequencies, without changing its rate during the playback. This is best done modifying the sample by shifting its pitch using a program such as Audacity, so that we will have two samples at different pitches. It is also possible to accept this limit, as We are not necessarily interested in sonifying the full range of intensity of every AU. We could think of sonifying some AUs, such as AU 5 (Upper Lid Raiser) or AU 7 (Lid Tightener), using an approach that is more similar to the one used for blinking sonification, considering as only threshold the intensity value of 1.

4.4.5 Head

When sonifying the head movements, we are interested in sonifying the head rotations rather than its location with respect to the camera. OpenFace outputs this information under the labels pose_Rx, pose_Ry and pose_Rz.

The first attempt in sonifying this information simply uses the s2 continuous synth and maps every rotation angle to different parameters of the synth. More precisely:

- Rotations around the x axe (yes movement) are mapped to the pitch.
- Rotations around the y axe (no movement) are mapped to the panning.
- Rotations around the z axe are mapped to the overtones number.

For all rotation angles, we consider the range $[-45°, +45°]$. Angles greater than 45° are theoretically possible, but they are not relevant in a normal conversational context; for this reason, any angles greater than 45° are mapped to the maximum value of the destination range.

The mapping of rotations around the z axe to the number of overtones is arbitrary, and the user has to learn it as it is. Moreover, the number of overtones is an integer number, which means that the angle or rotation around z will be perceived in a discrete manner. We could think of solving this problem by using a big number of overtones as upper limit, but when the number of overtones is high, it is difficult to perceive the difference in timbre. Here we used a maximum number of overtones equals to 7, as it seems like a good compromise.

This sonification is always playing, even when the head is in a neutral position, which can come off as annoying, especially when other sonifications are playing.

4.4.5.1 SilentHead

We improved the previous sonification by making sure that it would not generate any sound when the head is in a neutral position. Every axe has an independent silence threshold settable using using a slider parameter: if all rotation angles are lower than this value, no sound will be generated. Indeed, the biggest rotation is mapped to the amplitude linearly (i.e. without using a decibel scale). A linear mapping has the effect of making quiet sounds more distinguishable.

With this sonification an important question emerges: how to determine what is to be considered the neutral position of the head? Using this sonification, it appears evident that using zero as static neutral value for each

of the angles is not a good solution: this would consider the position of the head the more neutral the more it is facing the camera. This solution is not good for various reasons. First of all, it is not guaranteed that the camera will always be placed precisely in front of the user; this is especially true considering that cameras are frequently placed at a height that is different from the head height, e.g. capturing the face from a higher position. In these situations, the neutral position of the user's head will not naturally face precisely the camera. Secondly, people can assume different head postures during a conversation and find them comfortable in different moments. A more appropriate way to handle amplitude mapping in this context would be having the amplitude decaying with time when the head maintains a certain posture. Head movements would revive the sonification by increasing the amplitude proportionally to the movement amplitude. This way, whenever a user would keep his head still for some time, the sonification will become quiet, thing that intuitively evoke an idea of stillness.

Figure 4.5: Widget rendering for the SilentHead Sonification.

4.4.5.2 NoisyHead

The sonification described here uses **subtractive synthesis**. The three rotation angles are mapped to parameters of synth that generates band-pass filtered noise. This time, rotations around the z axe will be mapped to rq, which is the reciprocal of the quality; this is a measure of the bandwidth of the filter. By changing this argument, we can obtain a clearer tone where the central frequency is clearly identifiable or a more noisy kind of sound.

```
SynthDef("bpf_noise", { | amp=1, pan=0, lg=0.5, freq=440, rq=0.2 |
    var sig;
    sig = PinkNoise.ar(amp);
    sig = BPF.ar(
        sig,
        freq.lag(lg),
        rq.lag(lg),
```

```
        // when a bandpass filter narrows, the amplitude decreases
        // this will balance it
        1/rq.sqrt.lag(lg)
    );
    Out.ar(0, Pan2.ar(sig, pan));
})
```

This sonification does not take into account any concept of neutral position, hence does not modulate the amplitude; nonetheless. the noise sound that produced, if played at low volume, can be effectively used in the background to provide a continuous feedback regarding the position of the head. Rotation angles are mapped linearly to the synth's arguments, exception made for rq which is mapped exponentially to the interval [0.01, 1].

One possibility is to use other kinds of mapping; it would be interesting to use mapping functions that could highlight the neutral head position by changing the parameters more around the neutral position. As we discussed earlier, we would first need a meaningful way of determining the neutral position.

Figure 4.6: Widget rendering for the NoisyHead Sonification.

A "mirror" checkbox exists to mirror the sonification mapping. This is useful for switching from a situation where the user is sonifying their own expressions to one where is perceiving the sonification of someone else's expressions.

4.4.5.3 Discussion

Sonifications based on the s2 synth appears pretty annoying to hear even if their amplitude is modulated with respect to a neutral position, making it silent most of the time. As we discussed earlier, using a static neutral position, is not an appropriate solution. The neutral position has to be dynamic and adaptive, so that a maintained static pose would make the

sonification silent, while movement would excite the sonification and bring attention to the head movement.

The noisy sonification can nicely work with other sonifications as the generated noise is clearly distinguishable, but without being too intrusive (R2.2, R2.3). The sonification overall is easy to interpret when testing it, but not necessarily in an actual conversation. Most of relevant head gestures (e.g. yes and no gestures) only need minimum movement when done naturally. Possible solutions to this problem could be found experimenting with event-based sonifications designed specifically to detect and sonify the interesting gestures or using continuous sonifications adopting mapping functions designed to emphasize movements around a neutral position.

A sensitive next step in the development of head rotations sonifications would be implementing sonifications that adopt the methodologies described in [36].

4.4.6 Gaze

To sonify gaze we use the information found under the labels `gaze_angle_x`, `gaze_angle_y`. This is the eye gaze direction in radians in world coordinates averaged for both eyes. The two simple sonifications that we implemented use both s2 to map `gaze_angle_x` to the pitch and `gaze_angle_y` to the panning.

The first sonification implemented is simply what was just described, considering 90° as maximum absolute value for gaze angles.

The second sonification is similar to the first one, but adds the concept of neutral position as we saw it in the head rotations sonification. This time it is easier to establish a neutral position for the eyes. Two possible options are:

- Gaze directed at the camera. Both the angles are close to zero.
- Gaze directed at the screen. The values of the angle would depend on the camera positioning with respect to the screen.

In our sonification, we chose the first. Both angles have a silence threshold that can be regulated with a slider parameter.

4.4.6.1 Discussion

This sonifications are only appropriate as examples to show how to sonify gaze information and more work is needed to get to useful sonifications. From these, we learn that gaze angles are predicted with a decent accuracy

Figure 4.7: Widget rendering for the SilentGaze Sonification.

only with really good scene brightness. In any case, OpenFace seems to underestimate angle values; one countermeasure to take in this context is to take a small enough angle as maximum possible value.

What we can learn from these sonifications, is that, for gaze sonification in a conversational context, using `gaze_angle_x` and `gaze_angle_y` is enough and we would not normally need to consider other gaze features.

Using zero gaze angles (gaze directed towards the camera) as neutral position seems to work well, but only with our setup where we are using an integrated webcam. In general, we need to experiment with using as neutral position angles that specify a gaze directed towards the screen; it would be nice to have an automatic mechanism for calibration that could calculate this angle.

In a practical context, we also would have to consider the idea of silencing gaze sonification when eyelids are closed, i.e. when blinking AU intensity is greater than 1 for a prolonged amount of time. Assuming that the synth used generates a clearly distinguishable sound, it would be possible to deduce when eyes are closed by the absence of gaze sonification. The sonification of the activation of blinking, e.g. with a drop sound, would still be useful, but we could avoid sonifying the deactivation of blinking.

4.4.7 Smile

The smile seems to be in many context the most important source of information. We can design sonifications to focus specifically on this element and put it in the foreground. OpenFace recognises various AUs related to the smile:

- AU 6 - Cheek Raiser
- AU 12 - Lip Corner Puller
- AU 14 - Dimpler
- AU 15 - Lip Corner Depressor

- AU 17 - Chin Raiser

In general, we expect pairs of AUs 6 - 12 and 15 - 17 to be correlated. AU 14 seems to be an independent expression, not really correlated with the other ones and, for this reason, in this sonification we chose to ignore AU 14.

The adopted synth generates a sequence of blips of a certain density and applies reverb to it.

```
SynthDef("discrete_rev",
    { | amp=1, freq=440, density=2, mix=0.5, room=0.5, damp=0.2, lg=0.1 |
    var trig, sig, env;
    sig = SinOsc.ar(freq);
    // transform signal into short blips
    trig = Dust.kr(density);
    env = EnvGen.kr(Env.perc(0.001, 0.05), trig);
    sig = sig * env;
    sig = FreeVerb.ar(
        sig, mix.lag(lg), room.lag(lg), damp.lag(lg), amp.lag(lg)
    );
    Out.ar(0, sig!2);
})
```

In this sonification AU 12 and 15 are considered as "leading" expression, in the sense that they govern the expression of their related AUs (6 and 17) in sounds. AU 12 and 15 are antagonists and are assumed not to be detected together. When one of those is detected, its intensity is mapped to the pitch of the sound, while the intensity of the related AU is mapped to the mix argument of the reverb (dry/wet balance). The density of the blips is derived from the maximum intensity of the two AUs considered.

The effect of this sonification is not easy to understand without an example. If a user starts smiling (AU 12) the blips will increase in pitch and density; if the user also raises their cheeks (like it should happen in a true smile) the wetness of the reverb would increase noticeably.

Figure 4.8: Widget rendering for the DustSmile Sonification.

4.4.7.1 Discussion

Its unusual mapping makes this sonification interesting. Testing this sonification in real-time, we found the mapping appears quite intuitive to interpret, even if a practical evaluation is needed. If used as a support for visually impaired people, this sonification could provide important unobtrusive feedback regarding the smile. While this seems to work well for AUs 6 and 12 (Cheek Raiser and Lip Corner Puller), AUs 15 and 17 (Lip Corner Depressor and Chin Raiser) are detected clearly only if they are exaggerated. To make the sonification more balanced, we could think of designing a mapping that disproportionately emphasizes the intensities of AUs 6 and 12.

The sonification is not very intrusive, but manages anyway to provide continuous feedback on the AUs intensities (R2.3). For this reason, it would be interesting to test how this sonification would interact with other, more "global" sonifications, such as the one described in 4.4.4.2, ignoring the AUs covered by the smile sonification.

Regarding facial expressiveness practice, it is difficult to apply this sonification to that use case. AUs intensities are mapped to the density of blips and to the wetness of the reverb, both of which are properties that are difficult to compare precisely, making this sonification inappropriate for facial expressiveness practice.

Chapter 5

Discussion and Future Work

The facial feature sonification application described in the previous chapter shows how Panson can be effectively used to easily develop interactive sonification applications. Panson has proven itself capable of providing support for the expected features of the application without placing any limitation. The application leverages Panson's components for most of the features and only implements its own sonifications and data streams. In the following sections we will discuss the achieved results and possible future developments relative to both Panson and the facial feature sonification application.

5.1 Panson

Panson at its current status of development solves the main technical problems we had set out during requirement analysis and traces the path for further refinement and development. Some of the future work to be carried on has already been mentioned in sections 3.1, when appropriate. Here we are going to consider what has been achieved so far, what are the existent problems and how the framework could develop to solve those problems and add new relevant features.

5.1.1 Reflections on the Original Requirements

So far, Panson managed to satisfy the most important requirements established. It is provided with a relatively stable API, despite being still a young project. Its components are easy to connect together and provide support for interactive usage in Jupyter notebooks. Through Stream objects, it provides support for working with different data sources, achieving our goals regarding its **reusability** for different aims. The framework supports **re-**

producible sonification by making available functionalities to record and replay real-time data, along with an optional video stream.

- **R1.1**: support offline data sonification is provided through the DataPlayer class.
- **R1.2**: single stream support online data sonification is provided through the RTDataPlayer, while multi-stream support through RTDataPlayerMT and RTDataPlayerMP.
- **R1.3**: support for recording sonifications is provided through the "record" function of data player classes.
- **R1.4**: support for logging data coming from real-time streams is implemented in RTDataPlayer, RTDataPlayerMT and RTDataPlayerMP.
- **R1.5**: support for playback of logged data using different sonifications is provided through the DataPlayer class.
- **R1.6**: support video files and stream is provided respectively through the VideoPlayer (even if it is still a prototype) and RTVideoPlayer classes. Audio files and streams are still not supported by the framework. This is discussed in section 5.1.3.
- **R1.7**: applications can work with different kinds of data as long as they can be represented in a pandas Series object.
- **R1.8**: support interaction in Jupyter Notebooks is provided by providing many components of the framework with an ipywidget-based representation and by providing a library of widget parameters to interact with sonifications.
- **R1.9**: the framework does not use any platform specific tool or function; nonetheless, extensive testing on different systems still has to be carried on.

With regard to further development goals, there are mainly two ways of identifying them:

1. Analyzing the weaknesses of the current project and design solutions to the emerged problems.

2. Implementing other applications relying on the framework. This would make clear which important features are missing and would provide a first testing field for the development of those features.

   Some good candidates are the systems described in section 1.3.1. Of these, we find that none of them involves the usage of audio data streams; one system using this feature is InfoDrops: a sonification system for sonifying water consumption in the shower [32]. This system records the sound of the water drops and augments it using sensor data regarding water and energy consumption.

5.1.2 Source Organization

Surely, it will be necessary to rethink the structure of the project repository. Panson will have to be packaged to be later on published on PyPi, so that it would be easily installable using pip (Python's package manager). Before doing this, we have to consider which components belong to Panson and which are to be moved in independent repositories. For instance, everything related to the facial feature sonification notebook will have to be moved in another repository, but this is not the only component that does not belong in the Panson project.

We already mentioned the idea of moving the offline video player class (VideoPlayer) into its own independent project, in order to make it grow into a general-purpose and flexible video player adoptable by different projects. While this idea needs to be considered carefully before investing time in the development of the new project, the "editor" magic can be moved into its own independent project without too much thought; even if this component was developed in the context of Panson, it is not bound in any way to the project and provides a functionality that is only related to Jupyter notebooks and nothing else.

After having cleaned up the source tree from code unrelated to the framework, it is necessary to improve the documentation. A part from improving inline documentation of Python code, we can add some example notebooks (similarly to what is done in sc3nb repository [12]). These notebooks should contain tutorials of different components of Panson and they should be accompanied by sample data that could be sonified by the user using some pre-made sonifications; the user could experiment by trying to modify the sonifications.

An important element that is currently missing in the project is a suite of automatic tests. Testing a framework is not as easy as testing an application. The best way to do this would be to use unit testing with components that

are independent (e.g. video players) and implement simple testing applications to perform tests on to be able to test the framework in its ensemble. Unfortunately, there is no way to test automatically the behaviour of Panson in a notebook: this will have to be tested manually. For this purpose, there is already a testing notebook in place called `tests.ipynb`. In depth testing should be carried out to make sure that Panson is truly cross-platform. Given that it relies mainly on cross-platform tools and libraries for its interaction with the host system, this should be true, but extensive tests are necessary to verify that, focusing especially on the interaction with the file system and multi-processing.

5.1.3 Technical Issues

Some components of the framework are currently implemented as prototypes and proofs of concept and it is already clear how they need to be improved.

- The **Preprocessor** class only allows to specify simple and non-parametric preprocessing mechanisms, as no argument for its constructor is supported. This problem was discussed in section 3.1.3.

  We also need to carry on some in-depth performance evaluation, to make sure that preprocessing operations are handled by the framework with maximum efficiency.

- **Data Player's export method** creates a clone of the sonification object using shallow copy, as discussed in 3.1.4.1. This method works correctly only if all attributes of the sonification objects are instantiated in the starting step of the sonification, so that new objects are created for the clone. Even if this does not seem to be a great limitation now, it is likely that it could generate unexpected behaviours with sonifications that do not respect this limit. For this reason, it is better to rely on a well-defined deep copy operation to clone the sonification object.

The current state of Panson does not implement all the features that would be desirable in such a framework:

- Panson should be capable of handling **audio files and streams** in a similar way to videos. Analogously, two classes can be defined: AudioPlayer and RTAudioPlayer. Unlike video players, audio players could have a widget-based representation in a Jupyter notebook.

  A very important difference with video players would be dictated by the fact that it would be desirable to have the possibility of augmenting

the audio stream depending on the data. For example, we would like to be able to support use cases such as the one of the already mentioned sonification system InfoDrops [32], where the recorded sound of shower water drops is augmented based on sensor data. To support this feature, we need to design audio players carefully.

- Panson currently does not provide any support for **synchronous plotting of custom markers** on videos. The idea is to make the framework capable of displaying markers on the video (offline or real-time), where marker coordinates are computed from data values. Supporting this feature would require more than simple changes to video player classes as we would need to establish a way to efficiently communicate plotting information to processes running the video players.

- When sonifying offline data, we could want to display a window of data feature values around the current one. This is not possible using RTFeatureDisplay, which displays the history of the last played values. A **FeatureDisplay** class for exclusive offline usage should be implemented. A more in-depth discussion can be found in section 3.1.5.

Playing multiple sonifications simultaneously is supported in Panson only if every sonification interacts exclusively with its own synthesizers. This is true with most of the operations of creation and parameter update, but there could be problems when freeing synthesizers; the sonification would have to free only the synthesizer it has previously allocated: this means never freeing all the synthesizers. Currently, Panson frees by default all synths belonging to the default group associated to the SuperCollider client, but, in light of what we said, we can conclude that this is not a sensible behaviour. One way to handle this, would be to remove this default behaviour and leave all the responsibility to the user, which has the advantage of being a very simple and explicit approach. It is important to notice that the same sonification object cannot be played simultaneously by multiple data players: it is possible to do so only by using multiple instances of a same sonification class, one used by each data player. This has the downside that it is not possible to share sonifications' parameters in these kind of contexts. Shared sonification parameters would have been useful to support the "Support for Facial Expressiveness Practice" use case of the facial feature sonification application, as it would have been possible to simultaneously twist the parameters of the real-time sonification (run by RTDataPlayer) and of the offline one (run by DataPlayer).

For a second version of Panson, we could consider the use of the **Streamz** [23] library to strengthen the real-time processing. The adoption of this li-

brary would introduce only a minimal overhead and allow the user to specify modular processing pipelines. Streamz also allow to perform sophisticate operations on the flow of data, such as forking it: this is not possible elegantly using an approach based on generators as the one currently adopted. Regarding this, the challenging part is structuring Panson's API in such a way that it is easy to use. Before investing development time in this, it would be a good idea to look for actual use cases where this kind of functionality would prove itself useful to be sure that the effort is justified.

5.2 Facial Feature Sonification

The implemented sonifications allow to show many Panson's features and are a good starting point for further development, but they fail in implementing a complete set of sonifications for the established use cases. Discussions relative to the individual sonifications have been laid out on the various "Discussion" sections regarding each sonification along with some possible future developments; this section will deal with discussing the application in general and its possible future developments.

As we discussed in section 5.1.2, the application should be moved to in its own, independent project. Doing this, implies that it is no longer necessary to include in the notebook sections containing Panson tutorials and explanations, as the Panson project will have to develop its own documentation, having the effect of reducing the total size of the notebook and making it more manageable. In the application repository, it would be useful to provide some offline sample data along with the videos from which those features were extracted using OpenFace. Using offline data, the user could immediately test sonifications without having to install and setup OpenFace.

5.2.1 Reflections on the Original Requirements

5.2.1.1 Use Case 1 - Support for Visually Impaired People

The implemented sonifications are to be intended only as a starting point for the development of practically useful sonifications; to achieve this goal, practical experimentation and evaluation is needed. This can start with the testing of different sonifications from multiple visually impaired subjects (it may be enough to make subjects close their eyes). Subjects could first sonify their own facial expressions in real-time, so that they could learn how generated sounds correlate with their proprioception. After some practice, they could then switch to use it in a conversation with somebody else.

The most promising sonification concerning this use case is the DirectionalPercussive (4.4.4.2, which is capable of sonifying most of Facial AUs. After the feedback of practical evaluation, this sonification could be refined to be effectively used to support visually impaired people in their conversations.

5.2.1.2 Use Case 2 - Support for Facial Expression Analysis

Support for this use case is fully provided by Panson. In some cases it might not be possible to use the RTDataPlayer class and consequently it might not be possible to record the video stream. This topic is treated in section 5.2.2.

5.2.1.3 Use Case 3 - Support for Facial Expressiveness Practice

Designing sonifications to be used as tools to practice facial expressiveness has been the most challenging use case. Regarding this use case, results were not satisfying, and a lot of work has to be carried on as concerns both development and evaluation. Part of the difficulty is that we have to sonify a big amount of information in a way that is clearly perceivable and comparable by the user. While this could be easy if we are only taking into account a specific section of the face, it is not that easy in the general case. Moreover, we have to take into account the fact that OpenFace realtime predictions are less precise than offline, post-processed predictions. Beside this, it is not clear how comparable can features extracted from different faces be, but the best we can do is to sonify them in a way that its easy for the ear to compare, maybe filtering the data with low confidence away.

A more in-depth study about how to design sonifications for comparability is needed. In general, it is easy to think how to compare gaze, head rotations and a small group of AUs; it is not as easy to design sonifications that aim to sonify all possible AUs at once in a comparable manner. In a first moment of the study, it would be better to focus on sonifying only sections of the face rather than the whole face. It is worth noticing that some sound features, such as frequency and panning, are easier to compare than others such as the amplitude.

It is important to notice that Panson support different sonifications to be played at the same time. This is relevant for example in case we want to run a real-time facial feature sonification along with an offline one. If this is done with some visual feedback (possible using the VideoPlayer and RTVideoPlayer classes), the user has the possibility of imitating facial expressions using both visual and auditory feedback. The way to do this, is to create two separate Sonification objects for the DataPlayer and RTData-

Player object. By default, when stopping the sonification, Panson, frees the group associated with the SuperCollider client; we have to override this behaviour by defining a stop method that frees a group that is allocated when the sonification is started and in which all the synth started by the sonification are contained. This way, every sonification will have its separate group. Anyway, it is undesirable that the user has the responsibility to implement this kind of logic and Panson should be able to handle this hiding the logic from the user. It is worth noticing that this may not be completely possible without modifying sc3nb.

5.2.2 OpenFace

As we saw, OpenFace introduces some limitations on what we can achieve within the scope of our application. For instance, when doing real-time facial feature sonification, the camera device will be occupied by the OpenFace executable. This means that it will not be always possible to create an RTVideoPlayer object: depending on the operating systems and driver in use, access to the device could be or not be granted to multiple processes. If not, it will not be possible to record the video stream, which is an important part of part of the behaviour analysis support use case. The only way we can possibly face this issue is to research and document setup solutions for the different operating systems. In the worst case the user would have to record the video in advance and resort to offline processing.

So far, the application would work properly only on macOS and Linux, because it relies on Unix named pipes to obtain realtime features. We already mentioned in 4.2.4.1 the possibility of using the ZeroMQ messaging library to stream facial features and we added that there is an existent prototype working for Windows. Towards the end of the development, there was a contribution to the Github issue that we opened at the beginning of the project [10], stating that an integration of ZeroMQ into OpenFace has been implemented. Despite being a simple OpenFace adaptation, it should be capable of streaming AUs, gaze and head information on Windows, macOS and Linux. A Python script to retrieve features in real-time is also provided; adapting it to our project should not be very difficult. Adopting this solution would be a way to make the application cross-platform, but it would force users to install OpenFace without using docker.

When designing sonifications, we have to keep in mind that OpenFace detects different AUs with different precision [27]. A good idea would be to put emphasis on AUs that are detected more precisely. OpenFace also produces in its output accuracy information under the labels of “confidence” and “success”. These two labels provide information about how precise was

the tracking and whether it is considered successful respectively. These could be used to make sonifications more stable and capable of handling bad predictions.

Conclusion

In this work we described the design and implementation of Panson: a framework for interactive sonification. Despite still having several design and development challenges ahead, Panson is capable of providing support for many of the most common tasks that are common in sonification systems, with features for both online and offline sonification. It effectively demonstrates how to leverage multi-core systems in Python, bypassing the limitations imposed by the language and making it possible to tackle CPU-intensive tasks such as multi-stream processing and video data display. Through native ipywidgets support for its components, Panson is able to provide good graphical interaction in Jupyter Notebooks, which comes in handy especially in research contexts.

Next, we described the implementation of a facial feature sonification application based on Panson. Panson has proven itself valuable in the development of the application, providing adequate support all the desired features. The application is appropriate as example of Panson's potential, but, to be practically useful, it would require more effort invested into the design and practical evaluation of sonifications. From a technical perspective, the application is fully capable of real-time performance and no technical issues need further attention.

Panson is not yet mature enough to be practically used in production or research applications, but the current state of the project naturally suggests further developments that could improve it to that point. A successful refinement of Panson would provide a solid tool to support the development of interactive sonification applications.

Bibliography

[1] Action units. https://github.com/TadasBaltrusaitis/OpenFace/wiki/Action-Units. Accessed: 2022-04-22.

[2] Au_r vs au_c consistency? https://github.com/TadasBaltrusaitis/OpenFace/issues/228. Accessed: 2022-04-22.

[3] Framerate for output movie by facelandmarkvidmulti. https://github.com/TadasBaltrusaitis/OpenFace/issues/746. Accessed: 2022-04-25.

[4] integration openface in other projects. https://github.com/TadasBaltrusaitis/OpenFace/issues/409. Accessed: 2022-04-25.

[5] Is there a planned release date for the messaging server? https://github.com/TadasBaltrusaitis/OpenFace/issues/492. Accessed: 2022-04-25.

[6] Multithreaded python without the gil. https://github.com/colesbury/nogil. Accessed: 2022-04-11.

[7] Openface: Openface - a state-of-the art tool intended for facial landmark detection, head pose estimation, facial action unit recognition, and eye-gaze estimation. https://github.com/TadasBaltrusaitis/OpenFace. Accessed: 2022-04-22.

[8] A python api for supercollider. https://github.com/josiah-wolf-oberholtzer/supriya. Accessed: 2022-04-12.

[9] The python programming language. https://github.com/python/cpython. Accessed: 2022-04-11.

[10] Real-time usage with named pipe. https://github.com/TadasBaltrusaitis/OpenFace/issues/995. Accessed: 2022-04-25.

[11] Supercollider. https://supercollider.github.io. Accessed: 2022-04-11.

[12] Supercollider3 interface for python and jupyter notebooks. https://github.com/interactive-sonification/sc3nb. Accessed: 2022-04-11.

[13] Zeromq support for pubsub interaction with openface. https://github.com/TadasBaltrusaitis/OpenFace/issues/375. Accessed: 2022-04-25.

[14] Globalinterpreterlock - python wiki. https://wiki.python.org/moin/GlobalInterpreterLock, 12 2020. Accessed: 2022-04-11.

[15] Client vs server. https://doc.sccode.org/Guides/ClientVsServer.html, 2022. Accessed: 2022-04-12.

[16] Mdapiano. https://doc.sccode.org/Classes/MdaPiano.html, 2022. Accessed: 2022-04-29.

[17] Michele nalli / master thesis material. https://gitlab.ub.uni-bielefeld.de/mnalli/master-thesis-material, 2022.

[18] Multi-client setups. https://doc.sccode.org/Guides/MultiClient_Setups.html, 2022. Accessed: 2022-04-14.

[19] Non-realtime synthesis (nrt). https://doc.sccode.org/Guides/Non-Realtime-Synthesis.html, 2022. Accessed: 2022-04-15.

[20] Project jupyter. https://jupyter.org/, 2022. Accessed: 2022-04-11.

[21] Python image sequence: Load video and sequential images in many formats with a simple, consistent interface. https://github.com/soft-matter/pims, 2022. Accessed: 2022-04-17.

[22] Server guide. https://doc.sccode.org/Guides/Server-Guide.html, 2022. Accessed: 2022-04-14.

[23] Streamz. https://streamz.readthedocs.io/en/latest, 2022. Accessed: 2022-04-18.

[24] Abhinav Ajitsaria. What Is the Python Global Interpreter Lock (GIL)? - Real Python. https://realpython.com/python-gil, 12 2020. Accessed: 2022-04-11.

[25] Lorena Aldana, Steffen Grautoff, and Thomas Hermann. Cardiosounds: A portable system to sonify ecg rhythm disturbances in real-time. pages 20–27, 06 2018.

[26] Tadas Baltrusaitis, Amir Zadeh, Yao Chong Lim, and Louis-Philippe Morency. Openface 2.0: Facial behavior analysis toolkit. In *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*, pages 59–66, 2018.

[27] Tadas Baltrušaitis, Peter Robinson, and Louis-Philippe Morency. Openface: An open source facial behavior analysis toolkit. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–10, 2016.

[28] Daniel Cesarini, Thomas Hermann, and Bodo Ungerechts. Real-time sonification in swimming: from pressure changes of displaced water to sound. 09 2013.

[29] Mark Frank and Paul Ekman. Appearing truthful generalizes across different deception situations. *Journal of personality and social psychology*, 86:486–95, 04 2004.

[30] Mathias Funk, Kazuhiro Kuwabara, and Michael Lyons. Sonification of facial actions for musical expression. pages 127–131, 05 2005.

[31] Magdalena Gippert. Sonification in gymnastics: Can online auditory feedback improve the execution of a complex motor skill? Master's thesis, Bielefeld University, Bielefeld, Germany, 2018. Unpublished Master's thesis.

[32] Jan Hammerschmidt, René Tünnermann, and Thomas Hermann. Infodrops: sonification for enhanced awareness of resource consumption in the shower. In *ICAD 2013*, 2013.

[33] Thomas Hermann. Taxonomy and definitions for sonification and auditory display. 2008.

[34] Thomas Hermann, Andy Hunt, and John G. Neuhoff. *The Sonification Handbook*, chapter Sonification Techniques, pages 299–428. Logos Verlag, Berlin, 2011.

[35] Thomas Hermann, Andy Hunt, and John G. Neuhoff. *The Sonification Handbook*, chapter 15, pages 363–397. Logos Verlag, Berlin, 2011.

[36] Thomas Hermann, Alexander Neumann, and Sebastian Zehe. Head gesture sonification for supporting social interaction. In *Proceedings of the 7th Audio Mostly Conference: A Conference on Interaction with Sound*, pages 82–89, New York, NY, USA, 2012. Association for Computing Machinery.

[37] Thomas Hermann and Dennis Reinsch. *Sc3nb: A Python-SuperCollider Interface for Auditory Data Science*. Association for Computing Machinery, New York, NY, USA, 2021.

[38] Thomas Hermann, Bodo Ungerechts, Huub Toussaint, and Marius Grote. Sonification of Pressure Changes in Swimming for Analysis and Optimization. pages 60–67. The International Community for Auditory Display (ICAD), 2012.

[39] Antonino Ingargiola and contributors. What is the jupyter notebook? - jupyter/ipython notebook quick start guide 0.1 documentation. `https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html`, 2015. Accessed: 2022-04-11.

[40] IPython Development Team. Built-in magic commands. `https://ipython.readthedocs.io/en/stable/interactive/magics.html`, 2022. Accessed: 2022-04-17.

[41] Petr Janata and Edward Childs. Marketbuzz: Sonification of real-time financial data. 01 2004.

[42] P.N.Siva Jyothi and Rohita Yamaganti. A review on python for data science, machine learning and iot. *International Journal of Scientific & Engineering Research*, 10(12), 2019.

[43] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[44] G. Kramer, B. Walker, R. Bargar, and International Community for Auditory Display. *Sonification Report: Status of the Field and Research Agenda*. International Community for Auditory Display, 1999.

[45] Dave Kuhlman. *Section 1.1.* 2012.

[46] Ago Mesanovic. Development of a system for interactive sonification in physiotherapy. Master's thesis, Bielefeld University, Bielefeld, Germany, 2020. Unpublished Master's thesis.

[47] J.C. Hager P. Ekman, W.V. Friesen. *Facial Action Coding System (FACS): Manual*. A Human Face, Salt Lake City, USA, 2002.

[48] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[49] Project Jupyter. Output widgets: leveraging jupyter's display system. `https://ipywidgets.readthedocs.io/en/latest/examples/OutputWidget.html`, 2022. Accessed: 2022-04-16.

[50] Project Jupyter. Simple widget introduction. `https://ipywidgets.readthedocs.io/en/latest/examples/WidgetBasics.html`, 2022. Accessed: 2022-04-11.

[51] Project Jupyter. Widget list. `https://ipywidgets.readthedocs.io/en/latest/examples/WidgetList.html`, 2022. Accessed: 2022-04-14.

[52] Python Software Foundation. Initialization, finalization, and threads. `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`. Accessed: 2022-04-11.

[53] Python Software Foundation. Library and extension faq. `https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe`. Accessed: 2022-04-11.

[54] Python Software Foundation. Descriptor howto guide. `https://docs.python.org/3/howto/descriptor.html`, 2022. Accessed: 2022-04-11.

[55] Python Software Foundation. The hitchhiker's guide to python. `https://docs.python-guide.org/writing/style/`, 2022. Accessed: 2022-05-19.

[56] Python Software Foundation. multiprocessing - process-based parallelism. `https://docs.python.org/3/library/multiprocessing.html`, 4 2022. Accessed: 2022-04-11.

[57] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. Using the jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–2, 2017.

[58] scikit-image development team. Handling video files. `https://scikit-image.org/docs/dev/user_guide/video.html`, 2022. Accessed: 2022-04-17.

[59] Kyle Stratis. How to use generators and yield in python. `https://realpython.com/introduction-to-python-generators/`, 2019. Accessed: 2022-05-20.

[60] Alessandro Vinciarelli, Maja Pantic, and HervÃ© Bourlard. Social signal processing: Survey of an emerging domain. *Image and Vision Computing*, 27(12):1743–1759, 2009. Visual and multimodal analysis of human spontaneous behaviour:.

Acknowledgements

Many thanks to my supervisor, **Prof. Thomas Hermann**, and to my co-supervisor, **Dr. David Johnson**, for inspiring me with their enthusiasm for research and for their precious advice.

Many thanks to my father, **Raffaele Nalli**, and my mother, **Maria Giuseppe Scilimati**, as they are the reason why I had the opportunity to attend University.

Many thanks to **Prof. Renzo Davoli**, for making me discover the beauty of Computer Science.

Many thanks to **Massimiliano Muci**, for being one of my most loyal friends and being always willing to help others.

Many thanks to my dear friends **Elia Missud** and **Micaela D'Andrea**, for being close to me in some of my most difficult moments.