

# FACIAL BEHAVIOR SONIFICATION WITH THE INTERACTIVE SONIFICATION FRAMEWORK PANSON

*Michele Nalli, David Johnson<sup>1</sup>, Thomas Hermann<sup>2</sup>*

<sup>1</sup>Multimodal Behavior Processing Group, <sup>2</sup>Ambient Intelligence Group  
Faculty of Technology, Bielefeld University  
Bielefeld, Germany

[mnalli, djohnson, thermann]@techfak.uni-bielefeld.de

## ABSTRACT

Facial behavior occupies a central role in social interaction. Its auditory representation is useful for various applications such as for supporting the visually impaired, for actors to train emotional expression, and for supporting annotation of multi-modal behavioral corpora. In this paper we present a prototype system for interactive sonification of facial behavior that works both in real-time mode, using a webcam, and offline mode, analyzing a video file. The system is based on python and Jupyter notebooks, and relies on the python module `sc3nb` for sonification-related functionalities. Facial feature extraction is realized using OpenFace 2.0. Designing the system led to the development of a framework of reusable components to develop interactive sonification applications, called *Panson*, which can be used to easily design and adapt sonifications for different use cases. We present the main concepts behind the facial behavior sonification system and the *Panson* framework. Furthermore, we introduce and discuss novel sonifications developed using *Panson*, and demonstrate them with a set of sonified videos. The sonifications and *Panson* are Open Source reproducible research available on GitHub.

## 1. INTRODUCTION

Social signals, such as facial behavior, allow individuals to convey their feelings and attitudes of a social situation through non-verbal means [1] and play a critical role in communication [2]. They are generally expressed using behavioral cues such as facial expressions, eye gaze, gestures or speech prosody, typically lasting for a short period of time ranging from milliseconds to minutes [1]. Being able to understand the social signals of others during social situations is vital for the success and quality of an interaction [3]. Not everyone, however, is able to interpret or understand social signals equally. For example, most social signals rely on visual cues that are inaccessible by those with visual impairments putting them at a significant detriment in social interactions [2]. Typically those with visual impairments rely on other senses, or input modalities, to compensate for missed social cues during interactions, but this doesn't address missing signals such as eye gaze, facial expression or other subtle visual cues that cannot be compensated for with other senses [2]. Representing social signals through an-

other modality, such as audio via sonification approaches, may allow for such individuals to receive social signal information they would otherwise miss. To translate these signals to another modality, computational systems are needed to process and make sense of visual input.

Research in social signal processing aims to enable computers to sense, and possibly understand, human social signals through modalities such as audio and video [1]. When it comes to facial and eye behavior there are two main approaches for measurement of signals, message-based and sign-based [4]. A message-based approach aims to infer some understanding of the behavior, such as an emotional state, contained in the input signal. A sign-based approach, on the other hand, describes the shown behavior with no interpretation of its meaning. For example, a message-based system may infer that a video of a person smiling is *happy*, while a sign-based approach would code the video as having an upward movement of the person's lip corner [4]. Message-based methods require an interpretation of the meaning of a signal which depends on a number of factors such as gender, culture, or even mood. In contrast, sign-based approaches tend to be more objective leaving the interpretation to another party [1]. For the sonification of facial behaviour, using a sign-based approach, such as facial action coding (FAC), allows the receiver of the sonified information to make judgements regarding the meaning of the signal avoiding problems of human biases that may be present in models for message-based systems, such facial emotion recognition models [5].

FAC is a widely used method for describing facial expressions [4]. The FAC System (FACS) defines 44 facial action units (FAUs) 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 FAUs that can be interpreted to obtain information on basic emotions displayed, personality traits or social signals. The social signal sonification application described in this paper will consider and sonify FAUs as well as other head and eye behavior in a sign-based approach, leaving their eventual interpretation to listeners. We use OpenFace 2.0 [6] to extract these signals which are then sonified with a parameter-mapping sonification approach using the novel interactive sonification tool *Panson*.

In this paper, we first introduce the design of our prototype system for interactive sonification of facial behavior that works both in real-time mode, using a webcam, and offline mode by analyzing video files. The system is based on Python and Jupyter notebooks, and relies on `sc3nb` [7], a Python module for coding sonifications with `SuperCollider3`. Facial feature extraction is realized using OpenFace 2.0. In turn we focus on the framework



This work is licensed under Creative Commons Attribution – Non Commercial 4.0 International License. The full terms of the License are available at <http://creativecommons.org/licenses/by-nc/4.0/>

of reusable components, called *Panson*, that emerged while developing the application, which can be used for general interactive sonification applications, independent from the current use case. We focus here on the main concepts of *Panson*, and its utility is presented with interactive- and non-interactive video sonification examples.

## 2. RELATED WORK

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. A comprehensive review is impossible, instead the featured papers serve to bring up selected ideas and techniques to relate sensed data to auditory representations to enrich interaction.

We start with the sonification of posture: El Shimy et al. have presented an interactive system to use the relative orientation of a body with respect to fellow musicians to affect the individual mix the others obtain, allowing to tune into others' sound parts by moving and orienting the body [8]. While this is not a sonification setting as such, it illustrates that the technology-driven utilization of bodily gestures can serve and support interaction.

On the level of head gestures, Hermann et al. have introduced a head-gesture sonification system with the idea to enable visually impaired users to better integrate their interlocutors spontaneous non-verbal / communicative backchanneling, for instance via nodding, tilting the head, shaking the head, etc [9]. Such gestures are often accompanied by verbal backchanneling, the only part available to visually impaired users. They integrated an inertial sensor into a hat to deliver the data for various types of parameter mapping sonification. Event-based sonification, e.g. sonifying inflection points of the raw sensor data, proved useful to obtain a sparse signal able to be put into the background at a low distraction from verbal interaction. An important idea was to enable visually impaired users to experience the effect of their own head gestural activity – which could lead to building up one's own competences to backchannel non-verbally as sighted users do.

In the context of Augmented Reality, the indexical nature of head orientation as an expression of the focus of attention has been used by Neumann et al. to mutually inform interlocutors in dyadic interaction to changes of attention and to enhance joint attention [10]. A number of sonification techniques, including event-based sonification using auditory icons were used for that purpose.

With respect to facial features, the SoFA (Sonifier of Facial Actions) system is a realtime facial movement sonification for musical expression [11]. Apart from musical tasks, it could be applied to a variety of tasks, such as augmenting visual processing of facial gestures through sound. Initially, the optical flow for the image (a grid of motion vectors) is computed; secondly, face detection associates motion vectors with face regions. The mapping is based on 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.

Facial expressions are often related to an emotional expression – and sonification systems to convey affective states have been developed, such as the EmoSonics system [12], which, however, provides a systematic means to 'breed' (i.e. evolutionary optimize) sounds that are perceived as having a distinct affective quality. Such schemes could be helpful for sonification systems to convey messages (as opposed to sign-based approaches using

raw/analogic data).

## 3. FACIAL FEATURE SONIFICATION

This section discusses the main concepts behind our facial behavior sonification system, focusing on requirements and design.

### 3.1. Use Cases

Sonification can be used for many purposes. In designing our facial behavior sonification system and its sonifications, we considered three very different use cases that the system should support.

The first and main use case for the system is to provide support for **visually impaired people during verbal interactions**. Visually impaired users could learn to interpret facial expression sonifications, so that during a conversation they could perceive their interlocutor's facial expression through sound.

Sonifications should be informative and convey through sound as much of the communicatively relevant information as possible while being easy to perceive and interpret correctly. Ideally, the users should be able to quickly learn the generated sounds, so that their primary focus can shift (or stay) on the conversation. Sonifications designed for this use case should be easy to learn, easy to interpret correctly and unobtrusive.

The second use case is to support facial expression analysis for various groups of users. **Facial Expression Analysis** is the process of inferring from facial expressions higher order information, such as basic emotions, pain, personality traits and so on. Using FAC (see section 1), it is possible to code facial actions more accurately than using subjective judgments based on intuition; for instance, it is possible to distinguish truth from lies more effectively than using arbitrary judgements [13].

Sonification has proven to be a powerful tool for exploring unknown features in data to obtain insights. The system should provide support for facial expression analysis; practically, this means providing ways to record the video and data streams, play them back augmented by different sonifications while allowing users to navigate the videos. It should also provide ways to perform offline processing of video files that were not recorded by the system. The system should be capable of recording data (and the facial expression video) and playing them back using different sonification methods; it should also be able to record the generated sounds. Users interested in the analysis might include medical experts, actors, or analysts of multimodal interaction corpora.

Last but not least, the system could be used as a Facial Expression Practice Tool, i.e. to support the **practice of facial expressiveness**. For example, it could be used by actors; in an hybrid online/offline mode of interaction, they could compare the sounds generated by their own facial expressions recorded through a webcam to the sounds generated from the offline processing of a recording and practice matching the two.

### 3.2. Facial Feature Extraction using OpenFace 2.0

OpenFace 2.0 is an open-source and cross-platform computer vision tool capable of state-of-the-art results in tasks such as facial landmark detection, facial action unit recognition, head pose estimation and gaze estimation, with one or multiple faces. It is able to analyze both video files (or sequences of image files) and realtime video streams from devices such as a webcam. OpenFace can run without the need for any sort of specialized hardware, such as

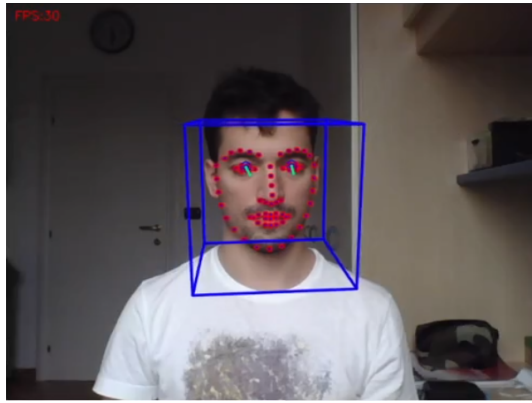


Figure 1: OpenFace display during real-time execution.

GPUs and it is freely available for research purposes in all its functionalities. Its usage and command line interface is documented in the Wiki of its GitHub repository [14].

OpenFace is capable of estimating the presence and intensity of 18 Facial Action Units, of which an illustration can be found at [15]. Presence is encoded as binary value, i.e. 0 or 1 for absent or present. Intensity is encoded numerically in the continuous range 0–5 where 0 indicates absence, 1 presence at minimum intensity and 5 presence at maximum intensity. No AU related to the head or gaze are taken into account, instead OpenFace estimates the head position and rotation, and eye gaze direction. It outputs a number of files, including a CSV file containing all feature values.

### 3.3. Facial Behavior Sonification System

We implemented our facial behavior sonification system as a Jupyter notebook to encourage interactive experimentation with sonification mappings; for demonstrations and sonification examples cf. Sec. 5. To make the process easier, the main focus of the notebook is on real-time sonification, but an example of offline usage is also provided. The notebook only contains code related to facial feature sonification: general-purpose code was encapsulated into the reusable components that make up *Panson*, and we will explore them in Section 4.

Many *Panson* components are able to be rendered in the notebook as widgets. Some of them, such as the “data players” (s. Fig. 2) allow the user to interact with the notebook – for instance by starting/pausing playback, en-/disabling recording; others, such as the “feature display” (s. Fig. 3) provide visual feedback, e.g. by displaying the data currently being sonified. Other components also allow to change parameters of a sonification mapping even while the playback is running.



Figure 2: DataPlayer widget for offline usage.

Data players (s. Sec. 4.4) allow users to control the playback of data in offline and online use cases and coordinate other components. For example, in the offline use case, a *VideoPlayer* object can optionally be used to play synchronously the video of the face

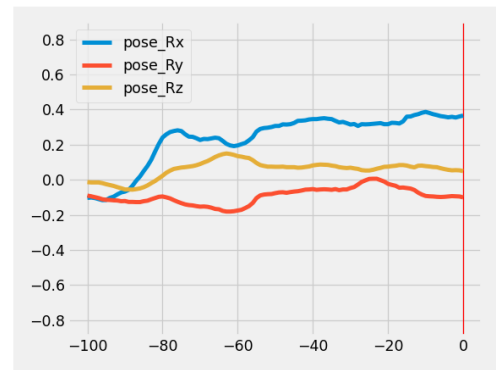


Figure 3: RTFeatureDisplay displaying head rotation data.

along with the sonification of features previously extracted from it; in the online use case, we can use an *RTVideoPlayer* object to record the video stream.

Unfortunately, OpenFace does not support any mechanism for streaming its predictions in real-time to another process, which is what we would need when performing real-time sonification. We overcome this problem by using Unix named-pipes, however, at the sacrifice to currently limit real-time application to Unix-like operation systems such as Linux or macOS.

## 4. THE PANSON SONIFICATION FRAMEWORK

*Panson* is a set of reusable components developed as part of the interactive sonification of facial behavior system presented in this paper. In this section, we provide an overview of the framework, its architecture, and its main components; an example can be seen in Figure 4. We will explain how the framework can be used to create new sonifications and how it can be used as a foundation for developing other systems. The aim of *Panson* is to provide a flexible and modular solution for interactive sonification, making it easier for researchers and developers to create and experiment with (interactive) sonifications.

A general guideline behind the development was the idea that *Panson* users should be able to implement applications without having to resort to advanced features of Python, such as generators, decorators and meta classes, unless these features would be really necessary. Moreover, *Panson* should encourage designers to write code in a concise and maintainable manner: in other words, it should encourage *pythonic code*.

### 4.1. Sonification

For the frequently applied approach of Parameter-Mapping Sonification, the *Sonification* class can be used to **define the sonification mapping**. It is an abstract class that provides a template for specifying the behaviour of the sonification at different stages of its execution (start, data processing, stop) independently from its context of execution (offline/online). The *Sonification* class integrates with the *sc3nb* library for sound synthesis, which makes it easy to define mappings. The `process` method is the central function where the mapping has to be defined; it expects a *Pandas Series* argument that describes the current data vector to be sonified. A set of *Sonification* objects can be grouped into a single object by using the *GroupSonification* class,

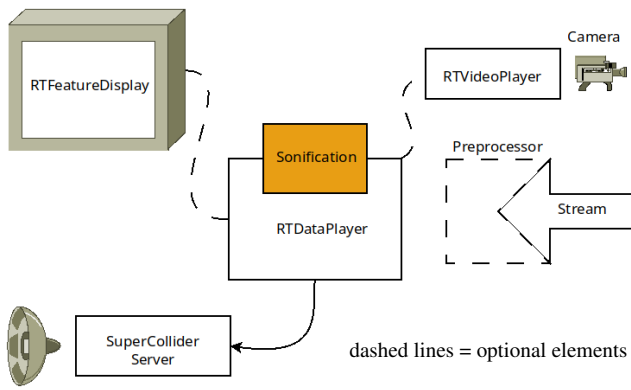


Figure 4: System diagram for the Real-time sonification.

allowing for the design of complex, aggregated sonifications in a modular way.

*Panson* provides support for interacting with sonifications through the “parameter” concept. *Parameters* are classes that declare attributes of the sonification class that influence how the mapping is computed – and they usually are expected to be modified by the user interactively, while the sonification is being executed. Most of the parameter classes have a notebook widget representation that allow changing parameter values through a GUI, when they are rendered in a Jupyter notebook. When a *Sonification* object is rendered in the notebook, its representation is the composition of all its widget parameters, as shown in Fig. 5: parameter values can be changed by interacting with these widgets. *Panson* includes a library of widget parameters that allow to use many ipywidgets, such as sliders, range sliders, dropdown widgets, checkboxes and more.

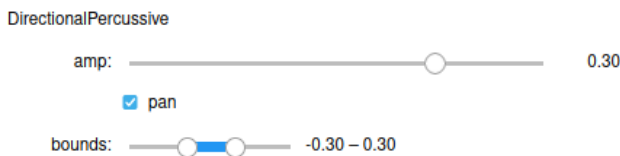


Figure 5: Example of Sonification object rendered in a Jupyter notebook.

### 4.2. Streams

Real-time data streams can be defined using the *Stream* class. To specify how to obtain real-time data, we have to implement a **generator function** that fetches and returns data. Generator functions are functions that return an iterator; Python provides a user-friendly construct to define generator functions through the *yield* statement. This statement controls the flow of the generator function, pausing the execution of the function and returning the value; when another value will be requested, the execution will be resumed from where it was paused.

In case users need to carry on operations when the stream is opened or closed, the *Stream* class provides a mechanism to specify hook functions to be executed, for instance, to switch on/off the data flow from the sensor system. If we specify a *Preprocessor* class (see Sec. 4.3), the preprocessing will be computed on each data vector before yielding the value out of the stream.

### 4.3. Preprocessors

The *Preprocessor* class is used to define preprocessing methods for the data, applicable for both offline and online usage. This class is an abstract class that requires the implementation of only one method named *preprocess*, where the logic relative to the preprocessing is implemented.

### 4.4. Data Players

Data players are components that 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 commands to *scsynth* with the correct timing.

#### 4.4.1. DataPlayer

*DataPlayer* is the class for *offline* processing: it supports loading data from CSV files or from pandas *DataFrame* objects. Data should contain timestamps, or alternatively, a static frame rate value should be specified. Internally, the data player loops over rows of the *DataFrame* with the correct timing and computes 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 time-reversed. All these operations can be carried on while the data player is running.

This class is also capable of **recording** the audio output generated throughout its playback. For that, users have to explicitly start and stop sound recording, which can either be done in code, or interactively using the widget view. Additionally, the *DataPlayer* class provides an *export* feature capable of rendering the full playback of the data into an audio file using SuperCollider’s **Non-Realtime Synthesis**. When creating a *DataPlayer* object, it is possible to specify a *VideoPlayer* object (s. Sec. 4.6.1). The *DataPlayer* will then use the *VideoPlayer* object to display the video synchronously with the data.

#### 4.4.2. RTDataPlayer

*RTDataPlayer* is designed for *online* (real-time) processing using a **single stream**. When started, the data player will open the stream and start 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 at the right time.

*RTDataPlayer*, similarly to *DataPlayer*, is capable of recording generated sounds into audio files. Additionally, it is also capable of logging data as CSV files; of course, thereby logged data can be later replayed using a *DataPlayer*.

When creating an *RTDataPlayer* object, it is possible to specify a *RTVideoPlayer* object (s. Sec. 4.6.2). When set, the video stream will be recorded when recording or logging data.

#### 4.4.3. RTDataPlayerMT and RTDataPlayerMP

*RTDataPlayerMT* and *RTDataPlayerMP* are both **multi-stream data players**: they are able to jointly process multiple independent data streams, using multi-threading and multi-processing approaches respectively. Think of applications where

sensor data is streamed asynchronously from two or more different sources, e.g. physiological data via a smartwatch, inertial data via a sensor and webcam-based facial features, all at different frame rates. To be flexible Panson supports the simultaneous processing of such multiple streams.

To compute sonifications on data from multiple streams with potentially different frame rates, the classes compute the sonification at regular intervals, but take into account only the most recent data sample from each of the streams for data fusion. Data are then concatenated into a pandas Series. The frequency at which the sonification is computed can be specified as a constructor argument; if not specified, the highest stream frame rate value is automatically chosen.

RTDataPlayerMT starts a thread for each stream and sonifies the latest incoming data available from each stream at a regular frequency. This solution is subject to Python’s multi-threading limitations imposed by the the Global Interpreter Lock (GIL), that forbid parallel execution (i.e. on multiple processor cores) of threads, limiting its possibilities of effective application. RTDataPlayerMP works in a similar way, but using processes instead of threads, avoiding these limitations. The processes continuously fetch data generated by streams into memory that is shared between different processes (Figure 6). In Python, this is possible using the Array class from the multiprocessing module, which provides synchronized access to a shared memory map, allowing to move data efficiently between processes. Before the sonification is computed, local copies of all the shared-memory arrays are made and joined into a Series object.

Multi-stream data players support a 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 pre-processing in the streams will distribute the CPU load.

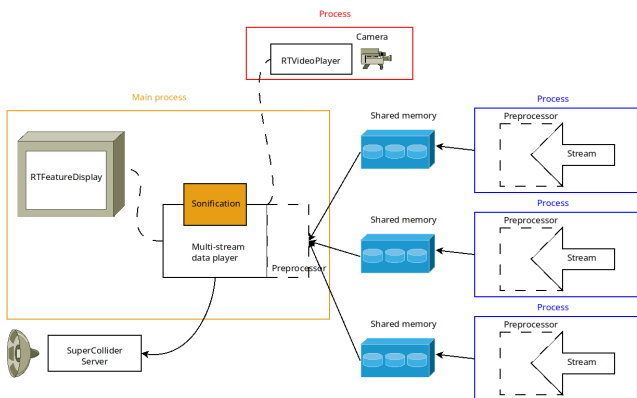


Figure 6: Real-time multi-stream sonification based on multi-processing. Dashed lines indicate optional elements.

#### 4.5. Feature Displays

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

RTFeatureDisplay. RTFeatureDisplay displays the history of most recent values sonified using a FIFO (first-in first-out) policy. At creation it requires specification of 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 provide a 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.

When sonifying *offline data* using a DataPlayer object for replay, the FIFO behaviour of the RTFeatureDisplay is not always desirable. In this case, it would be preferred to have a feature display that could highlight the current data point being sonified in a persistent sensor data series plot, rather than the history of most recently sonified data points. We plan to implement this additional functionality as FeatureDisplay class (i.e. name as above w/o RT).

#### 4.6. 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 for carrying out the actual operations; this is done to bypass Python’s GIL.

Video players display a separate window for the video playback even when being used from a Jupyter notebooks, because Jupyter notebooks only support displaying static image data or simple video file playback, not supporting all the necessary functionalities of Panson.

##### 4.6.1. 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 function to specify either 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 performs seek operations on the VideoPlayer that will in turn update the displayed video frame without visual delay.

##### 4.6.2. RTVideoPlayer

RTVideoPlayer provides support for **displaying and recording a video stream** coming from a device, e.g. a webcam. When the object is created, a process is started to grab frames from the device and display them into a window. If recording is started, captured video frames will be written/appended to a video file, and additionally, their precise timestamps will be written/appended into 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.

With these classes, most relevant interaction scenarios in the context of interactive sonification can be modelled, enabling much re-use of developed components. Any extension of Panson would benefit all existing sonification applications that rely on it. We

continue the discussion below, after presenting interaction examples and sonification designs using facial behavior data.

## 5. SONIFICATION DESIGN AND DEMONSTRATIONS

The Jupyter notebook that is included with *Panson* contains a number of examples of sonifications of facial features, a selection of which we are going to present and discuss briefly in this section. Videos are provided as supplementary material <sup>1</sup>.

### 5.1. Blinking Sonification

A sonification of blinking was realised by triggering a synthesized drop sound in an event-based fashion every time a blinking event was detected, considering AU 45. When the intensity value of this AU is greater than 1, blinking is detected as present; for this reason, the drop sound is triggered when this intensity threshold is crossed.

Oscillations in the continuously predicted intensity values can cause the drop sound to be re-triggered multiple times, making the sonification less effective. To mitigate the effects of noise created by immediate changes in AU intensity, we introduce a second threshold, using the higher for activation and the lower for deactivation of blinking, which are respectively set by default to 1.6 and 1 (within the value range of 0–5 for FAUs). In turn, as blinking will occur only when the state changes from non-blinking to blinking, oscillations that stay inside of the interval will not trigger any sound. These thresholds are *Parameters* of the sonification, and so they can be changed using widgets while the sonification is running. In addition to the activation of the feature, a simple extension would be to also sonify deactivation, for example using drop sound with a different pitch. This additional information allows speed of blinking, and also to recognize prolonged periods where the eyes are kept closed.

Videos v1 and v2 <sup>2</sup> demonstrate these event-based sonifications. Note that for the sake of demonstration in the videos the sound is pretty dominant – yet imagine it to be leveled to be at the threshold of perception so that it doesn't distract (but rather enriches) verbal communication for visually impaired interlocutors.

### 5.2. Head Movements Sonification

Next, we implemented and show a sonification of head movements, taking into account head rotations rather than head position, as rotations provide more information during social interactions.

The simplest approach is to map rotation angles around different axes to different parameters of a continuously synthesized sound stream. Specifically, we chose to map rotations around pitch (i.e. nodding movement) are mapped to auditory pitch (aka MIDI scale), rotations around yaw (i.e. head shake movement) to stereo panning, and rotations around roll (i.e. unsure movement) are mapped as absolute values to the number of harmonics, i.e. brightness.

We clip angles beyond the  $[-45^\circ, +45^\circ]$  range, because angles outside that range are not very relevant in a normal conversational context. In practical situations, head rotations around the roll axis are rarely observed. Video example v3 demonstrates the

<sup>1</sup>Videos are provided at <https://doi.org/10.4119/unibi/2979098>

<sup>2</sup>see URL provided for demos

three orientations, at full level without any amplitude mapping so that the relevant features can be easily discerned.

However, without amplitude modulation the generated sound will quickly be distracting and annoying. The natural choice would be to apply a derivative-driven activation so that sound fades into silence once orientation changes stop. Here we propose and show another mapping in video example v4: the displacement from direct line of sight: If we assume for a conversation, that the head remains in normal (neutral) position, a simple improvement can be obtained by silencing the output to the degree that the head orientation is in the neutral position. In the implemented sonification, we consider "neutral" the static position of the head faced directly towards the camera.

Of course developers can easily, quickly and flexibly experiment with other designs, such as "excitatory" mappings. It is precisely such incremental test-and-experience loops that *Panson* is meant to facilitate and support.

### 5.3. Upper Face Musical Sonification

Some sonifications explore the idea of a musical mapping of facial behavior to sound by mapping different AU intensities to different tones of a scale. This approach is straightforward and easy to implement using a synthesizer for each AU whose amplitude is modulated by a mapping of the intensity of the AU.

The same mapping idea was adopted with an event-based approach, using a piano synthesizer. 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 amplitude set according to the new level.

Intensity values may oscillate quite a lot. When they oscillate continuously around the specified thresholds, they will continuously re-trigger the same event over and over, causing a noticeable amount of unnecessary noise. To alleviate the problem in the online use case we implement `avg_openface_stream` as a stream object, where a moving average performed on the data in preprocessing acts as lowpass filter and significantly reduces unwanted oscillations. Videos v5 and v6 are examples of these approaches.

### 5.4. Percussive Sonification

An approach based on percussive sounds can be useful for capturing the overall behaviour of a face. Here we describe two sonifications that employ an event-based approach using percussive sampled sounds. This approach is similar to the event-based approach described in the last section, where the intensity range is divided into five levels and sample playback is triggered when the intensity value crosses one of these levels. These sonifications take into account all AUs except for AU 45 (which is already considered by the blinking sonification) and AU 28 (Lip Suck, for which OpenFace does not provide an intensity prediction).

To make learning easier, AUs situated close to each other were assigned with similar samples; in particular, we associated high-pitched bell sound to the eyebrow area, a snare drum sound 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.

The first sonification, video v7, plays samples with a volume proportional to the intensity level of the respective AU. Intensity values are mapped to a decibel scale so that the variation is perceptually linear. This method is not very precise/salient as it is difficult for the human ear to compare levels. Nonetheless, it would hopefully be easy for a user to intuitively perceive the general direction of the intensity of the AUs (increasing or decreasing).

The second sonification, video v8, improves the previous one in a number of 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 the use of panning to play decreasing samples on the left speaker and increasing samples on the right speaker. To counter feature intensity oscillation, we adopted a similar approach to the one described in section 5.1; this time, activation and deactivation boundaries have to be specified for every threshold that would trigger a sample playback. A parameter allows to specify relative boundaries that are applied to all thresholds.

## 6. DISCUSSION

The implemented sonifications highlight many of *Panson's* features and provide a good point of departure for further development. However they leave much room for improvement to be convincing for the initially proposed use cases. This is due to the reason that framework development and application development have competed for project time. In the end we sacrificed design optimizations towards getting the system to a state that it can be offered to the community for own experiments and design, as *sonification for the masses*. Discussions relating to the individual sonifications have been laid out on the previous section. In this section we discuss the application in general, dimensions of evaluation and possible future developments.

### 6.1. Evaluation

We briefly describe the results of some technical evaluations of *Panson* in real-time setups. The evaluation does not aim to be exhaustive, but aims at allowing readers to judge the limits and applicability, and 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** (4 cores, maximum frequency 4.20 GHz).

#### 6.1.1. Latency

An important metric to consider is **processing latency**. In real-time scenarios, processing latency is the amount of time from when data are obtained from the stream to when instructions are sent to scsynth; this measure does not include sound server or audio interface latency, which is dependent on the specified server's settings. We performed the measurement using a dummy sonification that does not produce any sound.

The resulting processing latency is **0.49 ms**, independently from the size of the data returned by the stream. This value was

obtained by varying the elements of the arrays from 1 to 1 million elements. 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. The resulting value is small if compared with typical server latency values, where 50 ms is considered a small value.

#### 6.1.2. CPU Usage

In Figure 7, we can see a comparison between the multi-threading (MT) approach to multiple streams and the multi-processing (MP) one. To perform these evaluations, we used an "empty" sonification that does nothing at any of its stages, so that its impact on CPU consumption is negligible. To test this, we considered two streams generating zeroed arrays of 1000 elements at different frame rates, for a total of three threads/processes running: two of those associated to streams and the other one performing the sonification. For 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.

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.

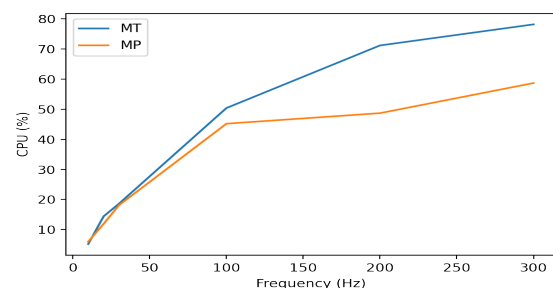


Figure 7: Comparison between CPU consumption of multi-stream data players.

With this setup the multi-threading data player seems to perform well for frequency values below 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 frame rates, multi-processing will be able to significantly reduce the main process' CPU usage, spreading the stream processes on other cores.

### 6.2. User Feedback

In a qualitative evaluation with stakeholders from the area of theatre, we learned that the first interaction phase is dominated by

exploring and learning about the sonification space. Participants indicated that percussive sonifications make it difficult to assess action unit intensity from the loudness of the sound, indicating a need to improve the mappings and to potentially focus more on continuous sounds. Participants did not find it useful to aid facial expression training, but came up with new ideas such as using the system for artistic expression to augment theatre performance.

## 7. CONCLUSION

In this paper, we described the design and implementation of an interactive facial behavior sonification system and of *Panson*, a general-purpose 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 in interactive sonification systems, with features for both on-line 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 multistream 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 an 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 recommended for production or research applications, but the current state of the project naturally suggests further developments that could improve it to attain that point. A successful refinement of *Panson* would provide a solid tool to support the development of interactive sonification applications. *Panson* is available via GitHub and we hope it may serve useful for the Auditory Display community.

## 8. REFERENCES

- [1] A. Vinciarelli, M. Pantic, and H. Bourlard, "Social signal processing: Survey of an emerging domain," *Image and Vision Computing*, vol. 27, no. 12, pp. 1743–1759, Nov. 2009.
- [2] S. Qiu, P. An, J. Hu, T. Han, and M. Rauterberg, "Understanding visually impaired people's experiences of social signal perception in face-to-face communication," *Universal Access in the Information Society*, vol. 19, pp. 1–18, Nov. 2020.
- [3] A. G. Halberstadt, S. A. Denham, and J. C. Dunsmore, "Affective Social Competence," *Social Development*, vol. 10, no. 1, pp. 79–119, Feb. 2001.
- [4] J. F. Cohn, Z. Ambadar, and P. Ekman, "Observer-based measurement of facial expression with the Facial Action Coding System," in *Handbook of Emotion Elicitation and Assessment*, ser. Series in Affective Science. New York, NY, US: Oxford University Press, 2007, pp. 203–221.
- [5] Y. Chen and J. Joo, "Understanding and mitigating annotation bias in facial expression recognition," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2021, pp. 14 980–14 991.
- [6] T. Baltrusaitis, A. Zadeh, Y. C. Lim, and L.-P. Morency, "OpenFace 2.0: Facial Behavior Analysis Toolkit," in *2018 13th IEEE International Conference on Automatic Face Gesture Recognition (FG 2018)*, May 2018, pp. 59–66.
- [7] "sc3nb: a Python-SuperCollider Interface for Auditory Data Science," in *AudioMostly 2021: Sonic experiences in the era of the Internet of Sounds*. ACM, New York, NY, USA, 2021, p. 8 pages. [Online]. Available: <https://pub.uni-bielefeld.de/record/2956377>
- [8] D. El-Shimy, T. Hermann, and J. Cooperstock, "A Reactive Environment for Dynamic Volume Control," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, G. E. Essl, B. Gillespie, M. Gurevich, and S. O'Modhrain, Eds. University of Michigan, 2012. [Online]. Available: <https://nbn-resolving.org/urn:nbn:de:0070-pub-25280616>, <https://pub.uni-bielefeld.de/record/2528061>
- [9] T. Hermann, A. Neumann, and S. Zehe, "Head gesture sonification for supporting social interaction," in *AM '12: Proceedings of the 7th Audio Mostly Conference: A Conference on Interaction with Sound*, A. Floros and I. Zannos, Eds. ACM, 2012, pp. 82–89. [Online]. Available: <https://nbn-resolving.org/urn:nbn:de:0070-pub-25327454>, <https://pub.uni-bielefeld.de/record/2532745>
- [10] A. Neumann, T. Hermann, and R. Tünnermann, "Interactive Sonification to Support Joint Attention in Augmented Reality-based Cooperation," in *Proceedings of ISON 2013, 4th Interactive Sonification Workshop*, 2013, pp. 58–64. [Online]. Available: <https://nbn-resolving.org/urn:nbn:de:0070-pub-26296398>, <https://pub.uni-bielefeld.de/record/2629639>
- [11] M. Funk, K. Kuwabara, and M. J. Lyons, "Sonification of facial actions for musical expression," *CoRR*, vol. abs/2010.03223, 2020. [Online]. Available: <https://arxiv.org/abs/2010.03223>
- [12] T. Hermann, J. Yang, and Y. Nagai, "EmoSonicS – Interactive Sound Interfaces for the Externalization of Emotions," 2016. [Online]. Available: <https://nbn-resolving.org/urn:nbn:de:0070-pub-29050360>, <https://pub.uni-bielefeld.de/record/2905036>
- [13] M. Frank and P. Ekman, "Appearing truthful generalizes across different deception situations," *Journal of personality and social psychology*, vol. 86, pp. 486–95, 04 2004.
- [14] "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.
- [15] T. Baltrusaitis, A. Zadeh, Y. C. Lim, and L.-P. Morency, "Openface 2.0: Facial behavior analysis toolkit," in *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*, 2018, pp. 59–66.