PiPo, A Plugin Interface for Afferent Data Stream Processing Modules

Norbert Schnell, Diemo Schwarz, Joseph Larralde, Riccardo Borghesi

PIPO, A PLUGIN INTERFACE FOR AFFERENT DATA STREAM PROCESSING MODULES

Norbert Schnell
UMR STMS
IRCAM-CNRS-UPMC
schnell@ircam.fr

Diemo Schwarz
UMR STMS
IRCAM-CNRS-UPMC
schwarz@ircam.fr

Joseph Larralde
UMR STMS
IRCAM-CNRS-UPMC
larralde@ircam.fr

Riccardo Borghesi
UMR STMS
IRCAM-CNRS-UPMC
borghesi@ircam.fr

ABSTRACT

We present *PiPo*, a plugin API for data stream processing with applications in interactive audio processing and music information retrieval as well as potentially other domains of signal processing. The development of the API has been motivated by our recurrent need to use a set of signal processing modules that extract low-level descriptors from audio and motion data streams in the context of different authoring environments and end-user applications.

The API is designed to facilitate both, the development of modules and the integration of modules or module graphs into applications. It formalizes the processing of streams of multidimensional data frames which may represent regularly sampled signals as well as time-tagged events or numeric annotations. As we found it sufficient for the processing of incoming (i.e. *afferent*) data streams, *PiPo* modules have a single input and output and can be connected to sequential and parallel processing paths. After laying out the context and motivations, we present the concept and implementation of the *PiPo* API with a set of modules that allow for extracting low-level descriptors from audio streams. In addition, we describe the integration of the API into host environments such as Max, Juce, and OpenFrameworks.

1. INTRODUCTION

1.1 Context and Motivation

Many of the interactive audio applications that we have developed over the past years in collaboration with artists and other researchers rely on signal processing techniques to automatically analyse and annotate audio and motion sensor streams. We often refer to the techniques we deploy in this context as *content-based audio processing* [1]. These techniques generally allows for interactively transforming recorded audio materials as a function of low-level audio descriptions such as pitch, intensity, and timbre descriptions as well as segmentations into temporal units such as notes, syllables, and musical phrases. Similar processing applies in this context to motion capture streams to extract movement qualities and meaningful events and temporal units such as onsets and gestures. In this processing, we frequently reuse a set of algorithms such as filters, projections, extractors, and detectors that may apply in real-time to incoming data streams or, offline, to data streams recorded into files. Since these data streams occur in the overall interactive systems we develop as inputs, we refer to them as *afferent* data streams.

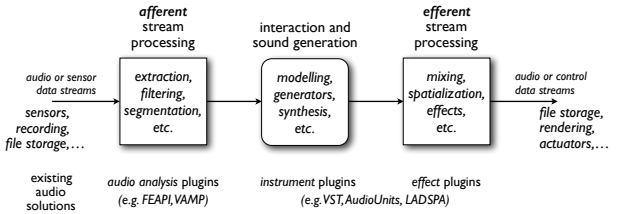

Figure 1. An interactive audio system with *afferent* and *efferent* data streams. The labels at the bottom cite existing plugin interfaces for audio applications.

Figure 1 shows the overall structure of such an interactive system. The schema does not distinguish whether the audio and motion data streams actually enter the system in real-time or whether they are read from files. Afferent streams processed in real-time typically originate from a microphone or motion sensors. In some of the systems we have developed, the streams are used to control an interactive system through sound (e.g. voice) or movement. In many of these systems, the same — or very similar — processing that is applied in real-time, applies to data streams read from files. For example for concatenative synthesis [21], audio descriptors are extracted from prerecorded materials. While some interactive systems may generate sound in real-time, the generated description and annotations may be used to create visualizations (e.g. in the context of musicology or education) as well as to support the editing and transformation of recordings in post-production systems.

In general, the processing of afferent streams can be described as *reducing* the data streams in terms of their complexity, dimensionality, and data rate. Typical terms used to characterize this processing include *filtering*, *anal-*

ysis, *extraction*, *description*, *detection*, *recognition*, *scaling*, and *mapping*.

The *PiPo* API (*Plug-in Interface for Processing Objects*) formalizes modules that in this sense transform an incoming data stream into an output data stream allowing for a possibly wide range of streams as well as for modules of arbitrary complexity going from simple scalings to sophisticated machine learning algorithms.

The major motivations for developing the *PiPo* API can be summarized as follows:

- facilitating the integration of algorithms of different origins (i.e. developers) into a given application
- facilitating the use/comparison of different algorithms of similar functionalities in a given context (e.g. applying different filters, extractors or classifiers to the same input stream)
- facilitating the integration of a given algorithm into different contexts and applications
- facilitating the development of applications where the same algorithm applies to data streams in real-time and offline

Ultimately, the motivation for developing the API is the idea of enabling the development of an ecosystem of stream modules and host environments in particular domains as well as across different domains of signal processing.

1.2 Requirements

In this section, we give an overview over the most important general requirements for an afferent data stream processing framework for real-time and off-line use. These requirements concern specific functionalities as well as their efficient implementation in a real-time system (see [22]).

1.2.1 Functional Requirements

Scheduling Processing should run either in batch on sound files and buffers, or on a live audio stream

Segmentation Allow several streams of segmentations in parallel and overlapping segments, or an implicit segmentation, where segments are analysis frames, elementary waveforms, or whole sound files.

Temporal Modeling Any number of temporal modeling algorithms can be integrated, either universal (modeling all descriptors, e.g. mean) or specific (modeling specific descriptors only, e.g. geometric mean for pitch).

Data Type Data can be numeric scalars, vectors, matrices, or strings

Multi-Modality The input data type and rate should allow motion and other data and not be limited to audio only.

User Composability Modules should be composable by the user in the host environment (without having to write and compile code), e.g. chaining feature extractors, smoothing filters, segmentation, and temporal modeling, in order to allow experimentation and rapid prototyping.

1.2.2 Implementation Requirements

Easy Integration and Efficiency It should be easy to integrate the framework in any platform and environment, including real-time and resource-constrained systems (e.g. single-board computers). This basically stipulates that the API be written in C or C++.

Dynamic Plugin Loading It should be possible to add processing modules as plugins to an existing host installation, e.g. by leveraging dynamic linking of shared libraries.

Efficient Modularisation The framework should allow an efficient implementation, notably by *sharing commonly used calculation results*, most of all the FFT representation, between modules, by *avoiding copying* and re-sending data, instead writing them directly to its destination.

External Data External data streams and sources of segmentation, such as a human tapping on attacks oder existing analysis files, must be integratable into the data flow.

Reanalysis A subset of descriptors or only the segmentation and subsequent temporal modeling can be re-run with changed parameters.

Almost all of these requirements are fulfilled by *PiPo*, with the exception of the possibility to pass strings as data elements. This has been avoided to simplify the API and avoid problems of memory-handling. A fixed set of strings (such as class labels for machine-learning) can always be transmitted by their index.

The top-level requirements, that best distinguish *PiPo* from other frameworks are dynamic linking of plugins, multi-modality, and user-composability of modules.

1.3 Related Work

In the rich existing work, we must distinguish audio analysis libraries and toolboxes (see the recent overview [15]) from plugin APIs which impliy a formalization of the input/output formats and the dynamic loading of modules.

Several plugin APIs are commonly used in the world of audio signal processing and virtual instruments, namely LADSPA (Linux Audio Developer's Simple Plugin API), [1] VST (Virtual Studio Technology by Steinberg), [2] and AU (Audio Units by Apple). [3] These APIs are mainly designed for transforming an input audio stream (effect processing) or for generating an audio stream in reaction to incoming MIDI events (virtual instruments). Thus they are not applicable to the demands of general data processing or audio feature extraction.

Many monolithic or collections of analysis modules for popular real-time environments exist, such as

[1] http://www.ladpsa.org
[2] http://ygrabit.steinberg.de
[3] http://developer.apple.com/audio/audiounits.html

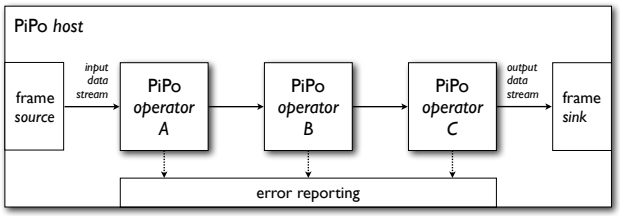

Figure 2. A chain of modules in a host environment.

`analyser~` [11], the patch-based ZSA [13] for MAX, imtr-analysis [22], *TimbreID* [3] for PureData. None of these can be integrated into other environments.

The first descriptor analysis frameworks that would allow the dynamic inclusion of external modules are either plugin frameworks such as the sadly defunct FEAPI [12], and the more lively VAMP [6]. [4] However, the latter does not propose user-composability nor multi-modality (the input is always audio).

There are many existing libraries for audio descriptor analysis (*Yaafe* [5] [14], *Essentia* [6] [2], *OpenSmile* [7] [7, 8], *libXtract*, [8] *IrcamDescriptor* [18]) see this comparison [15]. None of them allow for dynamic linking, easy integration of new algorithms, or user composability without having to code a new module.

The MARSYAS framework, dedicated to music information retrieval, is concerned with scheduling [5] as well as CLAM, [9] but neither is a common environment for real-time sound and music applications.

In summary, no existing API combines all three top-level requirements of dynamic linking of plugins, multi-modality, and user-composability of modules.

2. CONCEPTS AND FORMALIZATION

The *PiPo* API formalizes *modules* as objects that receive a data stream as a succession of *frames* at their input and send a stream as output. As shown in figure 2, modules can be connected to a chain by connecting the input of one module to the output of another. In the simplest case, the processing requires a single module. A *PiPo host*, constructs the modules and connects to the input of the first module of the chain as the source of the stream of frames to be processed. In addition, the host connects to the output of the last module of the chain as the terminating sink that receives the resulting stream.

The data streams received and produced by *PiPo* modules are described by a set of *stream attributes* that are defined before the modules actually receive and produce any frames. This way, the initialization of a module may depend on the attributes of the incoming stream and the module may determine the attributes of the stream it produces as a function of the attributes of the incoming stream.

[4] http://vamp-plugins.org
[5] http://yaafe.sourceforge.net
[6] http://essentia.upf.edu/
[7] http://opensmile.sourceforge.net/
[8] http://jamiebullock.github.io/LibXtract/documentation/
[9] http://clam-project.org

The propagation of the stream parameters and the actual processing of frames are separated into two phases that are both initiated by the host through its connection to the first module. In both phases, each module receives information from its predecessor in the chain and sends information to its successor. In the initialization phase, the host sends out the stream parameters of the input stream to the first module which sends its output stream parameters to the input of the next module, and so forth, until the last module sends the resulting stream parameters back to the host connected to its outlet. Similarly, once the modules are initialized, the host can start sending frames into the input of the first module and receives the resulting frames from the output of the last module. Only in the case of error, as for example when a module cannot accept a stream with a given set of attribute values at its input, a module would report directly to the host, which in turn can output the error message to the host environment.

2.1 Streams of Frames

Each frame of a data stream is composed of a *time-tag* and a two-dimensional matrix of numeric values. A data stream is described by the following set of attributes:

- *frame rate* of the stream
- whether the frames of the stream are *time-tagged*
- *dimensions* of the frames' two-dimensional matrix
- *labels* describing the columns of the data matrix
- whether the frames' data matrices have a *variable number of rows*

In case of streams of time-tagged frames of an irregular rate, the *frame rate* attribute should announce the worst case (the fastest) rate, so that succeeding modules — or the host — can take this parameter into account (e.g. for allocating memory).

This formalization of data streams allows for representing a large spectrum of different signals, event streams, and numeric annotations. For example:

- *mono or multi-channel audio signals* are represented as scalars or multi-dimensional vectors of a constant frame rate
- *real or complex frames of spectral data* are represented as single column vectors or matrices of two columns (i.e. labeled *'real'* and *'imag'*), usually of a constant frame rate
- *multi-dimensional motion capture data streams* are represented as multi-dimensional row-vectors (e.g. labeled *'x'*, *'y'*, *'z'*) that may be time-tagged or of a constant frame rate
- *onset markers* are represented as time-tagged frames without numeric data (i.e. an empty matrix)
- *segments* are represented as time-tagged frames with a row-vector of data including a *'duration'* column and, optionally, multiple columns of values describing the segment (e.g. *'pitch'*, *'intensity'*, *'category'*)

- *harmonics* are represented as two-dimensional matrices with variable number of rows, one row for each harmonic, with multiple columns (*'frequency'*, *'amplitude'*, *'phase'*) of a constant frame rate

From the host's point of view, once constructed, an arbitrary chain of modules is defined by the stream it produces as a function of the input stream provided by the host. Before starting the actual stream processing by sending frames into the chain, the host retrieves the attributes of the output stream that can be used, for example, to allocate memory or bandwidth and automatically determine display options as well as to configure and generate other interactions with connected sub-systems or users.

2.2 Chains of Modules

As mentioned above, *PiPo* modules have a single input and output and can be connected to chains. Hereby, a chain of modules — conceptually as well as by implementation — may appear as a single module communicating with its environment (i.e. a host or connected modules) through a single in- and output and an error channel.

Apart from the *stream attributes* of its incoming stream, each module is configured and controlled by a set of typed *module parameters* that are explicitly declared through the *PiPo* API. Possible parameter types are single values of 64-bit float, 32-bit integer, string, and declared enumerated types as well as fixed or variable sized arrays of values of these types and heterogeneous variable sized arrays. In addition to its type, a module parameter is declared with a name, a short description, and a flag whether changing a given module parameter requires the reinitialization of the module — and consequently of the following modules in a chain.

An important feature of the design of the API is that it allows for implementing modules of virtually any complexity and for composing chains of modules of any granularity. An extractor of MEL cepstrum coefficients (MFCCs), for example, may be implemented as a single monolithic module or composed of a chain of modules that include the successive calculation of STFT frames, MEL coefficients, and DFT coefficients.

2.3 Graphs Beyond Chains

The construction of certain algorithms from basic modules requires more complex graphs of modules. For example, the extraction of a set of basic audio descriptors shown in figure 3 requires to split and merge the processing of the implied data streams. While the first split allows for processing the same audio frames in time and frequency domain, the second applies the calculation of a loudness descriptor and a spectral centroid to the same frequency domain frames produced by the STFT. The final set of estimated descriptor values (i.e. pitch, periodicity, AC1, loudness, and spectral moments) is merged to a single vector at the output of the sub-graph.

As described in section 2.2, any chain (or sequence) of modules can be considered as a single module. In the formalization of graphs in the *PiPo* API, parallel modules can also be reduced to a single module. These two rules, illustrated by figure 4, provide a consistent basis to build a large variety of complex *PiPo* graphs.

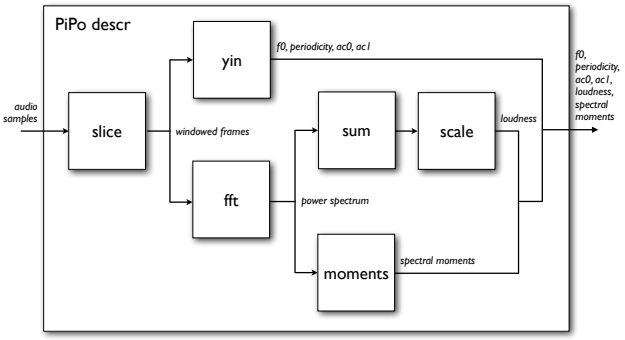

Figure 3. A complex graph of *PiPo* modules for calculating 9 basic audio descriptors.

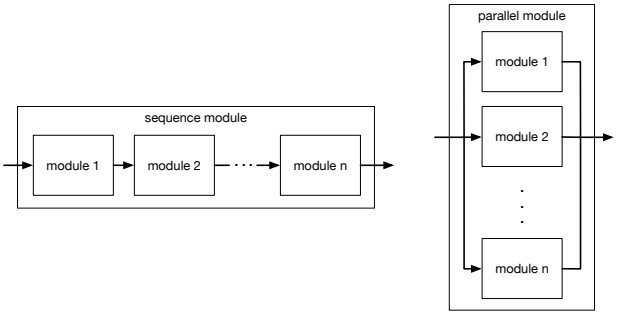

Figure 4. Any number of modules connected in sequence or in parallel can be reduced to a single module.

Figure 5 shows the structure of the `pipo.descr` module expressed in terms of sequence and parallel components.

2.4 Hosts

In summary, a *PiPo* host has to provide the following functionalities:

- constructing a single or a graph of modules
- parametrizing the modules

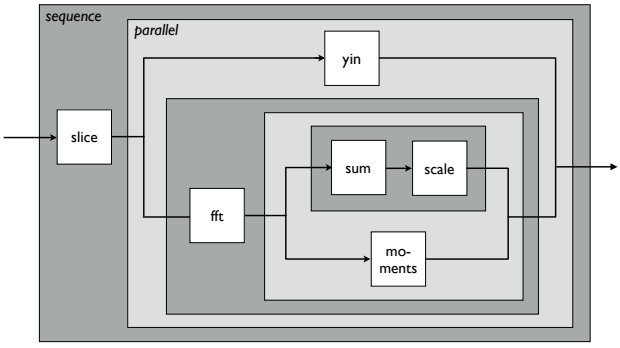

Figure 5. Decomposition of the `pipo.descr` module into sequence and parallel elements.

- connecting a terminating sink to the output of the chain
- acquiring the input stream
- initializing the modules by sending the input stream attributes into the chain
- handling initialization errors emitted by the modules
- sending the frames of the input stream into the chain and handling the frames of the output stream
- allowing for real-time parametrization of the modules (if applicable)

The *PiPo* API includes abstractions that support the implementation of hosts.

3. IMPLEMENTATION

PiPo is an API that essentially consists of a single C++ header file. This file defines the base *PiPo* class, and its declared parameters. [10] .

3.1 The PiPo API

The minimal module must inherit from the class `PiPo` and implement at least the `streamAttributes` and `frames` methods:

In `streamAttributes`, all initialisation can be done, as all input stream attributes are known. The output stream attributes are passed on to the receiving module via `propagateStreamAttributes`. In `frames`, only data processing and, when needed, buffering should be done. Output frames are passed on with `propagateFrames`.

If the module can produce additional output data after the end of the input data (e.g. filters), it must implement `finalize`, from within which more calls to `propagateFrames` can be made, followed by a mandatory call to `propagateFinalize`.

If the module keeps internal state or buffering, it should implement the `reset` method to put itself into a clean state.

A segmentation module calls the method `propagateSegment` to signal the onset, offset and exact boundaries of a new segment to following temporal modeling modules (which implement `segment`).

The utility function `signalError` can be used to pass an error message to the host.

3.2 Module Parameters

The template class `PiPo::Attr` permits to define scalar, enum, or variable or fixed size vector parameters of a pipo module that are exposed to the host environment.

They are initialised in the module constructor with a short name, a description, a flag if a change of value means the fundamental stream parameters must be reset (if true, `streamAttributes` will be called again for the whole chain), and a default value.

Their value can be queried in `streamAttributes` or `frames` (in real-time hosts, a parameter's value can change over time) with `PiPo::Attr::get()`.

[10] https://github.com/Ircam-RnD/pipo-sdk/tree/master/include

3.3 Example of a Minimal PiPo Module

```
class PiPoGain : public PiPo
{
private:
  std::vector<PiPoValue> buffer;

public:
  PiPoScalarAttr<double> factor;

  PiPoGain (Parent *parent, PiPo *receiver = NULL)
  : PiPo(parent, receiver),
    factor(this, "factor", "Gain Factor", false, 1.0) { }

  ~PiPoGain (void) { }

  int streamAttributes (bool hasTimeTags, double rate,
    double offset, unsigned int width, unsigned int height,
    const char **labels, bool hasVarSize,
    double domain, unsigned int maxFrames)
  { // can not work in place, create output buffer
    buffer.resize(width * height * maxFrames);

    return propagateStreamAttributes(hasTimeTags, rate,
             offset, width, height, labels,
             hasVarSize, domain, maxFrames);
  }

  int frames (double time, PiPoValue *values,
              unsigned int size, unsigned int num)
  { // get gain factor here, it could change while running
    double f = factor.get();
    PiPoValue *ptr = &buffer[0];

    for (unsigned int i = 0; i < num; i++)
    {
      for (unsigned int j = 0; j < size; j++)
        ptr[j] = values[j] * f;

      ptr    += size;
      values += size;
    }

    return propagateFrames(time, &buffer[0], size, num);
  }
};
```

3.4 Existing Modules

The list of existing *PiPo* modules can be organized into the following categories:

Stream Processing `slice` (windowing), `scale`, `sum`, `select` (get columns),

Filtering `biquad` (biquad filter), `mvavrg` (moving average filter), `median` (median filter), `delta` (derivative), `finitedif` [9], `bayesfilter` [10]

Segmentation `onseg` (segments starting at signal onset), `chop` (segments of regular intervals), `gate` (segments excluding weak sections), `sylseg` [16]

Temporal Modeling `mean`, `std`, `meanstd`, `min`, `max`

Analysis `descr` (basic audio descriptors), `yin` (pitch extractor), `moments` (centroid, spread, skewness, kurtosis), `lpc` (linear predictive coding), `lpcformants` (formant extraction), `psy` (pitch synchronous markers), `ircamdescriptor` [18]

Frequency Domain Processing `fft` (FFT from pre-windowed frames), `dct` (discrete cosine transform), `bands` (MEL bands and similar from power or amplitude spectrum), `mel` (MEL bands from audio stream), `mfcc` (MFCC from audio stream), `wavelet` (wavelet transform from audio stream)

They can be instantiated from C++ code using the precompiled *libpipo* library, thanks to the `PiPoCollection` class defined in the *PiPo* SDK, or be used in one of the environments described in 3.6. The `PiPoCollection` class acts as a module factory. It is able to instantiate *PiPo* graphs from a simple syntax, which can then be used as simple *PiPo*s in a host environment. It also allows users of the API to add their own *PiPo* modules to the original collection. Once added, more complex graphs combining modules from *libpipo* and these new modules can be instantiated and run inside a *PiPo* host.

3.5 PiPo Graph Construction

Graphs of *PiPo* modules can be either constructed in C++ code or — within a given host environment — through expressions of a very simple syntax. For the first case, the API defines a set of primitives that can be used to construct graphs of any complexity by arranging modules in sequence and in parallel. In the latter case, these primitives are instantiated by a parser function provided by the API.

3.5.1 Specific Graph Construction Modules

Additional to the data processing modules listed above, there are two internal modules that handle the connection of processing modules in sequence or in parallel.

The `sequence` module simply connects the upstream module to the downstream one (i.e. setting the latter as receiver, so that the API calls get propagated through the chain). The `parallel` module essentially consists of an ordered set of modules that receive the same input stream and output towards an internal `merge` module. Each incoming frame is processed by each of the parallel modules in the given order, whereby the `merge` module concatenates the output data column-wise into a single matrix that is output towards the receiver of the `parallel` module.

3.5.2 Graph Construction Syntax

The construction of sequences and parallel modules is also available at the user level via a simple syntax inspired by FAUST [17], with the following operators:

`:` (sequence)

`<` (branch)

`,` (parallel)

`>` (merge)

For example, the `pipo.descr` module described in section 2.3 would be written like this: `slice<yin,fft<sum:scale,moments>>`

A fifth operator, `_` (identity), allows the propagation of intermediate analysis results to the end of the graph. Following the sequence and parallel reduction rules from section 2.3, any *PiPo* graph is equivalent to a *PiPo*, and as a consequence must have a single input and a single output, which implies that the graph syntax must contain the exact same number of branch and merge operators.

3.6 Bindings

3.6.1 Max

The *PiPo* modules are available within the MAX visual programming environment via the MuBu package, [19] where they can run in real-time using the `pipo~` and `pipo` MAX objects, or offline using the `mubu.process` object.

3.6.2 Juce, OpenFrameworks, OpenMusic, Unity3D

PiPo has been integrated into the JUCE [11] framework, the creative coding framework OPENFRAMEWORKS, [12] the computer-aided composition environment OPENMUSIC [4] and the game development environment UNITY3D. [13] Most if these developments are based on the IAE (Interactive Audio Engine) library. [20] The library allows for loading a sound file as input of a user-specified *PiPo* chain and to retrieve the result at the output.

4. CONCLUSIONS AND FUTURE WORK

The *PiPo* API and modules are in production use in our department, and with research project partners, and artists in interactive gesture and music installations and digital instruments. We feel it could help a wider community for easy prototyping and transfer to developed products.

The *PiPo* API is currently in the process of being integrated in the RAPIDMIX API, a wider C++ software ecosystem including machine learning, signal feature extraction and audio processing libraries, as a standardized way of building modular signal descriptors and machine learning algorithms, integrating them into a global workflow and allowing users of this ecosystem to build sustainable code on top of a base collection of algorithms, by providing a flexible mean of interaction between its software components.

We made first steps to add an API similar to *PiPo* to also integrate the iterative training of machine learning and data processing easily into the same host environments.

The *PiPo* SDK that supports the development of modules as well as hosts, has been published under the BSD 3-Clause license at https://github.com/Ircam-RnD/pipo-sdk.

5. ACKNOWLEDGMENTS

The development of the *PiPo* API has received support from the RAPID-MIX project (H2020-ICT-2014-1 Project ID 644862), funded by the European Union's Horizon 2020 research and innovation programme.

[11] https://www.juce.com

[12] http://openframeworks.cc

[13] http://unity3d.com

6. REFERENCES

[1] X. Amatriain, J. Bonada, A. Loscos, J. Arcos, and V. Verfaille. Content-based transformations. *Journal of New Music Research*, 32(1):95–114, 2003.

[2] Dmitry Bogdanov, Nicolas Wack, Emilia Gómez, Sankalp Gulati, Perfecto Herrera, Oscar Mayor, Gerard Roma, Justin Salamon, José Zapata, and Xavier Serra. Essentia: An open-source library for sound and music analysis. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 855–858, New York, NY, USA, 2013. ACM.

[3] W Brent. A Timbre Analysis and Classification Toolkit for Pure Data. In *International Computer Music Conference*, New York City, NY, 2010.

[4] Jean Bresson, Carlos Agon, and Gérard Assayag. OpenMusic – Visual Programming Environment for Music Composition, Analysis and Research. In *ACM MultiMedia (MM'11)*, Scottsdale, United States, 2011.

[5] Neil Burroughs, Adam Parkin, and George Tzanetakis. Flexible scheduling for dataflow audio processing. In *Proceedings of the International Computer Music Conference (ICMC)*, New Orleans, Louisiana, USA, August 2006.

[6] Chris Cannam. The VAMP Audio Analysis Plugin API: A Programmers Guide. http://vamp-plugins.org/guide.pdf, 2008.

[7] Florian Eyben, Felix Weninger, Florian Gross, and Björn Schuller. Recent developments in opensmile, the munich open-source multimedia feature extractor. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 835–838, New York, NY, USA, 2013. ACM.

[8] Florian Eyben, Martin Wöllmer, and Björn Schuller. Opensmile: The munich versatile and fast open-source audio feature extractor. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, pages 1459–1462, New York, NY, USA, 2010. ACM.

[9] B. Fornberg. Finite difference method. *Scholarpedia*, 6(10):9685, 2011. revision #91262.

[10] Jules Françoise. *Motion-Sound Mapping by Demonstration*. PhD thesis, Université Pierre et Marie Curie, 2015.

[11] Tristan Jehan. Musical signal parameter estimation. Master's thesis, IFSIC, Université de Rennes, France, and Center for New Music and Audio Technologies (CNMAT), University of California, Berkeley, USA, 1997.

[12] Alexander Lerch, Gunnar Eisenberg, and Koen Tanghe. FEAPI: A Low Level Feature Extraction Plugin API. In *8th International Conference on Digital Audio Effects (DAFx05*, 2005.

[13] M. Malt and E. Jourdan. Zsa. Descriptors: a library for real-time descriptors analysis. In *5th Sound and Music Computing Conference*, pages 134–137, Berlin, Germany, August 2008.

[14] Benoit Mathieu, Slim Essid, Thomas Fillon, Jacques Prado, and Gaël Richard. YAAFE, an easy to use and efficient audio feature extraction software. In *Proceedings of the International Symposium on Music Information Retrieval (ISMIR)*, 2010.

[15] David Moffat, David Ronan, Joshua D Reiss, et al. An evaluation of audio feature extraction toolboxes. In *Proceedings of the COST-G6 Conference on Digital Audio Effects (DAFx)*, Trondheim, Norway, 2015.

[16] Nicolas Obin, François Lamare, and Axel Roebel. Syll-o-matic: an adaptive time-frequency representation for the automatic segmentation of speech into syllables. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[17] Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: an efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, 290, 2009.

[18] Geoffroy Peeters. A large set of audio features for sound description (similarity and classification) in the Cuidado project. Technical Report version 1.0, Ircam – Centre Pompidou, Paris, France, April 2004.

[19] Norbert Schnell, Axel Röbel, Diemo Schwarz, Geoffroy Peeters, and Ricardo Borghesi. MuBu & friends – assembling tools for content based real-time interactive audio processing in Max/MSP. In *Proceedings of the International Computer Music Conference (ICMC)*, Montreal, Canada, August 2009.

[20] Norbert Schnell, Diemo Schwarz, Roland Cahen, and Victor Zappi. IAE & IAEOU. In Roland Cahen, editor, *Topophonie research project : Audiographic cluster navigation (2009-2012)*, Les Carnets d'Experimentation de l'Ecole Nationale Superieure de Creation Industrielle, pages 50–51. ENSCI - Les Ateliers / Paris Design Lab, December 2012.

[21] Diemo Schwarz. Corpus-based concatenative synthesis. *IEEE Signal Processing Magazine*, 24(2):92–104, March 2007. Special Section: Signal Processing for Sound Synthesis.

[22] Diemo Schwarz and Norbert Schnell. A modular sound descriptor analysis framework for relaxed-real-time applications. In *Proceedings of the International Computer Music Conference (ICMC)*, New York, NY, 2010.