VISR User documentation

The S3A project team

Nov 16, 2018

CONTENTS

CHAPTER

ONE

ABOUT

The VISR framework is a collection of software for audio processing that forms the backbone for most of the technology created in S3A. In this extensible software framework, complex audio algorithms can be formed by interconnecting existing building blocks, termed components.

It can be used either interactively in the Python language, in custom applications (for instance in written C++, or integrated into other applications, for instance as DAW plugins or Max/MSP externals. While the VISR provides several renderers and building blocks for spatial and object-based audio, it is nonetheless a generic audio processing framework that can be used in other applications, for example array processing or hearing aid prototypes. The Python integration makes the system accessible, and enables easy algorithm development and prototyping.

CHAPTER

TWO

GETTING STARTED WITH THE VISR FRAMEWORK: OVERVIEW

2.1 Python integration

CHAPTER

THREE

VISR TUTORIAL USING PYTHON

Note: This tutorial is an extended version of the paper: Andreas Franck and Filippo Maria Fazi. VISR – a versatile open software framework for audio signal processing. In Proc. Audio Eng. Soc. 2018 Int. Conf. Spatial Reproduction. Tokyo, Japan, August 2018. http://www.aes.org/e-lib/browse.cfm?elib=19628

Software plays an increasingly important role in spatial and object-based audio. Realtime and interactive rendering is often needed to subjectively evaluate and demonstrate algorithms, requiring significant implementation effort and often impeding the reproducibility of scientific research. In this paper we present the VISR (Versatile Interactive Scene Renderer) – a modular, open-source software framework for audio processing. VISR enables systematic reuse of DSP functionality, rapid prototyping in C++ or Python, and integration into the typical workflow of audio research and development from initial implementation and offline objective evaluation to subjective testing. This paper provides a practical, example-based introduction to the VISR framework. This is demonstrated with an interconnected example, from algorithm design and implementation, dynamic binaural auralization, to a subjective test.

3.1 Introduction

Many areas of audio research depend heavily on software for audio processing and rendering. This includes not only research on sound reproduction, but also basic and applied research that use audio reproduction as a tool. Very often, this rendering must be responsive and interactive, requiring realtime-capable software tools. Combined with the number of DSP building blocks required for most rendering approaches, this entails a significant implementation effort for many audio research tasks. At the same time, the increasing importance of reproducible research imposes more demanding requirements on software in order to enable others to reproduce and evaluate your results or to use them in their work *[T1][T2]*. Software reuse is a central aspect to tackle these challenges.

Therefore we introduce the VISR (Versatile Interactive Scene Renderer), a novel portable, modular, and open-source software framework for audio processing and reproduction. Created in the S3A project (http://www.s3a-spatialaudio.org), e.g., *[T3]*, with focus on multichannel spatial and object-based audio, it is nonetheless a generic, application-agnostic framework. It consists of a set of pre-configured rendering signal flow, a library of DSP and audio processing building blocks, and supporting software to run these components in both realtime and offline environments. As a software framework, extensibility by users is a central aspect of the VISR. This can be done by arranging existing building blocks in new ways, incorporating new atomic functionality, or combinations thereof.

There are many existing software projects for audio processing, from DSP and sound synthesis languages as Faust *[T4]*, CSound *[T5]*, and SuperCollider *[T6]*; libraries of spatial processing components as IRCAM's Spat *[T7]*; frameworks as CLAM *[T8]*; Matlab software as the Audio System Toolbox; or rendering applications as the SoundScape Renderer *[T9]*.

Compared to these projects, we believe that the main advantages of VISR emerge as a combination of the following features: Firstly, its relatively high level of abstraction based on a component-based, object-oriented architecture enables the creation of complex audio processing schemes with relative ease. Secondly, the VISR is open-source and highly portable, supporting Windows, Mac OS X, and Linux including single-board computers as the Raspberry Pi, as well as integration into software environments as digital audio workstations (DAWs) or Max/MSP.

Thirdly, the use of Python as an additional implementation language significantly improves the productivity of using the framework and makes it more approachable to users that are not expert programmers. Finally, and closely related to the Python integration, the VISR framework allows a seamless integration of audio algorithm development with interactive realtime rendering and subjective evaluation.

3.2 The VISR Framework

This section explains the main concepts and entities in the VISR framework.

3.2.1 Component-Based Design

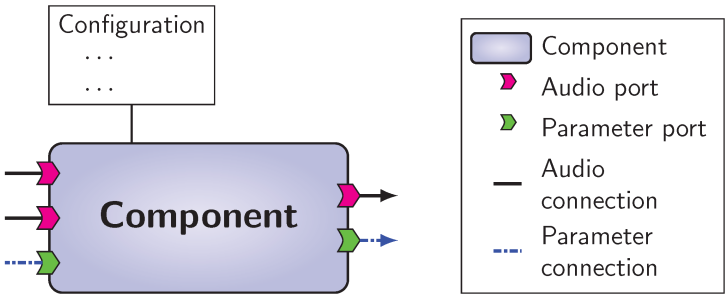

Fig. 1: General interface of a VISR component.

VISR is a software framework, which means that it enables a systematic reuse of functionalities and is designed for extension by users. To this, all processing tasks are implemented within *components*, software entities that communicate with other components and the external environment through a defined, common interface. Figure *General interface of a VISR component.* depicts the general structure of a component. Configuration parameters are passed to the component's constructor to customize its behavior. The external interface of components is defined by ports, which can be connected to other components or may represent external communication.

3.2.2 Audio and Parameter Ports

Ports represent data inputs and outputs of a component. They enable a configurable, directional flow of information between components or with the outside environment. There are two distinct types of ports: audio and parameter ports. Audio ports receive or create multichannel audio signals with an arbitrary, configurable number of channels (single audio signal waveforms), which is referred as the *width* of the port. Audio ports are configured with a unique name, a width and a sample type such as `float` or `int16`.

Parameter ports, on the other hand, convey control and parameter information between components or from and to the external environment. Parameter data is significantly more diverse than audio data. For example, parameter data used in the BST includes vectors of gain or delay values, FIR or IIR filter coefficients, audio object metadata, and structures to represent the listener's orientation. In addition to the data type, there are also different communication semantics for parameters. For example, data can change in each iteration of the audio processing, be updated only sporadically, or communicated through messages queues. In VISR, these semantics are termed *communication protocols* and form an additional property of a parameter port. The semantics described above are implemented by the communication protocols `SharedData`, `DoubleBuffering`, and `MessageQueue`, respectively.

Several parameter types feature additional configuration data, such as the dimensions of a matrix parameter. In the VISR framework, such options are passed in `ParameterConfig` objects. This allows extensive type checking, for instance to ensure that only matrix parameters of matching dimensions are connected. Combining these features, a parameter port is described by these properties: a unique name, a parameter type, a communication protocol type and an optional parameter configuration object.

3.2.3 Atomic Components

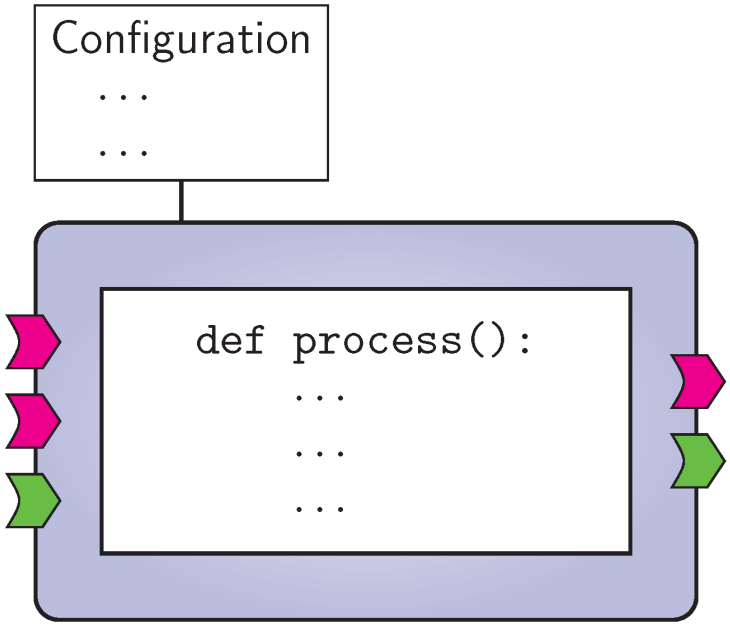

Fig. 2: VISR atomic component.

To create and reuse more complex functionality out of existing building blocks, VISR signal flows can be structured hierarchically. To this end, there are two different kinds of components in VISR, *atomic* and *composite*. They have the same external interface, that means that they can be used in the same way. Fig. *VISR atomic component.* schematically depicts these two types. *Atomic components* implement processing task in program code, e.g., in C++ or Python. They feature a constructor which may take a variety of configuration options to tailor the behaviour of the component and to initialize its state. The operation of an atomic component is implemented in the `process()` method. It typically involves accessing audio input data, performing DSP operations on it and writing it to audio outputs, or receiving, manipulating, and creating parameter data. Examples how atomic components are implemented are given in Section *Prototyping Atomic functionality*.

3.2.4 Composite Components

In contrast, a *composite component* contains a set of interconnected components (atomic or composite) to define its behavior. This is depicted in Figure *VISR composite component.*. This allows the specification of more complex signal flows in terms of existing functionality, but also the reuse of such complex signal flows. As their atomic counterparts, they may take a rich set of constructor options. These can control which contained components are constructed, how they are configured, and how they are connected. It is worth noting that nested components do not impair computational efficiency because the hierarchy is flattened at initialization time and therefore not visible to the runtime engine.

Using hierarchy to structure signal flows is a powerful and central technique in the VISR framework. As explained in more detail later, its uses include the reuse of processing functionality and enabling the use of components in different software environments.

3.2.5 Standard Component Library

The runtime component library (`rcl`) of the VISR framework contains a number of components for general-purpose DSP and object-based audio operations. They are typically implemented in C++ and therefore relatively efficient. The `rcl` library includes arithmetic operations on multichannel signals, gain vectors and matrices, delay lines, FIR and IIR filtering blocks, but also network senders and receivers and components for decoding and handling of object audio metadata.

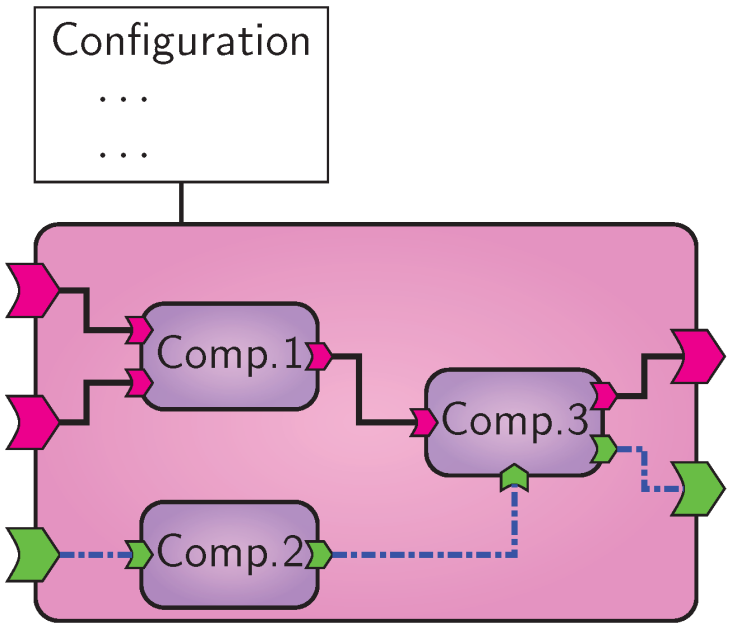

Fig. 3: VISR composite component.

3.2.6 Runtime Engine

A key objective of the VISR framework is to enable users to focus on their processing task – performed in a component – while automating tedious tasks, such as error checking, communication between components, or interfacing audio hardware, as far as possible. The rendering runtime library (`rrl`) serves this purpose. Starting from a top-level component, it is only necessary to construct an object of type `AudioSignalFlow` for this component. All operations from consistency checking to the initialization of memory buffers and data structures for rendering is performed by this object. The `audiointerfaces` library provides abstractions for different audio interface APIs (such as Jack, PortAudio, or ASIO). Realtime rendering is started by connecting the `SignalFlow` object to an `audiointerfaces` object.

3.2.7 Python interface

While the core of the VISR framework is implemented in C++, it provides a full application programming interface (API) for the Python programming language. This is to enable users to adapt or extend signal flows more productively, using an interpreted language with a more accessible, readable syntax and enabling the use of rich libraries for numeric computing and DSP, such as NumPy and SciPy *[T10]*. The Python API can be used in three principal ways:

Configuring and running signal flows

Components can be created and configured from the interactive Python interpreters or script files. This makes this task fully programmable and removes the need for external scripts to configure renders. In the same way, audio interfaces can be configured, instantiated and started from within Python, enabling realtime rendering from within an interactive interpreter.

Extending and creating signal flow

As described above, complex signal flows are typically created as composite components. This can be done in Python by deriving a class from the base class `visr.CompositeComponent`. The behavior of the signal flow is defined in the class' constructor by creating external ports, contained components, and their interconnections. Instances of this class can be used for realtime rendering from the Python interpreter, as described above, or from a standalone application.

Adding atomic functionality

In the same way as composites, atomic components can be implemented by deriving from `visr.AtomicComponent`. This involves implementing the constructor set up the component and the `process()` method that performs the run-time processing. The resulting objects can be embedded in either Python or C++ composite components (via a helper class `PythonWrapper`).

Offline Rendering

By virtue of the Python integration, signal flows implemented as components are not limited to realtime rendering, but can also be executed in an offline programming environment. Because the top-level audio and parameter ports of a component can be accessed externally, dynamic rendering features such as moving objects or head movements can be simulated in a deterministic way. In the majority of uses, this is most conveniently performed in an interactive Python environment. Applications of this feature range from regression tests of atomic components or complex composite signal flows, performance simulations, to offline rendering of complete sound scenes.

Use in multiple software environments

The VISR framework aims to ease the reuse of audio processing functionality in different software environments, for instance as plugins for digital audio workstations (DAWs) or as blocks in visual programming languages as Max/MSP. The main goal is to make the functionality available to a larger group of users, enable them to integrate them into their workflow, and to use it as building blocks for their applications. Moreover, providing software components that are integrated into a host environment often reduces the complexity of a audio processing system, as opposed to a standalone renderer that requires a separate binary, dedicated connections, and possibly an additional computer.

Such an integration is enabled both by architectural features of the framework and support libraries that are part of VISR. Firstly, the modular design, in particular the component abstraction which is independent of a specific audio API or a specific model of execution, enables running VISR components within a multitude of software environments. Secondly, the extensible parameter subsystem that makes control data input and output accessible from outside the components eases a translation to the host-specific communication mechanism. Finally, VISR contains a set of support libraries that simplifies the translation between the VISR interfaces and the API of the host environment and reduces the necessary amount of code for wrapping a VISR component in, e.g, a VST plugin or a Max/MSP external.

3.3 Application Example: Panning Algorithm Development

In this section we explain the use of the VISR framework in a continued application example. To this end we describe the prototyping and testing of a multichannel amplitude panning technique. For expressiveness and conciseness, the examples are presented in the Python language. However, the same functionality could also be achieved in C++ at the expense of an increased code size and a steeper learning curve. This tutorial example displays only relevant code sections, sometimes in abridged form. The full source code is available through *[T11]*.

3.3.1 Obtaining and installing the VISR framework

The VISR framework is available under a permissive open-source license which allows for free use and modification. Installation packages and setup instructions are provided in section *Getting VISR*. To use the Python integration, Python 3 must be installed. For Windows ans Mac OS we recommend the Anaconda distribution (https://anaconda.org/). Note that the installer must match the Python major and minor version number on the target system, e.g., Python 3.5.

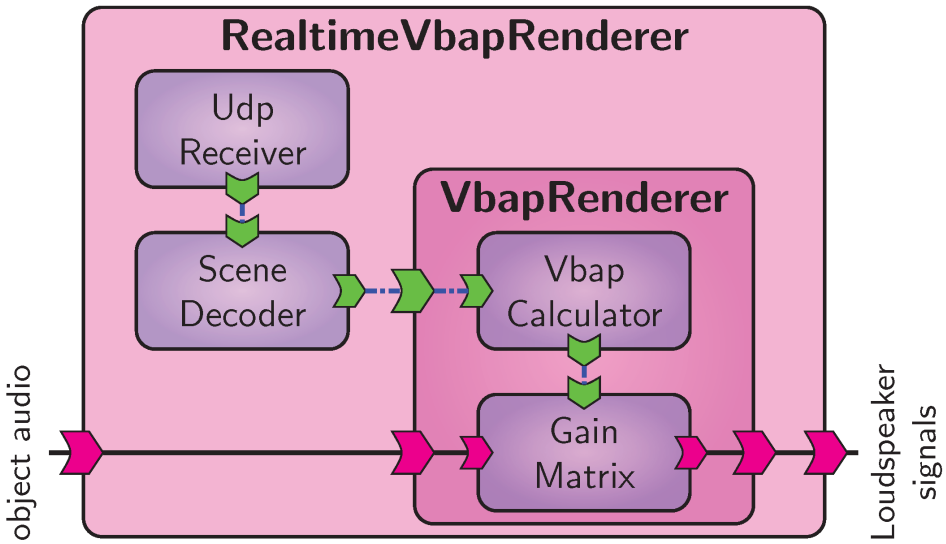

Fig. 4: Basic amplitude panning system for real-time rendering.

Creating and Adapting Signal Flows

The most basic way to create and adapt audio processing functionality in the VISR framework is to parameterize and connect existing building blocks. To this end we create a new `CompositeComponent`. We demonstrate that by building a simple multichannel amplitude panning renderer, which is schematically depicted in Figure *Basic amplitude panning system for real-time rendering.*. The corresponding source file is `vbap_renderer.py`.

```
import visr, pml, rcl
class VbapRenderer( visr.CompositeComponent ):
```

This starts by importing several Python modules that provide VISR functionality, namely `visr` for the core API, `rcl` for the built-in component library (see Section *Standard Component Library*) and the `pml` for parameter data and communication protocols (Section *Audio and Parameter Ports*). The signal flow class `VbapRenderer` is derived from the VISR base class `visr.CompositeComponent`.

```
def __init__( context, name, parent,
              numObjects, lspConfig ):
  super().__init__( context, name, parent )
```

As explained in Section *Composite Components*, the functionality of a composite component is embodied in its constructor, which defines external inputs and outputs, the contained components, and their connections. Here, the constructor takes the standard parameters `context` to specify audio block size and sampling rate, `name` to assign a unique name, and `parent` for an optional parent component (`None` denotes a top-level flow). The class-specific parameters `numObjects` and `lspConfig` customize instances of this specific class. The call `super().__init__(...)` initializes the base class object..

```
self.audioIn = visr.AudioInputFloat( "in",
  self, numObjects )
self.audioOut = visr.AudioOutputFloat( "out",
  self, numLsp )
self.objectIn = visr.ParameterInput("objects",self,
    pml.ObjectVector.staticType,
    pml.DoubleBufferingProtocol.staticType,
    pml.EmptyParameterConfig() )
```

Here, we define an audio input and an audio output as well as an parameter input to receive object metadata. The simple panning renderer contains two components, `VbapGainCalculator` and `GainMatrix`. The former calculates a matrix of panning gains from set of object metadata, and the latter applies these gains to a set of object audio signals to form loudspeaker outputs. These components are instantiated as members variables of the

VbapRenderer class.

```
self.calculator = rcl.PanningCalculator(
  context, "VbapGainCalculator", self,
  numberOfObjects, lspConfig )
self.matrix = rcl.GainMatrix( context,
  "GainMatrix", self, numberOfObjects, numLsp
  interpolationSteps=context.period )
```

Finally, connection between the components' input and output ports as well as external ports are defined by using API methods of CompositeComponent.

```
self.audioConnection( self.audioIn,
  self.matrix.audioPort("in") )
self.audioConnection(self.matrix.audioPort("out"),
self.audioOut)
self.parameterConnection(self.objectIn,
  self.calculator.parameterPort("objectIn"))
self.parameterConnection(
  self.calculator.parameterPort("gainOutput"),
  self.matrix.parameterPort("gainInput" ))
```

This example shows how complex, multichannel signal flows can be readily created in VISR using high-level building blocks and sophisticated multichannel audio and data connections.

To use the VbapRenderer in a realtime setting, we must provide means to receive audio object metadata, e.g., from an audio workstation. In the VISR framework, this is best done by creating a new top-level composite component, denoted RealtimeVbapRenderer in Figure *Basic amplitude panning system for real-time rendering.*, that contains the VbapRenderer and means to receive and decode object metadata from network messages. This is done by the components UdpReceiver and SceneDecoder, both contained in the VISR rcl library. Organizing functionality hierarchically into composite components, including moving supplemental tasks as network communication outside the core algorithm, is recurring design pattern in VISR signal flows. As demonstrated in later sections, it fosters reuse of functionality and helps to use the same algorithm in different software environments.

Realtime Execution and Binaural Auralization

Components – both composite and atomic – can be readily used for realtime rendering, both as a standalone application or from a Python session. We first describe the latter, interactive approach. To run a VISR component, we first construct an object of type rrl.AudioSignalFlow.

```
import rrl
...
flow = rrl.AudioSignalFlow( component )
```

It contains all information and data structures needed to execute the component's signal flow. During construction, it flattens hierarchical signal flows and performs consistency checks.

In a second step we create an AudioInterface object representing an audio device, e.g., a sound card.

```
flow = rrl.AudioSignalFlow( renderer )
aiConfig = ai.AudioInterface.Configuration(
aIfc = ai.AudioInterfaceFactory.create("PortAudio",
 numberOfInputs=numObjects, numberOfOutputs=2,
 samplingFrequency=fs, period=bs )
aIfc.registerCallback( flow )
aIfc.start()
```

At the moment the audiointerfaces library contains classes for two backends, namely the cross-platform PortAudio library (http://www.portaudio.com/) and the Jack sound server (http://www.jackaudio.org/) (Linux and Mac OS X). Additional backends can be implemented in the future, which is supported by the

`AudioInterfaceFactory` factory interface to instantiate audio interfaces. This method accepts also an optional `optionalConfig` parameter to pass backend-specific options.

The second way to run a VISR signal flow is to build a standalone application, typically in C++. This application would create an instance of the VISR component to be executed, and could use the functionality of the `audiointerfaces` and `rrl` libraries to perform the realtime rendering. For top-level components implemented in Python, a simpler way exists by using the `python_wrapper` application that is included in the VISR distribution.

```
python_runner -m vbap_renderer
 -c RealtimeVbapRenderer -a "2,'../data/stereo.xml'"
 -k "{'nwPort': 4242}" -D PortAudio
```

Here, `vbap_renderer` is the name of the Python module, `RealtimeVbapRenderer` is the name of class name of a top-level component implemented in this module, `$-$D PortAudio` denotes the audio interface to be used, while `$-$a` and `$-$k` are used to pass positional and keyword arguments to the component's constructor. For this to work, the main module (`vbap_renderer` in this example) and other used Python modules must be contained in the Python module search path, either by setting the `PYTHONHOME` environment variable or by passing the path via the `$-$d` option. In this way, the `python_runner` utility app allows for an easy use of VISR signal flows specified in Python without requiring an interactive Python interpreter.

Because multi-loudspeaker panning algorithms require sophisticated setups, they are difficult to auralize. The VISR framework makes it easy to couple an algorithm with tools for auralization over headphones using dynamic binaural synthesis, e.g., *[T12][T13]*. The VISR framework allows for a straightforward integration of such tools. Here we use the Binaural Synthesis Toolkit *[T14]*, a set of components for binaural rendering implemented in the VISR framework The BST implements different approaches to binaural synthesis, namely dynamic HRTF-based rendering, synthesis based on Higher Order Ambisonics, and virtual loudspeaker rendering (or binaural rool scanning). Here we use the `VirtualLoudspeakerRenderer` component that transforms a set of loudspeaker signals into a binaural signal.

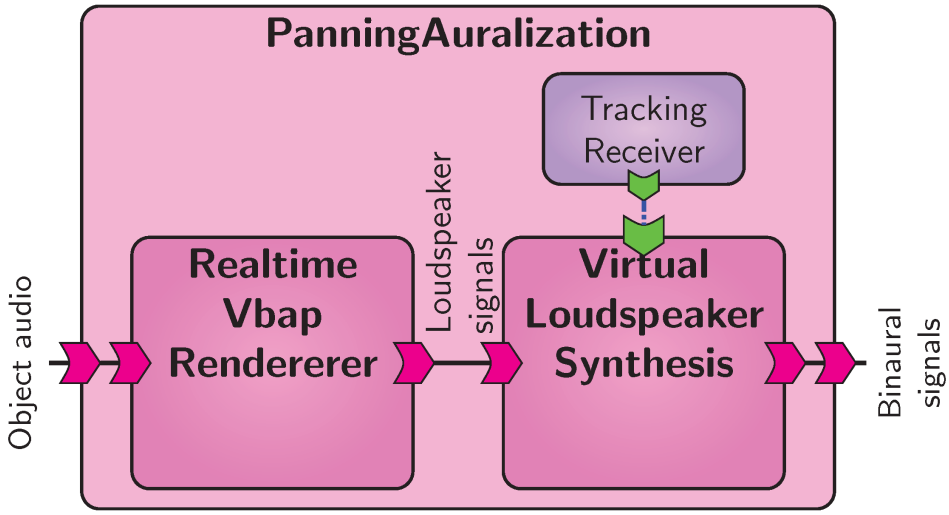

Fig. 5: Panning algorithm auralization.

To this end, we create a new top-level component depicted in Figure *Panning algorithm auralization.*. It is composed of the `RealtimeVbapRenderer` component whose loudspeaker signal output is connected to the `VirtualLoudspeakerRenderer` instance. The latter can also receive listener tracking information to incorporate the listener's head orientation. In this case, a `TrackingReceiver` component specific to the tracking system used is instantiated and connected to the binaural module.

It is noted that the two parts could also be instantiated as separate realtime applications, and connected using a sound server as Jack. VISR's ability to combine the components, however, makes such applications less dependent on the capabilities of the operating system and reduces the effort to control such complex configurations.

Prototyping Atomic functionality

In addition to creating new functionality by interconnecting existing components, the VISR framework can be extended by implementing new primitive (or atomic) functionality as C++ or Python code. This is done by creating new atomic components, which can then be used either standalone or combined with other VISR components. In this section we demonstrate this functionality by prototyping a novel multi-loudspeaker panner – an algorithm for calculating loudspeaker gains – as an atomic component in Python. The novelty of this approach is that it uses a convex optimization algorithm to compute a solution with similar properties to VBAP, but uses ℓ_2 optimality to resolve ambiguity issues of VBAP, see *[T15]*. Using the VISR framework it is possible to use the rich set of libraries for technical and scientific computing available in Python, in this case the numerical optimization package **cvxpy** *[T16]*, for algorithm prototyping and realtime evaluation. The source code can be found in `vbap_l2_panner.py`.

An atomic component is implemented as a class derived from `visr.AtomicComponent`

```
class VbapL2Panner( visr.AtomicComponent ):
  ...
```

The component has a constructor similar to that of a composite component, taking the mandatory arguments `context`, `name` and `parent` plus a custom set of configuration parameters.

```
def __init__( self, context, name, parent,
  numObjects, lspConfig, spread ):
  super().__init( context, name, parent )
  self.objectIn = pml.ParameterInput( "objects", self,
    pml.DoubleBuffering.type, pml.ObjectVector.type,
    pml.EmptyParameterConfig() )
  ...
  self.L = lspConfig
  ...
  self.problem = cvxpy.Problem( self.objective,
                                self.constraints )
  ...
```

Like in composite components, the `__init__()` method calls the constructor of the superclass and creates audio and parameter ports. In addition, it sets up any internal data structures, for instance storing the loudspeaker positions in `self.L` and setting up the optimization problem as `self.problem`. As described in Section *Atomic Components*, atomic components implement their behavior in the `process()` method.

```
def process( self ):
  if self.objectIn.protocol.changed():
    self.objectIn.protocol.resetChanged()
    objVec = self.objectIn.protocol.data()
    gains = np.array( self.gainOut.protocol.data() )
    pointSources = [o for o in objVec
     if isinstance( o, objectmodel.PointSource )]
    for obj in pointSources:
      self.b.value = obj.position
      self.problem.solve()
      gains[:,obj.id] = normalise( np.squeeze(
        np.asarray(self.g.value) ), norm=2 )
```

Here, the code first gains access to the object vector input and checks if the input has changed. In this case, both the current object vector and the output gain matrix are obtained, and the computation is performed. This consists of calling the optimization method for each point source object and assigning the resulting gain vector to the respective row in the panning gain matrix. The resulting component can be used in the same way as a built-in atomic component. That is, it can be executed as a standalone top-level component or in a composite flow defined in either Python or C++.

As an example for an audio-processing atomic component we show a simplified gain matrix as used in the VBAP renderers.

```
class GainMatrix( visr.AtomicComponent ):
  def __init__(self,context,name,parent,nIn,nOut):
   super().__init__( context, name, parent )
   self.audioIn=visr.AudioInputFloat("in",self,nIn)
   self.audioOut=visr.AudioOutputFloat("out",self,nOut)
   self.mtxIn=visr.ParameterInput( "gainInput",
     self, pml.MatrixParameterFloat.staticType,
     pml.SharedDataProtocol.staticType,
     pml.MatrixParameterConfig(nOut, nIn ))
  def process( self ):
   gains  = np.array( self.mtxIn.protocol.data() )
   ins = self.audioIn.data()
   self.audioOut.set( gains <at> ins )
```

It defines an audio input and an audio output with distinct widths and a parameter input to receive the gain coefficients. The `process()` function accesses the sample data of the audio ports as **NumPy** arrays and uses the matrix multiplication operator `@` to calculate the result. This shows how numeric and DSP functionality can be prototyped on a high level of abstraction. But if required, the `AudioPort` interface also enables access to individual audio channels and samples.

Offline Testing and Objective Evaluation

As explained in Section *Offline Rendering*, VISR enables the offline execution and analysis of components, both C++ and Python, within an interactive Python environment This allows for the use of the same source code for algorithm development, offline processing, and realtime rendering. In this section we show how this feature is used to design and evaluate the panning component created in Section *Prototyping Atomic functionality*. The source code is contained in the file `simulate_l2_renderer.py`.

The code for the offline simulation is very similar to the realtime case shown in Section *Realtime Execution and Binaural Auralization*. As there, the main tasks consist of configuring and creating the top-level component and the creation of an `AudioSignalFlow` object. But instead of instantiating and registering to an audio interface, the multichannel audio input is provided as a matrix of samples, and another matrix is provided for the output signals.

```
fs,inSignal=scipy.io.wavfile.read('test.wav').T
sigLen=inSignal.shape[-1]
outSignal=np.zeros( (numLsp, sigLen ) )
```

Parameter data is transferred to and from top-level parameter ports by retrieving their communication protocol endpoints through the AudioSignalFlow object. In this example, the signal flow has a parameter input named `objects` to receive object metadata.

```
objectIn = flowparameterReceivePort('objects')
```

Data can be sent and received using the semantics of the port's communication protocol.

```
numBlocks = sigLen // bs
for bi in range(0,numBlocks):
  ps = objectmodel.PointSource(0)
  ps.pos = trajectory[bi,:]
  ps.level = 1
  ps.channels = [0]
  objectIn.data().set([ps])
  objectIn.swapBuffers()
  outSignal[:,bi*bs:(bi+1)*bs]
     = flow.process( inSignal[:,bi*bs:(bi+1)*bs] )
```

Here, the audio signal is partitioned into a number of blocks. For each block, a point source object with a new position is created and transmitted to the rendering component using the methods `set()` and `swapBuffers()`

of the object vector input port. Then the signal flow is executed for one block of input data per iteration and the generated audio samples are concatenated into a multichannel output signal.

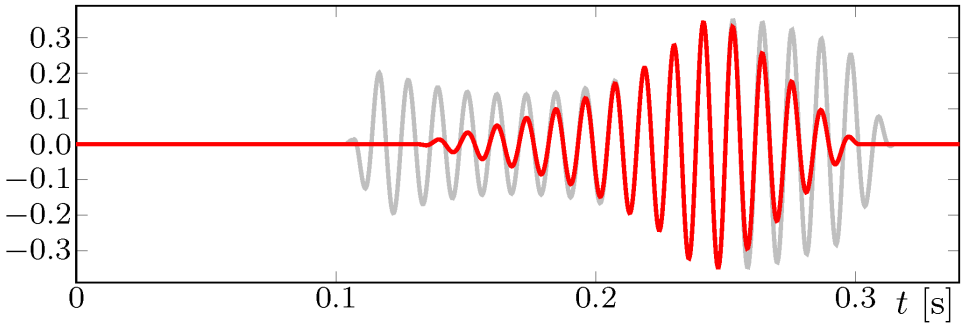

Fig. 6: Offline simulation plots of the proposed panning algorithm: Audio signal.

Figure *Offline simulation plots of the proposed panning algorithm: Audio signal.* shows the output signal for one loudspeaker (top-rear right position `U-135`)) of a 9-loudspeaker setup according to ITU-R BS.2051 *[T17]* and compares it to the output of a standard VBAP implementation. The sound object is a sinusoidal tone rotating around the central listener at a constant elevation of 10°. The plot shows that the proposed algorithm yields a more localized, monotonically ancreasing and decreasing signal magnitude as the source psition moves, whereas in the VBAP algorithm the same loudspeaker is activated for a wider range of source position and exhibits a fluctuating magnitude. Audio signals obtained in this way can be used for playback or used as input for further analysis, e.g., by using binaural localization models.

In addition to analyzing audio signals, it is also possible to analyze smaller building blocks and their input and output parameter data. Two features of the VISR framework make this possible. Firstly, all building blocks are components. That means that they can be inspected at all levels of the hierarchy, from complex top-level signal flows down to atomic algorithms. Secondly, by exposing the extensible parameter data subsystem to the Python language, the VISR framework enables the generation of sophisticated control data trajectories and the evaluation of parameter data generated by the signal flow in an interactive programming environment. Here, we test the `VbapL2Panner` component separately, setting object positions for the same set of azimuths as in the previous example.

```
pannerVbapL2 = VbapL2Panner( ctxt, 'renderer',
   None, numObjects, lc )
flow = rrl.AudioSignalFlow( pannerVbapL2 )
objectIn = flow.parameterReceivePort('objects')
gainOut = flow.parameterSendPort('gains')
gainsVbapL2 = np.zeros( (numLsp,len(az) )
for bi in range(0,numBlocks):
  ps1 = ojectmodel.PointSource(0)
  ps1.position = sph2cart( az[bi], el, r )
  ps1.channels = [0]; ps1.level=1.0
  objectIn.data().set( [ps1] )
  objectIn.swapBuffers()
  flow.process()
  gainsVbapL2[:,bi] =  np.squeeze(gainOut.data())
```

The flow object is executed for each setting of the azimuth value. Since the `VbapL2Panner` component has no audio ports, the call `flow.process()` does not involve audio signals. Instead, the protocol of the parameter output `"gains"` is accessed and the panning gains for all azimuths are collected in a matrix.

Figure *Offline simulation plots of the proposed panning algorithm: Panning gains.* shows the resulting gains of the `VbapL2Panner` and the standard VBAP algorithm for the upper-rear loudspeakers `U-135` and `U+135` as a function of the source's azimuth. For plain VBAP, the gains for the symmetrical loudspeaker positions are asymmetrical, and `U-135` is active for a wide range of azimuths with some rapid gain changes. This is in accordance with the audio signal depicted in Figure *Offline simulation plots of the proposed panning algorithm: Audio signal.*, and is caused by ambiguities inherent to the VBAP algorithm, see e.g., *[T18]*. In contrast, the proposed algorithm offers symmetric, localized loudspeaker activations that with smoother transitions as the source moves.

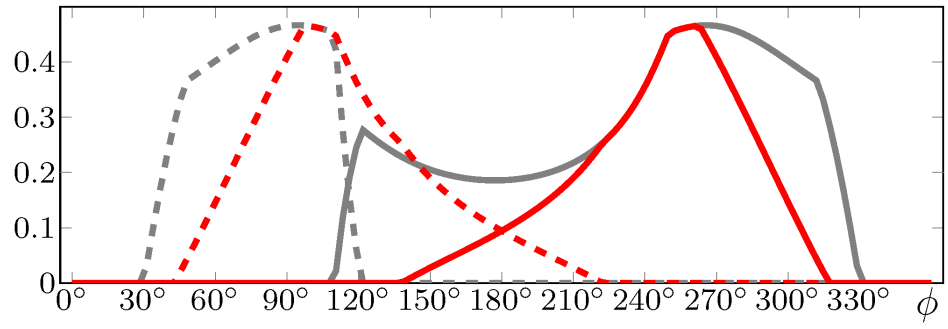

Fig. 7: Offline simulation plots of the proposed panning algorithm: Panning gains.

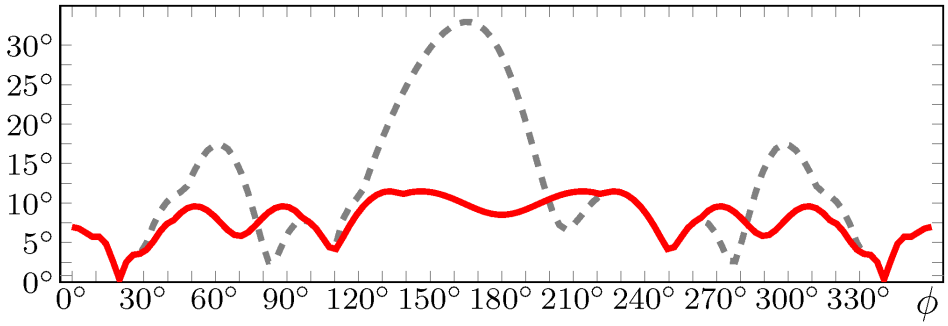

Fig. 8: Offline simulation plots of the proposed panning algorithm: Energy vector direction difference.

Access to such data also enables more sophisticated analyses. As an example, Figure *Offline simulation plots of the proposed panning algorithm: Energy vector direction difference.* shows the difference of the energy vector, a metric to estimate the localisation of mid-to-high frequency content, for both algorithms. It shows that the error is smaller and more even for the proposed VBAP L2 algorithm than for the standard algorithm. This corroborates that the proposed panning scheme overcomes some shortcomings of the plain VBAP algorithm. This example shows how VISR's offline execution features can ease the development, testing, and evaluation of audio signal processing components, from algorithmic building blocks to complex signal flows. By using the same implementation as for realtime rendering, this offers the potential of unifying conventional signal processing development and realtime-capable implementation.

Subjective Listening Test Application

In this section we demonstrate how audio processing algorithms implemented as VISR components can be readily integrated into larger, interactive applications and graphical user interfaces. To this end we create a listening test tool, which allows for a subjective evaluation of the proposed panning algorithm and comparison to existing approaches by presenting multiple stimuli as standardised, e.g., in ITU-R BS.1534 *[T17]*.

With the VISR framework, the signal processing is implemented as a new top-level composite component, shown in Figure *Subjective listening test application: VISR signal flow.*, which contains the proposed renderer as one component, alongside renderers for other rendering schemes. The top-level component provides audio and object metadata to all candidate algorithms, and facilitates an multichannel selector for glitch-free switching between the methods.

In this example the graphical user interface, shown in Figure *Subjective listening test application: Python user interface.*, is implemented in Python (using the PyQt library), and the rendering is embedded into this application within a `rrl.AudioSignalFlow` instance. The flexible parameter subsystem allows for an interactive control of the media playback and the rendering method selection. To this end, control data is sent directly from the UI code to the parameter ports `transport` and `switch`. This example shows how the use of the VISR framework can reduce the time and effort needed to perform a subjective evaluation of audio processing algorithms.

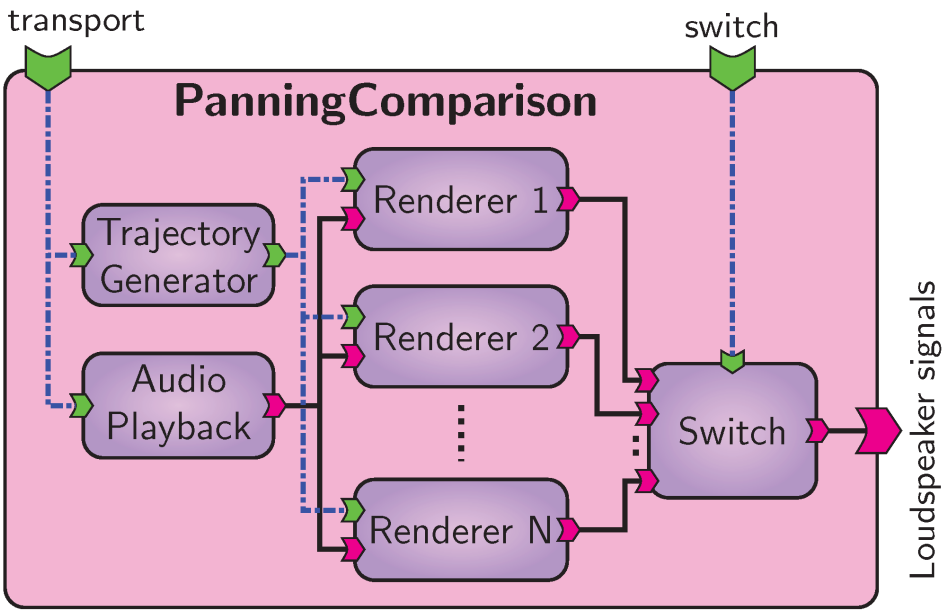

Fig. 9: Subjective listening test application: VISR signal flow.

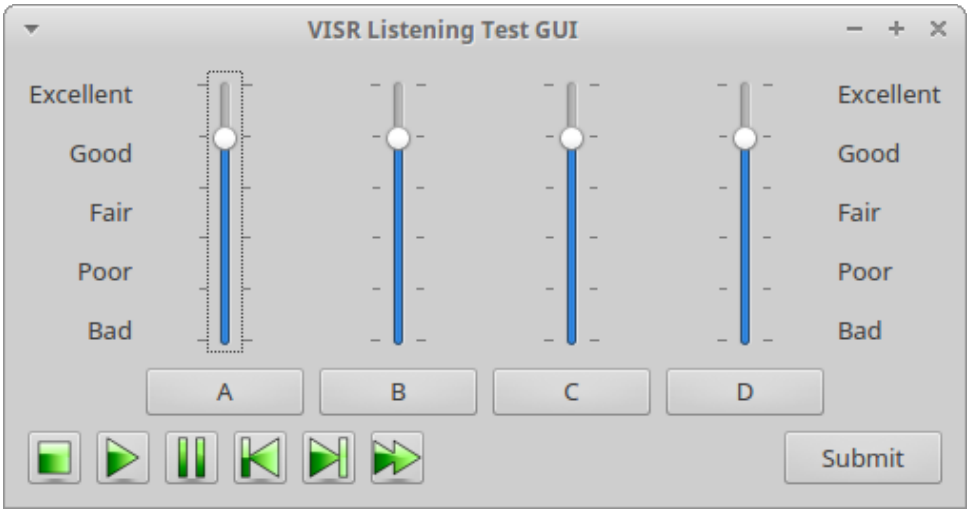

Fig. 10: Subjective listening test application: Python user interface.

Use in Different Software Environments

As explained in Section *Use in multiple software environments*, the VISR framework enables the use of audio processing components by embedding them into other audio software environments.

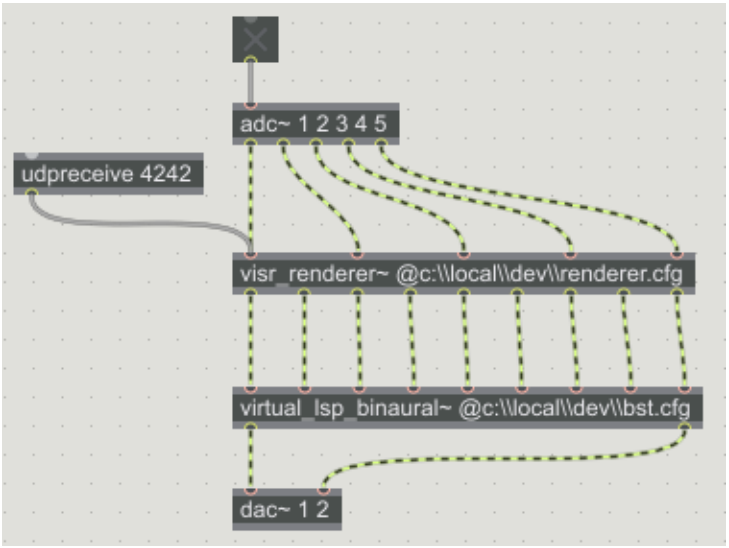

Fig. 11: Interactive loudspeaker rendering binauralization implemented in Max/MSP using VISR components as externals.

Figure *Interactive loudspeaker rendering binauralization implemented in Max/MSP using VISR components as externals.* shows an example of an interactive multichannel loudspeaker rendering combined with a binaural auralization, both implemented as Max/MSP externals. The input to the `visr_renderer~` external is provided by a Max Patch that creates audio object metadata, which are transmitted to the renderer as string messages. The loudspeaker outputs are connected to the `virt_lsp_binaural~` binauralization algorithm, which receives head orientation data from a tracking device through a serial port. This example shows how VISR components can be integrated into interactive, possibly audio-visual applications.

The second example, depicted in Figure *DAW plugin for object-based rendering using VISR components.*, shows a DAW plugin, that renders a point source object to a multi-loudspeaker setup. The rendering is performed by a VISR component within the plugin. VISR's flexible control parameter subsystem allows for controlling object parameters as the point source position and to connect them to automation parameters of the DAW.

3.4 Conclusion and Outlook

In this paper we introduced the VISR framework, an open-source portable and general-purpose framework for audio processing that is well-suited for multichannel, object-based, and spatial audio. Based on a continued application example of a multi-loudspeaker panning algorithm, we showed how the elements of the framework can be used to create and modify complex audio signal flows. Moreover, we explained how the extensibility features of the VISR, in particular the Python language interface, can be used to design, prototype, and evaluate novel audio processing algorithms, demonstrating how this can streamline a typical workflow in audio research and development. Finally, we demonstrated how audio processing functionality implemented in the VISR framework can be embedded in other software environments, for example Max/MSP or plugins for digital audio workstations.

Future developments will focus on the following aspects: Firstly, adding support for additional hardware platforms, operating system versions, and audio APIs, and improved multiprocessor support. Secondly, providing code libraries to ease the embedding of VISR components into software environments as Max/MSP, PureData or DAW plugin SDKs. Thirdly, to create libraries of building blocks and ready-made renderers for different application areas of audio processing.

We provide the VISR framework as an open-software framework to foster reproducible research in audio signal processing.

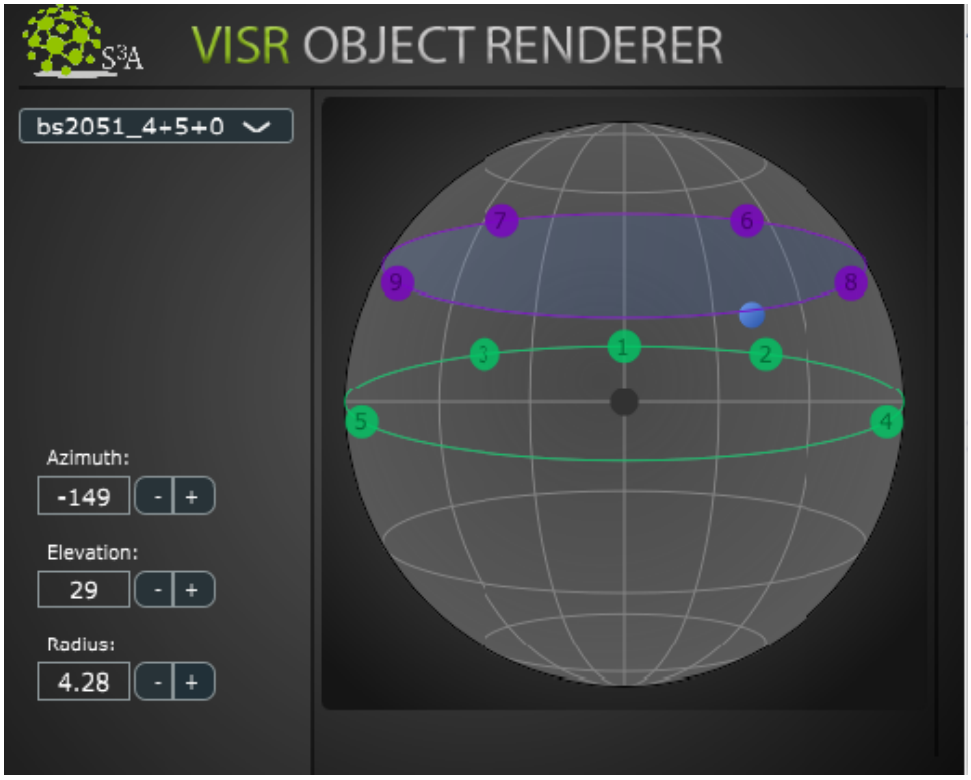

Fig. 12: DAW plugin for object-based rendering using VISR components.

3.5 References

CHAPTER

FOUR

PEOPLE

CHAPTER

FIVE

GETTING VISR

5.1 Download

The VISR framework can be obtained in diffeent forms. FOr most persons, however, downloading and installing an installer package is the most convenient way to use this framework.

Installation packages can be downloaded from the S3A software download page .

Installation packages are available for the following platforms:

Windows (x86_64) Recent versions (Windows 8 and Windows 10) 64 Bit only

Mac OS X Version 10.11 and above, 64 Bit only

Linux Ubuntu 16.04 LTS and Ubuntu 18.04 LTS, 64 bit

Rasspberry Pi (ARM) Raspbian Stretch, 32 Bit

5.2 Installing VISR

Binary installation packages are the suggested way to uses the VISR framework. A binary installers enables all uses of the framework, including

- Running standalone applications
- Using DAW plugins based on the VISR
- Using the Python interfaces and creating new functionality in Python
- Creating standalone applications and extension libraries in C++

Hint: Building the VISR from source is necessary only in these cases:

- Porting it to a platform where no binary installer exists
- Fixing or changing the internal workings of the framework.

Installation packages are available on the S3A Software download page.

Note: If you plan to use the Python integration of the VISR framework (see *Python integration*), you need to select an installation package matching the Python version you are using, for example `VISR-X.X.X-python36-Windows.exe`.

5.2.1 Windows

The graphical installer is provides as an `.exe` file and provides a dialog-based, component-enables installation. Figure *figure_windows_installer* shows the component selection dialog of the installer. The choices are detailed below in section *Installation components*.

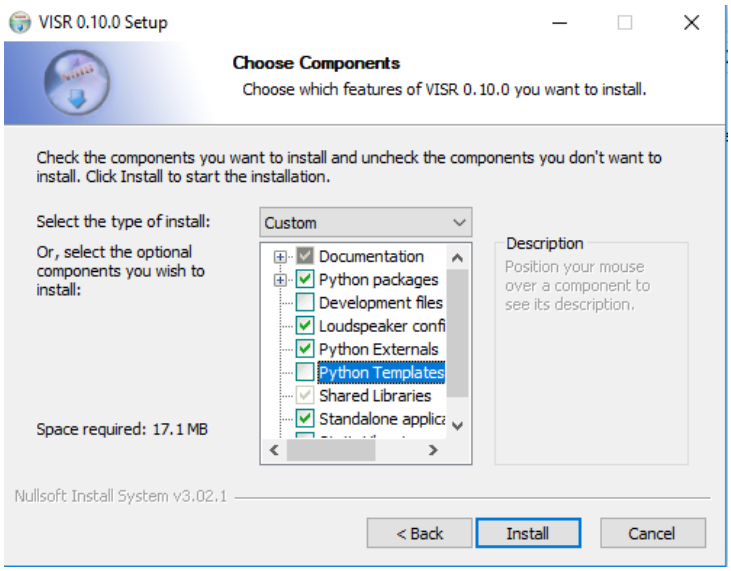

Fig. 1: Graphical Windows installer.

An executable installer (.exe) with a graphical user interface and corresponding uninstall functionality. Supported are 64-bit versions of Windows. If required, install the "Microsoft Visual C++ Redistributable for Visual Studio 2017", package, for example from the Visual C++ downloads page.

On Windows, it is necessary to add the directory containing the VISR libraries (DLLs) as well as the directory containing third-party libraries shipped with the VISR installer to the `PATH` variable. To this end, open the environment variable editor (Settings -> System -> Advanced system settings -> Environment variables). The environment variable on Windows 10 is depicted in figure windows_environment_variables_editor .

Append the value `C:\Program Files\VISR-X.X.X\lib;C:\Program Files\VISR-X.X.X\3rd` if the standard installation location was used (Note: Replace `X.X.X` with the actual version number of VISR). Depending on your system permissions and whether you VISR shall be used by all users of the computer, you can either set the `PATH` user variable or the `PATH` system variable.

Note: Any applications used to access VISR (for example command line terminals, Python development environments, or DAWs) must be closed and reopened before the changed paths take effect.

Append the path ''<install-directory>/lib" to the path variable, where ''install_diectory" is the directory specified during the installation. For the default path, the setting would be `c:\Program Files\VISR-N.N.N\lib`, where `N.N.N` is replaced by the actual version number. If the `PATH` variable is edited as a string, subsequent paths are separated by semicolons.

Note: Future versions of the installer might adjust the paths automatically. However, as pointed out in NSIS Path manipulation, this needs an extremely cautious implementation to avoid potential damage to users' systems.

To use standalone applications (see section *Using standalone applications*), it may be useful to add the `bin/` directory to the user or system path. For the default installation location, add `c:\Program Files\VISR-N.N.N\bin` to the `%PATH%` environment variable.

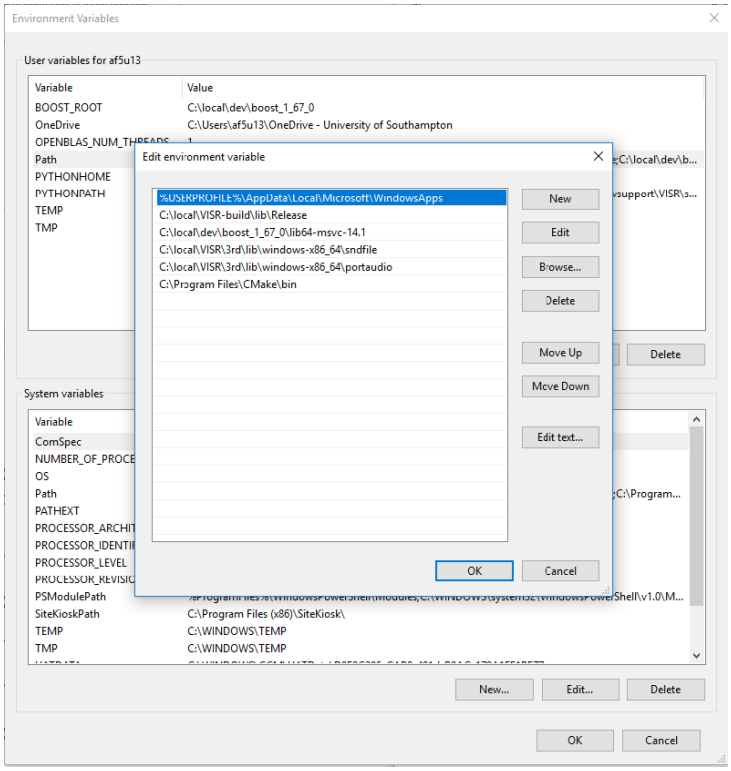

Fig. 2: Environment variable editor on Windows 10.

5.2.2 Mac OS X

An installer with a graphical user interface guides through the installation process and allows the selection of optional components. Figure *Component-based installer for Mac OS X.* shows a screenshot of this installer. By default, it installs the VISR into the directory `/Applications/VISR-X.X.X/` where `X.X.X` denotes the version number.

To access the component selection dialog, use the button "Customize" on the "Installation Type" screen (see figure *"Installation type" screen of Mac OS X installer. Use "Customize" to get to the component selection.*)

To use the standalone applications from the command line, the `bin/` subfolder of the installation directory, e.g., `/Applications/VISR-X.X.X/bin`. This can be done, for example, by adding

```
export PATH=$PATH:/Applications/VISR-X.X.X/bin
```

to the file `$HOME/\.bash_profile`. However, this works only for running standalone applications from a shell (i.e., a terminal window). If you need this path also from applications that are not started from a shell, we recommend the solution used in section *Configuration*.

5.2.3 Linux

For Linux, installation packages are provided as *.deb* (Debian) packages. At the moment, this package is monolithic, i.e., it contains all components. They are installed via the command

```
sudo apt install VISR-<version>.deb
```

If this command reports missing dependencies, these can be installed subsequently with the command

```
sudo apt install --fix-broken
```

After that the framework is ready to use.

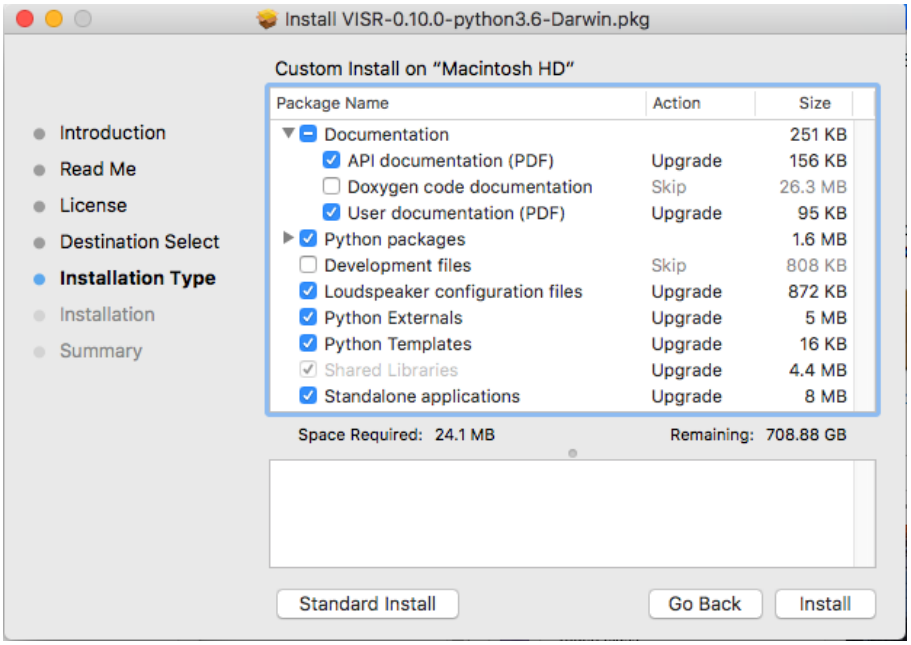

Fig. 3: Component-based installer for Mac OS X.

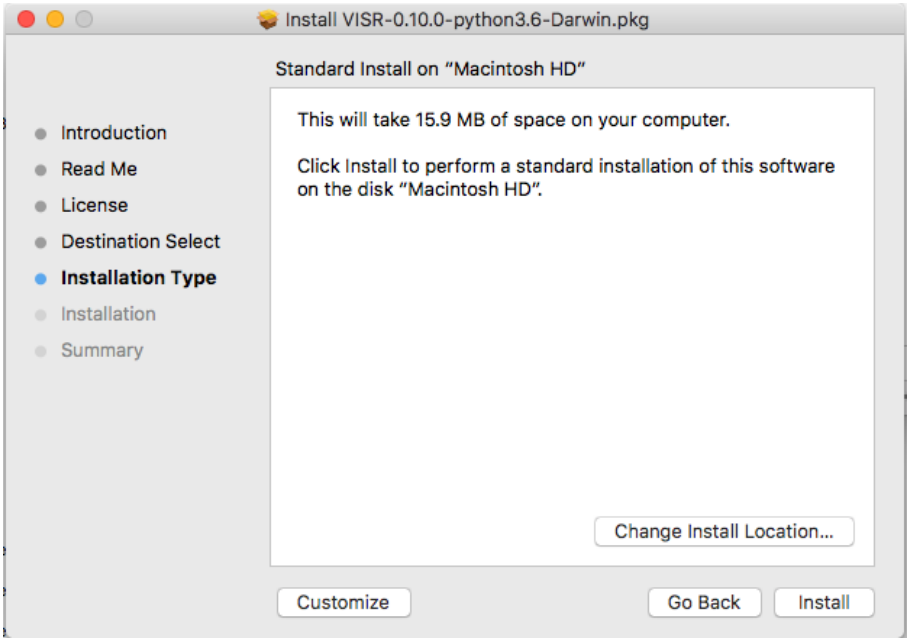

Fig. 4: "Installation type" screen of Mac OS X installer. Use "Customize" to get to the component selection.

5.3 Installation components

With the dialog-based, component-enabled installers, parts of the framework can be chosen depending on the intended use of the framework.

Shared Libraries The core VISR libraries. This component is mandatory and cannot be unselected.

Standalone applications. Renderers and small tools to be run as command-line applications.

Python externals Python modules that give access to the functionality of the framework from Python. Also needed to run applications that use Python internally (e.g., the binaural synthesis toolkit or metadapter-enabled rendering).

Python Packages VISR extensions implemented in Python. This group of components requires the component "Python externals".

Development files Header files and CMake build support - Needed to extend the VISR with components using C++ or use the framework in external C++ applications.

Loudspeaker configurations A set of standard loudspeaker configuration files and additional example files from actual locations.

Python templates A set of commented template files for different types of VISR components.

Documentation User and code reference documentation as PDF documents. The Doxygen code documentation covering the complete source code can be optionally selected. However, the latter documentation is deprecated and will be contained in the code reference documentation in the future.

5.4 Setting up Python

As explained in section *Python integration*, the Python integration is an optional, albeit central, part of the VISR framework that enables a number of its functionalities, for example:

- Using the framework interactively from a Python interpreter.
- Using application that use Python internally, for instance the Binaural Synthesis Toolkit or metadata adaptation processes using the metadapter.
- Creating new signal flows or algorithms in Python.

To use these functionalities, a Python 3 distribution must be installed on the computer, and some configuration steps are required.

5.4.1 Python distribution

Depending on the system, we suggest different Python distributions:

Linux

Use the system-provided Python3 installation.

To install, use the package manager of your distribution, e.g.,

```
sudo apt install python3
```

Windows and Mac OS X

We recommend Anaconda. Please make sure you install the Python3 / 64-Bit variant.

Note: Some Mac OS variants (for example 10.12) come with a pre-installed Python 3 variant in `/Library/Frameworks/Python.framework`. In this case, care must be taken that it does not interferes with the chosen Python distribution. In particular, the `PYTHONHOME` environment variable must be set correctly.

5.4.2 Configuration

Two environment variables must be set to ensure the working of the VISR Python subsystem.

- `PYTHONPATH` This variable is used to add the directory containing the VISR python modules to the system path. To this end, the `python/` subdirectory of the installation folder must be added to `PYTHONPATH`.

  Note that other ways exist to add to the system path, for example

  ```
  import sys
  sys.path.append( '<visr_installation_dir>/python' )
  ```

  However, we recommend setting `PYTHONPATH` and assume this in the examples throughout this document.

PYTHONHOME This variable is needed to locate the files and libraries of the Python distribution. This is especially important if there are more than one distributions on the system, most often on Mac OS X. Strictly speaking, this variable is required only if VISR Python code is executed from a C++ application, for instance some DAW plugins, `python_runner` standalone application (section ??), or the `visr_renderer` with metadata processing enabled. (see section *VISR object-based loudspeaker renderer*).

This variable has to be set to the root directory of the Python distribution, i.e., one level of hierarchy above the `bin/` folder conatining the Python interpreter. Depending on the platform and the distribution, the correct value might be:

Windows with Anaconda `C:\ProgramData\Anaconda3`

Mac OS X with Anaconda *$HOME/anaconda3/*

Linux `/usr`

It is necessary to check whether these settings match with your directory layout.

If the Python distribution provides a `python-config` or `python3-config` binary, the command

```
python-config --prefix
```

or

```
python3-config --prefix
```

can be used to retrieve the required value for `PYTHONHOME` On Linux, setting `PYTHONHOME` is not necessary in most cases, because there is only the system-provided Python installation available.

OPENBLAS_NUM_THREADS It is advisable, in many cases, to set the value of this environment variable to 1. It controls how `numpy` numerical algebra functions are distributed to multiple CPU cores. `numpy` is used by the VISR Python integration as well as in many Python-based VISR components performing mathematical or DSP operations. For the matrix/vector sizes typically encountered in our code, the overhead for distributing the work over multiple cores typically exceeds the potential gains. Multithreading is disabled by setting the maximum number of cores (or threads) to 1:

```
OPENBLAS_NUM_THREADS = 1
```

This setting is optional. However, if you encounter excessive processor loads, for example a constant 100% load in the real-time thread, this setting can help to resolve the problem.

Depending on the operating system, these variables can be set as follows:

Linux Append the lines

```
export PYTHONPATH=$PYTHONPATH:/usr/share/visr/python
export OPENBLAS_NUM_THREADS=1
```

to `$HOME/.profile`.

Windows Add `PYTHONPATH` entries either as a user or system variable as described in *Windows* section. The corrects setting are (assuming the default installation directory and the Anaconda distribution):

```
PYTHONPATH=c:\Program Files\VISR-X.X.X\python
PYTHONHOME=c:\ProgramData\Anaconda3
OPENBLAS_NUM_THREADS=1
```

Note that if there is already a `PYTHONPATH` variable, the recommended value should be appended, using a semicolon as a separator.

Mac OS X In order to set the environment variables system-wide, without requiring that the applications in question is started from a shell, (e.g., a command-line terminal), we recommend a custom `launchd` property list file, as detailed, e.g., in this StackExchange thread.

Note: For convenience, the installers create a pre-configured `VISR-X.X.X.plist` file in the `etc` subdirectory of the installation directory (e.g., `/Applications/VISR-X.X.X/etc/VISR-X.X.X.plist`). This file can be either loaded directly or copied to the `LaunchAgents/` directory first. Please check the values in this file first and adjust them accordingly.

The `VISR-X.X.X.plist` will have this contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>Label</key>
 <string>my.startup</string>
 <key>ProgramArguments</key>
 <array>
   <string>sh</string>
   <string>-c</string>
   <string>
   launchctl setenv PYTHONPATH /Applications/VISR-X.X.X/python
   launchctl setenv OPENBLAS_NUM_THREADS 1
   launchctl setenv PYTHONHOME <BASE_DIRECTORY_OF_PYTHON_INSTALLATION>
   </string>
 </array>
 <key>RunAtLoad</key>
 <true/>
</dict>
</plist>
```

By convention, these files are stored in `/Users/<loginname>/Library/LaunchAgents/`. To activate the settings, call

```
launchctl load <path-to-file>/VISR-X.X.X.plist
```

To take effect, all applications using these settings (e.g., terminals, Python interpreters, DAWs) must be quit and reopened.

These settings are preserved if the machine is restarted. To deactivate them, the property list file must be unloaded:

```
launchctl unload <path-to-file>/VISR-X.X.X.plist
```

If you made changes to the settings, you have to perform the `unload` command followed by a `load`.

5.5 Verifying the installation

We suggest some basic tests to verify that the VISR framework has been correctly installed and configured.

5.5.1 Testing a standalone application

This test is to ensure that that the installation is successful, and that the VISR shared libraries can be located and are compatible with the system. When using the component-enabled installers, the component `Standalone applications` must have been selected in order to perform this check.

In a terminal (Linux shell, Mac OS Terminal application, Windows command line `cmd`), execute this command:

```
<visr-installation-dir>/bin/matrix_convolver --version
```

For the different platforms, the full commands are (assuming the default installation directory) Windows

```
  "c:\Program Files\VISR-X.X.X\bin\matrix_convolver.exe" --version

Note that the quotes are necessary to cope with the space in the path.
```

Mac OS X

```
/Applications/VISR-X.X.X/bin/matrix_convolver --version
```

Linux

```
/usr/bin/matrix_convolver --version
```

If you added the `bin/` directory as described above, calling

```
matrix_convolver --version
```

is sufficient.

In any case, the call should generate a statement like

```
VISR Matrix convolver utility  0.10.0
```

If there is an error message about a missing shared library (or DLL), you should consult the respective section about installation. In particular this applies Windows, where the `PATH` variable needs to be set accordingly.

Testing the interactive Python integration

This test ensures that the VISR framework can be used interactively from Python interpreters.

First start a Python 3 interpreter (for example `python` or `ipython`). Depending on the system, the binaries might be called `python3` or `ipython3`, respectively It must be the interpreter of the Python distribution you intend to use (e.g., Anaconda).

In the interpreter, try to import the `visr` modules

```
import visr
```

This command should return without an error message. In this case, you can check whether the module is loaded from the correct location:

```
getattr( visr, '__file__' )
```

The directory of the resulting file path should be `<visr-installation-dir>/python`. For example, on Windows this returns `C:\Program Files\VISR 0.10.0\\python\\visr.pyd`.

5.6 Source Code

Alternatively, the VISR framework can be installed and build from source code. It is hosted at the GitLab repository https://gitlab.eps.surrey.ac.uk:s3a/VISR.git

To retrieve the source code, clone the repository with

```
git clone https://gitlab.eps.surrey.ac.uk:s3a/VISR.git
```

Setting up a build environment, including the required software tools, and compiling the source code is detailed in the VISR API documentation.

5.7 Support and help

Sopport for installing and using the VISR is available through several ways.

First, you should check the FAQ section of the website (TODO: Insert link here)

Second, the mailing list (insert link to the registration page of the 3a-software list here).

Third, problems and supected bugs can be reported on (insert link to issues page of GitLab repository / later GitHub repo).

CHAPTER

SIX

VISR PRINCIPLES

CHAPTER

SEVEN

USING VISR

7.1 Using VISR standalone renderers

7.1.1 Using standalone applications

The VISR framework provides a number of standalone real-time rendering applications for some of its audio processing functionality.

If a component-aware installer is used (see Section *Installation components*), then the component "Standalone applications" has to be selected during installation.

The standalone applications are started as command line applications, and configured through a number of command line options or a configuration file.

Common options

All standalone applications provided with the VISR provide a common set of command line options:

–version or -v Returns a short description of the tool and its version information.

–help or -h Returns a list of supported command line options with brief descriptions.

–option-file <filename> or @<filename> Pass a configuration file containing a set of command line options to the applications. This options allows to store and share complex sets of command line options, and to overcome potential command line length limitations.

A typical option file has the format

```
-i 2
-o 2
-f 48000
-c "/usr/share/visr/config/generic/stereo.xml"
```

where,by convention, one option is stored per line.

–sampling-frequency or -f The sampling frequency to be used for rendering, as an integer value in Hz. Typically optional. If not given, a default value (e.g., 48000 Hz) will be used.

–period or -p The period, or blocksize, or buffersize to be used by the audio interface.

In most cases, the period should be a power of 2, e.g., 64, 128, 256, 512, ..., 4096. Lower values mean lower audio latency, but typically higher system load and higher susceptibility to audio underruns.

Typically an optional argument. If not given, a default value (e.g., 1024) is used.

–audio-backend or -D Specify the audio interface library to be used.

This option is mandatory.

The audio interfaces depend on the operating system and the configuration of the user's system. The most common options are **"PortAudio"** (all platforms) and **"Jack"** (Linux and Mac OS X). Note that additional

libraries (or backends) can be available for a specific platform, and new backends might be added in the future.

-audio-ifc-options A string to provide additional options to the audio interface.

This is an optional argument, and its content is interface-specific.

By convention, the existing audio interfaces expect JSON (JavaScript Object Notation) strings for the backend-configuration.

To pass JSON strings, the whole string should be enclosed in single or double quotes, and the quotes required by JSON must be escaped with a backslash. For example, the option might be used in this way:

```
visr_renderer ... -audio-ifc-options='{ \"hostapi\": \"WASAPI\" }'
```

Section *Interface-specific audio options* below explains the options for the currently supported audio interfaces.

-audio-ifc-option-file Provide a interface-specific option string within a file.

This can be used to avoid re-specifying complex options strings, to author them in a structured way, and to store and share them.

In addition, it avoids the quoting and escaping tricks needed on the command line. For example, the option shown above could be specified in a file **portaudio_options.cfg** as

```
{
  "hostapi": "WASAPI"
}
```

and passed as

```
visr_renderer ... -audio-ifc-option-file=portaudio_options.cfg
```

Note: The options **–audio-ifc-options** and **–audio-ifc-option-file** are mutually exclusive, that means other none or one of them can be provided.

VISR object-based loudspeaker renderer

These renderers facilitate object-based rendering to arbitrary loudspeaker setups. They use the VISR audio object model and the corresponding JSON format described in Section *The VISR object model*.

Note that there are two binaries for loudspeaker rendering: **visr_renderer** and **baseline_renderer**. The provision of these separate binaries has technical reasons - mainly their dependency on a compatible and configured Python installation, as explained below.

The two binaries provided are:

visr_renderer This is the full object-based renderer, including a powerful metadata adaptation engine for intelligent object-based rendering - the Metadapter - implemented in Python. This metadapter is integrated into the rendering binary as an optional part, and is used if the option **–metadapter-config** is specified. The binary itself, however, needs a Python istallation to start at all, irrespective whether this option is set.

baseline_renderer This is the legacy object-based loudspeaker renderer. At the time being, it provides the same functionality as the **visr_renderer**, but without the optional integrated metadapter component. In this way, the binary is independent of a Python distribution on the user's computer.

In general, we recommend to use **visr_renderer** if possible, and to use **baseline_renderer** on systems where the Python features of the VISR framework are not available.

The command line arguments supported by the **visr_renderer** application are:

```
$> visr_renderer.exe --help
-h [ --help ]                    Show help and usage information.
-v [ --version ]                 Display version information.
--option-file arg                Load options from a file. Can also be used
                                 with syntax "@<filename>".
-D [ --audio-backend ] arg       The audio backend.
-f [ --sampling-frequency ] arg Sampling frequency [Hz]
-p [ --period ] arg              Period (blocklength) [Number of samples per
                                 audio block]
-c [ --array-config ] arg        Loudspeaker array configuration file
-i [ --input-channels ] arg      Number of input channels for audio object
                                 signal
-o [ --output-channels ] arg     Number of audio output channels
-e [ --object-eq-sections ] arg Number of eq (biquad) section processed for
                                 each object signal.
--reverb-config arg              JSON string to configure the object-based
                                 reverberation part, empty string (default) to
                                 disable reverb.
--tracking arg                   Enable adaptation of the panning using visual
                                 tracking. Accepts the position of the tracker
                                 in JSON format"{ "port": <UDP port number>,
                                 "position": {"x": <x in m>, "y": <y im m>,
                                 "z": <z in m> }, "rotation": { "rotX": rX,
                                 "rotY": rY, "rotZ": rZ } }" .
-r [ --scene-port ] arg          UDP port for receiving object metadata
-m [ --metadapter-config ] arg   Metadapter configuration file. Requires a
                                 build with Python support. If empty, no
                                 metadata adaptation is performed.
--low-frequency-panning          Activates frequency-dependent panning gains
                                 and normalisation
--audio-ifc-options arg          Audio interface optional configuration
--audio-ifc-option-file arg      Audio interface optional configuration file
```

The arguments for the **baseline_renderer** application are identical, except that the `--metadapter-config` option is not supported as explained above.

`--audio-backend` or `-D` The audio interface library to be used. See section *Common options*.

`--audio-ifc-options`: Audio-interface specific options, section *Common options*.

`--audio-ifc-option-file`: Audio-interface specific options, section *Common options*.

`--sampling-frequency` or `-f`: Sampling frequency in Hz. Default: 48000 Hz. See section *Common options*.

`--period` or `-p`: The number of samples processed in one iteration of the renderer. Should be a power of 2 (64,128,...,4096,...) . Default: 1024 samples. See section *Common options*.

`--array-config` or `-c`: File path to the loudspeaker configuration file. Path might be relative to the current working directory. Mandatory argument. The XML file format is described in Section *The loudspeaker configuration format*.

`--input-channels` or `-i`: The number of audio input channels. This corresponds to the number of single-waveform objects the renderer will process. Mandatory argument. A (case-insensitive) file extension of `.xml` triggers the use of the XML format for parsing.

`--output-channels` or `-o`: The number of output channels the renderer will put write to. If not given, the number of output channels is determined from the largest logical channel number in the array configuration.

`--object-eq-sections`: The number of EQs (biquad sections) that can be specified for each object audio signal.

Default value: 0, which deactivate EQ filtering for objects.

`--low-frequency-panning`: Switches the loudspeaker panning between standard VBAP and a dual-frequency approach with separate low- and high-frequency panning rules.

Admissible values are `true` and `false`. The default value is `false`, corresponding to the standard VBAP algorithm.

`--reverb-config`: A set of options for the integrated reverberation engine for the RSAO (`PointsourceWithReverb`) object (see section *Object-Based Reverberation*). To be passed as a JSON string. The supported options are:

`numReverbObjects`: The number of RSAO objects that can be rendered simultaneously. These objects may have arbitrary object ids, and they are automatically allocated to the computational resources avalable.

To be provided as a nonnegative integer number The default value is 0, which means that the reverberation rendering is effectively disabled.

`lateReverbFilterLength`: Specify the length of the late reverberation filters, in seconds.

Provided as a floating-point value, in seconds. Default value is zero, which results in the shortest reverb filter length that can be processed by the renderer, typically one sample.

`lateReverbDecorrelationFilters`: Specifies a multichannel WAV file containing a set of decorrelation filters, one per loudspeaker output. The number of channels must be equal or greater than the number of loudspeakers, channels that exceed the number of loudspeakers are not used.

To be provided as a full file path. The default value is empty, which means that zero-valued filters are used, which effectively disables the late reverb.

`discreteReflectionsPerObject`: The maximum number of discrete reflections that can be rendered for a single RSAO object.

Given as a nonnegative integer number. The default value is 0, which means that no discrete reflections are supported.

`maxDiscreteReflectionDelay`: The maximum discrete reflection delay supported. This allows a for tradeoff between the computational resources, i.e., memory required by the renderer and a realistic upper limit for discrete reflection delays.

To be provided as a floating-point number in seconds. Default value is 1.0, i.e., one second.

`lateReverbFilterUpdatesPerPeriod` Optional argument for limiting the number of filter updates in realtime rendering. This is to avoid processing load peaks, which might lead to audio underruns, if multiple RSAO objects are changed simultaneously. The argument specifies the maximum number of objects for whom the late reverb filter is calculated withon one period (audio buffer). If there are more pending changes than thix number, the updates are spread over multiple periods. This is a tradeoff between peak load and the timing accuracy and synchronity of late reverb updates.

Optional value, default value is 1, meaning at most one update per period

An example configuration is:

```
--reverb-config='{ \"numReverbObjects\": 5, \"lateReverbFilterLength\": 4.0,
          \"lateReverbDecorrelationFilters\": "/home/af5u13/tmp/decorr.wav\",
          \"discreteReflectionsPerObject\": 10 }'
```

`--tracking` Activates the listener-tracked VBAP reproduction, which adjust both the VBAP gains as well as the final loudspeaker gains and delays according to the listener position. It takes a non-empty string argument containing a JSON message of the format: `{ "port": <UDP port number>, "position": {"x": <x in m>, "y": <y im m>, "z": <z in m> }, "rotation": { "rotX": rX, "rotY": rY, "rotZ": rZ } }`. The values are defined as follows:

ID	Description	Unit	Default
port	UDP port number	unsigned int	8888
position.x	x position of the tracker	m	2.08
position.y	y position of the tracker	m	0.0
position.z	z position of the tracker	m	0.0
rotation.rotX	rotation the tracker about the x axis, i.e., y-z plane	degree	0.0
rotation.rotY	rotation the tracker about the y axis, i.e., z-x plane	degree	0.0
rotation.rotZ	rotation the tracker about the z axis, i.e., x-y plane	degree	180

Note: The option parsing for `--tracking` not supported yet, default values are used invariably. To activate tracking, you need to specify the `--tracking` option with an arbitrary parameter (even `--tracking=false` would activate the tracking.

`--scene-port` The UDP network port which receives the scene data in the VISR JSON object format.

`--metadapter-config` An optional Metadapter configuration file in XML format, provided as a full path to the file. If specified, the received metadata are passed through a sequence of metadata adaptation steps that are specified in the configuration file. If not given., metadata adaptation is not performed, and objects are directly passed to the audio renderer.

This option is not supported by the **baseline_renderer** application.

The matrix convolver renderer

The matrix convolver renderer is a multiple-input multiple-output convolution engine to be run as a command line application.

It implements uniformly partitioned fast convolution for arbitrary routing points between input and output files.

Basic usage

```
$> matrix_convolver --help
-h [ --help ]                    Show help and usage information.
-v [ --version ]                 Display version information.
--option-file arg                Load options from a file. Can also be used
                                 with syntax "@<filename>".
-D [ --audio-backend ] arg       The audio backend. JACK_NATIVE activates the
                                 native Jack driver insteat of the PortAudio
                                 implementation.
--audio-ifc-options arg          Audio interface optional configuration
--audio-ifc-option-file arg      Audio interface optional configuration file
--list-audio-backends            List the supported audio backends that can be
                                 passed to the the "--audio-backend" ("-D")
                                 option.
--list-fft-libraries             List the supported FFT implementations that
                                 can be selected using the "--fftLibrary"
                                 option.
-f [ --sampling-frequency ] arg Sampling frequency [Hz]
-p [ --period ] arg              Period (block length): The number of samples
                                 per audio block, also the block size of the
                                 partitioned convolution.
-i [ --input-channels ] arg      Number of input channels for audio object
                                 signal.
-o [ --output-channels ] arg     Number of audio output channels.
--filters arg                    Initial impulse responses, specified as
                                 comma-separated list of one or multiple WAV
```

(continues on next page)

(continued from previous page)

```
                                files.
--filter-file-index-offsets arg Index offsets to address the impulses in the
                                provided multichannel filter files. If
                                specified, the number of values must match
                                the number of filter files.
-r [ --routings ] arg           Initial routing entries, expects a JSON array
                                consisting of objects "{"inputs": nn,
                                "outputs":nn, "filters":nn ("gain":XX)
-l [ --max-filter-length ] arg  Maximum length of the impulse responses, in
                                samples. If not given, it defaults to the
                                longest provided filter,
--max-routings arg              Maximum number of filter routings.
--max-filters arg               Maximum number of impulse responses that can
                                be stored.
--fft-library arg               Specify the FFT implementation to be used.
                                Defaults to the default implementation for
                                the platform.
```

Operation

The matrix convolver consists of the following elements:

- A number of **input channels**.
- A set of **FIR filter**, which can be reused multiple times.
- A set of **output channels**.
- A set of **routings**, which defines that a given input is filtered through a specific filter (with an optional gain), and the result is routed to a given output channels. All filtering results that are routed to a given output are summed together.

This interface allows for several different operation modes, for example:

- Multi-channel filtering where each input is filtered with one filter to give produce the same number of output channels.
- Filtering to produce multiple, different copies of the same input signal.
- Filtering multiple signals and adding them together, as, for example, in filter-and-sum beamforming.
- MIMO filtering with complete matrices, where a filter is defined for each input-output combination.
- MIMO filtering with sparse matrices, corresponding to sophisticated routings between inputs and outputs.

Detailed option description

`--help` or `-h`:

`--version` or `-v`: Standard options, described in *Common options*

`--option-file`: Standard options, described in *Common options*

`--audio-backend` or `-D`: Standard options, described in *Common options*

`--audio-ifc-options`: Standard options, described in *Common options*

`--audio-ifc-option-file`: Standard options, described in *Common options*

`--sampling-frequency` or `-f` Standard options, described in *Common options*

`--period` or `-p`: Standard options, described in *Common options*

`--input-channels` or `-i`: The number of input channels. Must not exceed the number of capture channels of the sound card.

`-o` or `--output-channels`: The number of output channels. Must be less or equal than the number of sound card output channels.

`--filters` The filters, specified as a comma-separated list of WAV files. WAV files can be multichannel, in this case, every channel is handled as a separate filter.

All filters are combined into a single array, where each filter is associated to a unique index (starting from zero if not specified otherwise.)

This argument is optional. If not provided, all filters are zero-initialised. Note that if the `filters` argument is not provided, then the option `max-routings` must be provided.

`--filter-file-index-offsets` Specify the start filter index for each WAV file specified by the `--filters` argument. To be provided as a comma-separated list of nonnegative filter entries, one for each file in the `filters` argument. This argument is optional. If not provided, the start index of the first file is 0, and the start offset af all subsequent filter files follows the end index of the previous filter file. This facility can be used to decouple the number of filters in the WAV files from the indexing scheme used to define the routings.

Example:

```
--filters ="filters_2ch.wav, filters_6ch.wav, filters_4ch.wav"
--filter-file-index-offsets="2, 8, 16"
```

Here, three WAV files are provided: `filters_2ch.wav`, `filters_6ch.wav`, and `filters_4ch.wav`, with 2, 6, and 4 channels respectively. The filter offsets "2, 8, 16" mean that the filters of `filters_2ch.wav` will be associated to the indices 2 and 3, that of `filters_6ch.wav` by indices 8-13, and that of `filters_4ch.wav` by the indices 16-19.

Any filters below, between, or above the initialized filter channels (here, indices 0-1, 4-7, 14-15, and >=20) will be zero-initialised.

If the `--filter-file-index-offsets` hadn't been provided in this example, the start offsets for the filter sets from the three files would have been 0,2,8.

`--routings` or `-r` Provide a list of routings points. This is to be specified as a JSON string. A routing defines a filter being applied between a specific input channel and a specific output channels. The JSON representation for a single entry is

```
{ "input": "<i>", "output": "<o>", "filter": "<f>", "gain": "<g>" }
```

Here, `<i>` is the index of the input channel, `<o>` is the channel index of the output, and `<f>` is the index of the filter (see above). All indices are zero-offset. The gain specification `,"gain": <g>` is optional, with `<g>` representing a linear-scale gain value.

A routing list is a JSON array of routing entries, for example

```
[{"input":"0", "output":"0", "filter":"2" },
 {"input":"0", "output":"1", "filter":"1" },
 {"input":"0", "output":"2", "filter":"0" }]
```

A routing entry can define multiple multiple routings using a Matlab-like stride syntax for `<i>`, `<o>`, `<f>`, or several of them. If an index is a stride sequence, then the routing entry is duplicated over all values of the stride sequence. If more than one index in the routing entry are strides, then all of them must have the same length, and each of the duplicated routing entries contains the respective value of the respective stride sequence. For example, the strided routing entry

```
{"input":"3", "output":"0:3:9", "filter":"1" }
```

routes input 3 to the outputs 0, 3, 6, and 9, using the filter indexed by 1 for each routing. In contrast.

```
{"input":"0", "output":"0:2", "filter":"2:-1:0" }
```

is equivalent to the routing list shown above.

```
[{"input":"0", "output":"0", "filter":"2" },
 {"input":"0", "output":"1", "filter":"1" },
 {"input":"0", "output":"2", "filter":"0" }]
```

--max-filter-length or -l: Define the maximum length of the FIR filters. If the `--filters` option is provided, this argument is optional. In this case, admissible filter length is set to the largest length of all specified filter. an error is reported if any specified filter exceeds the admissible length. If `--filters` and `--max-filter-length` are both provided, then an error is generated if the length of any specified filter exceeds the value of `--max-filter-length`.

--max-routings : Define the maximum number of routings. If the `--routings` options is present, this argument is optional, and the maximum number of permissible routings is set to the number of routing entries in the `--routing` argument. If `routings` and `--max-routings` are both specified, the number of entries in `--routings` must not exceed the value of `--max-routings`.

--max-filters: Define the maximum number of filter entries. This parameter is optional if the argument `--filters` is provided. In this case, the maximum filter number is set to the number of filters generated by the `--filters` argument.

Note: If combined with `--filter-file-index-offsets`, this automatically computed number of filters includes any gaps in the generated filter set.

If `--filters` and `max-filters` are both provided, then the number of filter entries created by `--filters` must not exceed the value of `--max-filters`.

--fft-library: Select a FFT implementation from the set of available FFT libraries. The admissible values (strings) can be obtained through the `--list-fft-libraries` option.

Note: The current implementation accepts only a static configuration.

Future versions, however, will provide runtime control through a network command interface.

Some arguments or argument combinations do not make sense at the moment, but will do when combined with runtime control. Examples include the ability to provide empty routings, zero-valued filters, or to specify values for `--max-routings` or `--max-filters` that are larger than the currently set values.

Examples

A channel-wise multichannel convolution can be performed as

```
$> matrix_convolver -i 2 -o 2 -p 512 -D PortAudio -f 48000 --filters="filters.wav"
  -r '[ {\"input\": \"0:1\", \"output\":\"0:1\", \"filter\":\"0:1\"}]'
```

Note: The quoting is necessary when started from the command line.

The following example shows a convolution with binaural room impulse responses, where a 9-loudspeaker multichannel signal is routed to 9x2 BRIRs that are summed to form two ear signals.

```
$> matrix_convolver -i 9 -o 2 --max-filters=18 --max-routings=18
   -r "[{\"input\":\"0:8\", \"output\":\"0\", \"filter\":\"0:2:16\"},
        {\"input\":\"0:8\", \"output\":\"1\", \"filter\":\"1:2:17\"}]"
   --filters="bbcrdlr9ch_brirs.wav"
   -D Jack -f 48000 -p 512
```

Here, the file `bbcrdlr9ch_brirs.wav` contains the 18 BRIRs, with the first nine channels for the left and the remaining channels for the right ear filters.

The python_runner application

This standalone application is an alternative way to run arbitrary VISR components in real-time.

Compared to instantiating the processing from a Python interpreter, this can be easier to control, for example within a script or when running a device in 'headless mode'.

For obvious reasons, this application requires an installed and correctly configured Python distribution, as described in Section *Configuration*.

Usage

The supported options are displayed when started with the `--help` or `-h` option:

```
$> python_runner --help
  -h [ --help ]                         Show help and usage information.
  -v [ --version ]                      Display version information.
  --option-file arg                     Load options from a file. Can also be used
                                        with syntax "@<filename>".
  -D [ --audio-backend ] arg            The audio backend.
  -f [ --sampling-frequency ] arg       Sampling frequency [Hz]
  -p [ --period ] arg                   Period (blocklength) [Number of samples per
                                        audio block]
  -m [ --module-name ] arg              Name of the Python module to be loaded
                                        (without path or extension).
  -c [ --python-class-name ] arg        Name of the Python class (must be a
                                        subclass of visr.Component).
  -n [ --object-name ] arg              Name of the Python class (must be a
                                        subclass of visr.Component).
  -a [ --positional-arguments ] arg Comma-separated list of positional options
                                        passed to the class constructor.
  -k [ --keyword-arguments ] arg        Comma-separated list of named (keyword)
                                        options passed to the class constructor.
  -d [ --module-search-path ] arg       Optional path to search for the Python
                                        module (in addition to the default search
                                        path (sys.path incl. $PYTHONPATH). Provided as␣
↪a
                                        comma-separated list of directories.
  --audio-ifc-options arg               Audio interface optional configuration.
  --audio-ifc-option-file arg           Audio interface optional configuration file.
```

If the processing is correctly started, a message is displayed on the command line:

```
VISR Python signal flow runner. Press "q<Return>" to quit.
```

To terminate the `python_runner`, press the "q" key followed by <Return>.

Note: On Linux and Mac OS X, the standard program termination via <Ctrl-C> does not work at the moment. Instead, this key combination is ignored, and Python exception message is shown if the program is later terminated via "q<Return>". See issue https://gitlab.eps.surrey.ac.uk/s3a/VISR/issues/23 .

Detailed option description

The standard options `--help`, `--version`, `--audio-backend`, `sampling-frequency`, *:code:*–period', *:code:*–audio-ifc-options', and *:code:*–audio-ifc-option-file' are described in Section *Common options*.

The remaining options are:

--module-name or -m: Specify the name of a Python module that contains the VISR component to be executed. That is, use the module name that would need to be imported in an interactive Python session. The module name must be provided without the file extension. It can be specified either with a full file path, or as a pure module name. In the latter case, the directory containing the module must be on the Python module search path or included in the `--module-search-path` option.

The module can be in one of several forms:

- A Python file (normally with extension `.py`) that contains the component class. The module name must be specified without the extension.
- A directory containing a multi-file package.
- Compiled extension modules implemented in C++. Typical file extesnions are `.so` (Linux and Mac OS X) or `.pyc` (Windows). The module name must be specified without the extension.

This is a mandatory argument.

--python-class-name or -c: The name of the Python class to be instantiated, without the leading namespace name. This class must be derived from `visr.Component` and must be defined in the module `module-name`.

Note: At the moment, only classes in the top-level namespace are supported. That is, classes of the form `moduleName.submodule.className` cannot be used.

This argument is mandatory.

--object-name or -n: Set a name for the top-level component. This name is used, for example, in error messages and warnings emitted from the component.

This argument is optional. If not provided, a default name is used.

-a --positional-arguments: Provide a sequence of parameters to the component's constructor as positional arguments.

The fixed first three arguments to a component constructor, i.e., `context`, `name` and `parent`, do not need top be specified. That means the first value of the sequence is passed to the fourth argument, the second value to the fifth argument, and so on.

The parameters are passed as a Python tuple. See, e.g., the Python documentation on tuples. Following these conventions, the arguments can be specified as follows:

- A comma-separated list of values, for example

```
-a "3, 2.7,'foobar'"
```

Note that the enclosing double quotes are required to separate the argument to `-a` from other options on the command line. They are strictly necessary only if the parameter sequence contains spaces, but we recommend to use double quotes for consistency.

If the parameter sequence consists of a single value, a training comma is required. That is, a single positional argument is specified as

```
-a "3,"
```

If two or more arguments are provided, the trailing comma is optional.

- A comma-separated list of values, enclosed in parentheses. Apart from the additional parentheses, the syntax is identical to the comma-separated lists above. That is, the argument list above would be specified as

```
-a "(3, 2.7,'foobar' )"
```

As above, single arguments require a trailing comma.

```
-a "(3,)"
```

- A tuple constructed using the `tuple()` keyword, that is

```
-a "tuple(3, 2.7,'foobar' )"
```

and in the single-parameter case

```
-a "tuple(3)"
```

That is, no trailing comma is required in this case.

The `--positional-arguments` option is optional. If it is not provided, no positional arguments are passed to the component's constructor.

`--keyword-arguments` or `-k`: A set of keyword arguments to be passed to the component's constructor. To be provided as a Python dictionary, for example:

```
-k "{ 'argument1': value1, 'argument2': value2, ..., 'argumentN': valueN }"
```

Hint: As in case of positional arguments, we suggest to enclose the complete argument in double quotes. When following this convention, single quotes can be used for the keywords as `'argument1'` and string parameters without the need for escaping quotes.

Following Python conventions, keyword arguments must not be provided for arguments already handled by the `--positional-arguments` option. Likewise, keyword arguments must not be provided for the fixed first three constructor arguments of a component: `context`, `name` and `parent`.

This argument is optional; no keyword arguments are passed to the component if it is not given.

`--module-search-path` or `-d`: Specifies additional search paths for Python modules.

To be specified as a comma-separeted list of directory path.

These search paths can be used to locate the module containing the component to be run, unless a directory path is passed to the `--module-name` option. In addition, the search paths are evaluated to locate transitive dependencies of the module to be loaded. For example, the path to VISR Python externals can be specidied in this way, thus avoiding the use of the `PYTHONPATH` environment variable, as described in section *Configuration*. The additional search paths are added to the Python search path `sys.path` before the main module specified by the `-m` option is loaded.

This argument is optional, no additional search paths are added if the option is not provided.

Examples

In this example we use a simple Python-based VISR component `PythonAdder`.

```
class PythonAdder( visr.AtomicComponent ):
  """ General-purpose add block for an arbitrary number of inputs"""
  def __init__( self, context, name, parent, numInputs, width ):
  ...
```

that implements generic addition with `numInputs` signals to be added with `width` signals each. Here, the component class `PythonAdder` is contained in a source file `pythonAtoms.py`.

The `python_runner` can be invoked using positional arguments through

```
$> python_runner -D PortAudio -f 48000 -p 512
    -m $HOME/VISR/src/python/scripts/pythonAtoms -c PythonAdder -a "3,2"
```

which creates a `PythonAdder` component with three inputs and a width of two.

The same component is constructed with the keyword argument option as

```
$> python_runner -D PortAudio -f 48000 -p 512
    -m $HOME/VISR/src/python/scripts/pythonAtoms -c PythonAdder -k "{'width':2,
↪'numInputs':3}"
```

Positional and keyword arguments can also be mixed, as long as the corresponding Python rules are observed:

```
$> python_runner -D PortAudio -f 48000 -p 512
    -m $HOME/VISR/src/python/scripts/pythonAtoms -c PythonAdder -a "3," -k "{'width
↪':2}"
```

Note the trailing comma for the positional option.

So far, the examples specified the path to the module explicitly. If this path (`$HOME/VISR/src/python/scripts` in the example) is contained in the default Python search path, i.e., `sys.path`, then the pure module name suffices

```
$> python_runner -D PortAudio -f 48000 -p 512
    -m pythonAtoms -c PythonAdder -a "3," -k "{'width':2}"
```

Another way to locate the module is to provide the path through the `module-search-path` option.

```
$> python_runner -D PortAudio -f 48000 -p 512
    -m pythonAtoms -c PythonAdder -a "3," -k "{'width':2}"
    --module-search-path $HOME/VISR/src/python/scripts
```

Finally, the option `--module-search-path` can also be used to locate modules needed by the main module. For example, the path to the core VISR modules can be specified in this way, thus eradicating the need to add them to the default Python search path, for example by adding them to the `PYTHONPATH` variable.

```
$> python_runner -D PortAudio -f 48000 -p 512
    -m pythonAtoms -c PythonAdder -a "3," -k "{'width':2}"
    --module-search-path
    $HOME/VISR/src/python/scripts,/usr/share/visr/python
```

Interface-specific audio options

This section described the audio-interface-specific options that can be passed through the `--audio-ifc-options` or `--audio-ifc-option-file` arguments.

PortAudio interface

The interface-specific options for the PortAudio interface are to be provided as a JSON file, for example:

```
{
 "sampleformat": "...",
 "interleaved": "...",
 "hostapi" : "..."
}
```

Note: When used on the command line using the `--audio-ifc-options` argument, apply the quotation and escaping as described in Section *Common options*.

The following options are supported for the PortAudio interface:

sampleformat Specifies the PortAudio sample format. Possible values are:

- `signedInt8Bit`
- `unsignedInt8Bit`
- `signedInt16Bit`
- `unsignedInt16Bit`
- `signedInt24Bit`
- `unsignedInt24Bit`
- `signedInt32Bit`
- `unsignedInt32Bit`
- `float32Bit` .

interleaved: Enable/disable interleaved mode, possible values are `true, false`.

hostapi: Used to specify PortAudio backend audio interface. Possible values are:

- `default`: This activates the default backend
- `WASAPI` : Supported OS: Windows.
- `ASIO` : Supported OS: Windows.
- `WDMKS`: Supported OS: Windows.
- `DirectSound` : Supported OS: Windows.
- `CoreAudio` : Supported OS: MacOs.
- `ALSA` : Supported OS: Linux.
- `JACK` : Supported OSs: MacOs, Linux.

PortAudio aupports a number of other APIs. However, they are outdated or refer to obsolete platforms and therefore should not be used: - `SoundManager` (MacOs) - `OSS` (Linux) - `AL` - `BeOS` - `AudioScienceHPI` (Linux)

This configuration is an example of usage of PortAudio, with Jack audio interface as backend.

```
{
  "sampleformat": "float32Bit",
  "interleaved": "false",
  "hostapi" : "JACK"
}
```

Jack audio interface

The following options can be provided when using Jack as our top level component's Audio Interface:

`clientname`: Jack Client name for our top level component.

`servername`: Jack Server name. If not provided, the default Jack server is used.

`autoconnect`: Globally enable/disable the automatic connection of ports. Admissible values are `true` and `false`. This setting can be overridden specifically for capture and playback ports in the port configuration section described below.

`portconfig`: Subset of options regarding the configuration and connection of Jack Ports, see following section.

Port Configuration

The port configuration section allows to individually set properties for the capture, i.e., input, and the playback, i.e., output, ports of an application.

`capture`: Specifies that the following options regard the top level component's capture ports only

- `autoconnect` : Enable/disable auto connection to an external jack client's input ports, possible values are `true, false`
- `port`: Jack ports specification
  - `basename`: Common name for all top level component's capture ports
  - `indices`: list of port numbers to append to top level component's capture port name. It is possible to use Matlab's colon operator to express a list of numbers in a compact fashion (es."0:4" means appending numbers 0 to 3 to port names)
  - **`externalport`: Specification of an external jack client to connect to if `autoconnect` is enabled.**
    - `client`: Name of an external jack client to use as input for our top level component (es. "system")
    - `portname`: Common name for all external jack client input ports
    - `indices`: List of port numbers that together with :code:' portname' describe existing external jack client input ports. It is possible to use Matlab's colon operator to express a list of numbers.

`playback`: Specifies that the following options regard the top level component's playback ports only.

- `autoconnect` : Enable/disable auto connection to an external jack client's output ports, possible values are `true, false`
- **`port`: Jack ports specification**
  - `basename`: Common name for all top level component's playback ports
  - `indices`: list of port numbers to append to top level component's playback port name. It is possible to use Matlab's colon operator to express a list of numbers in a compact fashion (es."0:4" means appending numbers 0 to 4 to port names)
  - **`externalport`: Specification of an external jack client to connect to if `autoconnect` is enabled.**
    - `client`: Name of an external jack client to use as output for our top level component (es. "system")
    - `portname`: Common name for all external jack client output ports
    - `indices`: List of port numbers that together with :code:' portname' describe existing external jack client output ports. It is possible to use Matlab's colon operator to express a list of numbers.

Simple Example

This configuration example shows how to auto-connect the Jack input and output ports of an application to the default jack client (`system`), specifying which range of ports to connect.

```
{
  "clientname": "BaseRenderer",
  "autoconnect" : "true",
  "portconfig":
  {
    "capture":
```

(continues on next page)

(continued from previous page)

```
    {
      "port":
      [{ "externalport" : {"indices": "1:4"} }]
    },
    "playback":
    {
      "port":
      [{ "externalport" : {"indices": "5:8"} }]
    }
  }
}
```

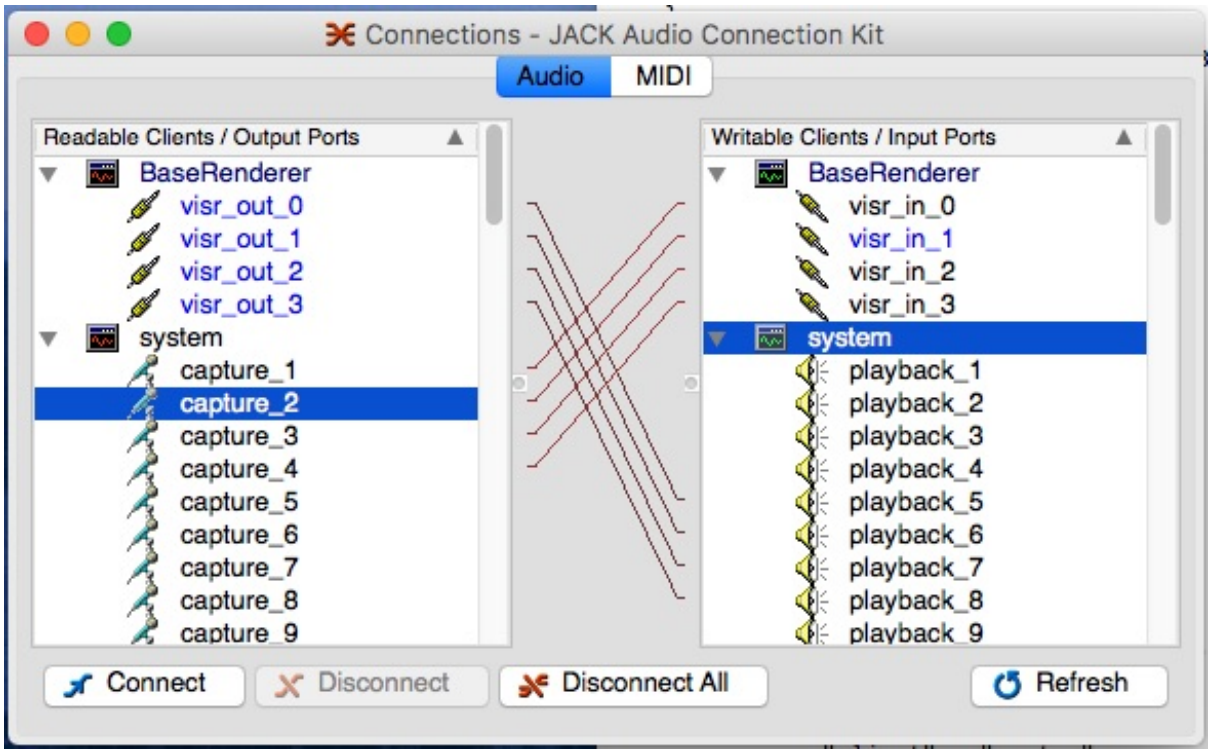

Fig. 1: Jack audio complex configuration example.

Complex Example

Follow a more complex example where auto-connection of ports is performed specifying different jack clients and the ranges of ports to be connected are described both for the top level component and for external clients.

```
{
  "clientname": "VisrRenderer",
  "servername": "",
  "autoconnect" : "true",
  "portconfig":
  {
    "capture":
    {
      "autoconnect" : "true",
      "port":
      [
        {
         "basename" : "Baseinput_" ,
```

(continues on next page)

(continued from previous page)

```
          "indices": "0:1",
          "externalport" :
           {
             "client" : "REAPER",
             "portname": "out",
             "indices": "1:2"
           }
         },
         {
          "basename" : "Baseinput_" ,
          "indices": "2:3",
          "externalport" :
           {
             "indices": "4:5"
           }
          }
       ]
     },
     "playback":
     {
       "autoconnect" : "true",
       "port":
       [{
         "basename" : "Baseoutput_" ,
         "indices": "0:1",
         "externalport" :
          {
           "client" : "system",
           "portname": "playback_",
           "indices": "4:5"
          }
        }]
     }
   }
}
```

7.2 Using VISR with Python

7.3 Using VISR audio workstation plugins

7.4 Using Max/MSP externals

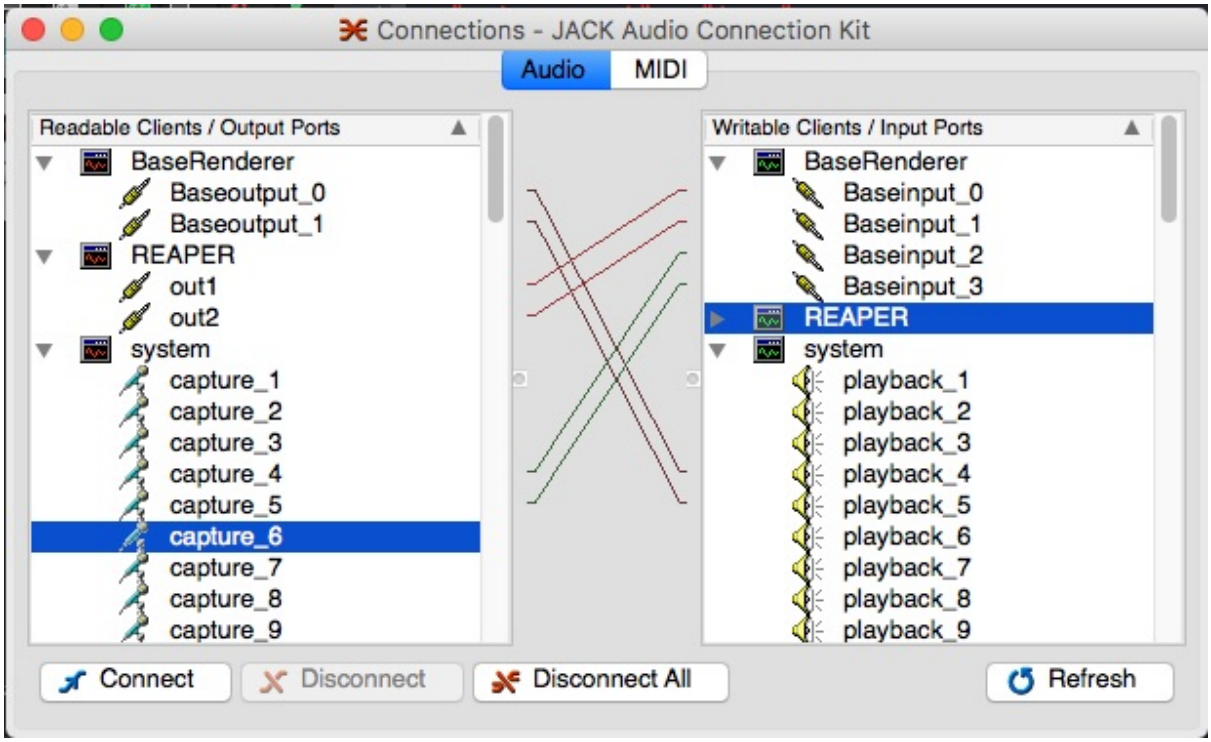

Fig. 2: Jack audio complex configuration example.

CHAPTER

EIGHT

EXTENDING VISR

In this part we describe how to use the VISR framework to implement new functionality, i.e., functionality that is not contained in the existing components or standalone renderers. This part is basically an extended version of the tutorial presented in

8.1 Creating signal flows from existing components in Python

8.2 Writing atomic functionality in Python

8.3 Implementing atomic components in C++

8.4 Creating composite components in C++

CHAPTER

NINE

OBJECT-BASED AUDIO WITH VISR

9.1 Overview

Although the VISR framework is deliberately application-agnostic, it is well-suited for working with spatial and object-based audio.

9.1.1 The VISR object model

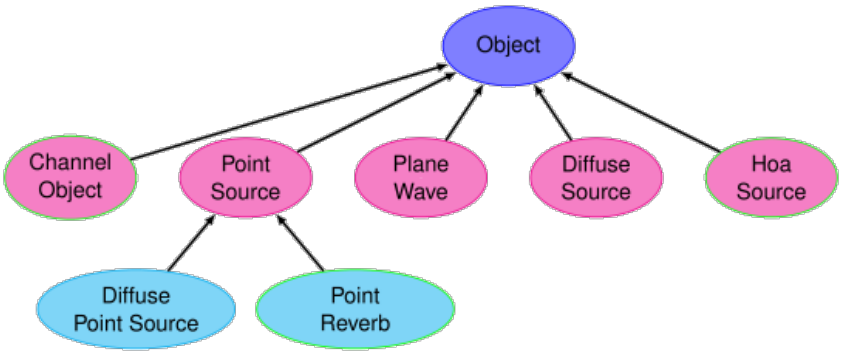

Fig. 1: Object types and hierarchy.

The VISR object model supports a hierarchical and extensible set of object types. These types and their relations are shown above.

JSON representation

For transmission, object vectors are encoded as JSON messages.

A scene vector (or a part thereof) has the format

```
"objects": [ {<object 0>}, {<object 1>}, ... , {<object n>}]
```

where `<object k>` stands for the encoding of a single object. The objects can be arranged in arbitary order as long as the object ids are unique. Moreover, the object vector can be split into arbitrary subsets and be transmitted as individual `"objects"` messages.

Encoding of the individual object types

Coordinate system

Depending on the object type, either Cartesian and spherical coordinates are use. The coordinate axes follow, e.g., the ITU-R BS.2051 conventions.

For Cartesian coordinates, this means:

- x axis points to the front
- y axis points to the left.
- z axis points up.

Coordinates are measured in meters.

Likewise, for spherical coordinates:

- The azimuth angle is measured counterclockwise from the x axis (front).
- The elevation angle is measure up (positive values) or down (negative values) from the horizontal plane.

Coordinates are represented in the JSON format in degree (this does not necessarily hold for the internal representation in the renderer).

Object

`Object` is the base type of all objects. Therefore, the attributes are common to all objects. The following attributes are supported:

`"id"` The object id, a nonnegative integer that must be unique withing the object vector (mandatory attribute).

`"group"` The group id, a nonnegative integer (mandatory attribute). Not used in the core renderer, but potentially in the metadata adaptation process.

`"channels"` A list of audio channel indices referencing the audio signals associated with this object. The list is formed as a string consisting of comma-separated unsigned integers enclosed in quotation marks, e.g., "0,3, 5, 7 " with arbitrary amounts of whitespace in between. The format also allows Matlab-style ranges for any part of the list. For instance, "0, 2 : 2 : 8, 10" is equivalent to "0,2,4,6,8,10". This is a mandatory argument. The reuired number of channels is typically determined by the object type and its parameters. For instance, point source objects are invariably single-channel, while the number of required channels of a HOA object depends on the Ambisonics order specified by the "order" of this object.

`"level"` The level of the audio object in linear scale as a floating-point number. Note that this value does not necessarily denote the loudness of the reproduced object, since the latter also depends on the level of the audio signal(s). (Mandatory argument).

`"priority"` The priority of the object given as an unsigned integer (mandatory argument). Lower numbers represent higher priority, with "0" being the highest prority. Not currently used in the core renderer, but potentially (and moe appropriately) in the metadata processing.

`"eq"` An array of parametric EQ parameters to be applied to all audio signals for this object. This is an optional attribute, if not present, a 'flat', i.e., unity-gain equalisation curve is applied. The attribute has the format

```
"eq": [{<eq 0>}, {<eq 1},...{<eq n-1>}]
```

: The number of admissible EQ sections is renderer-dependent. Providing more EQ parameters for a single object than supported by the renderer might result in an error message and termination of the renderer. If less EQ parameters are sent than supported by the renderer, the remaining EQ sections are padded with 'flat' characteristics. The individual EQ section have the form

```
{ "type": "<type>", "f": (center/cutoff frequency), "q": (quality) [, "gain": (dB)␣
↪}
```

with the following attributes:

`"type"` A type string chosen from the following values: `"lowpass"`, `"highpass"`, `"bandpass"`, `"bandstop"`, `"peak"`, `"lowshelf"`, `"highshelf"`, `"allpass"`.

`"f"` Centre/cutoff frequency in Hz (depending on the filter type).

`"q"` Dimensionless Q (quality) parameter.

"gain" Optional gain parameter (in dB). If not provided, the default value of 0 dB is used. Only used by the filter types "peak", "lowshelf", and "highshelf". The filter characteristics follow the Audio EQ Cookbook formulas.

PointSource

Point sources are invariable single-channel objects, that is the "channels" attribute of the base type Object must contain a single channel index. The type string is "point".

The point source coordinates sre specified in the "position", which is an object holding either Cartesian coordinates "x", "y", and "z" or spherical coordinates "az", "el", "radius"

Example

```
{ "id": "5", "channels": "2", "type": "point", "group": "2", "priority": "0",
↪"level": "0.350",
  "position": {"x": "3.0", "y": "-0.5", "z": "0.25" } }
```

or, using polar coordinates,

```
{ "id": "5", "channels": "2", "type": "point", "group": "2", "priority": "0",
↪"level": "0.350",
  "position": {"az": "30", "el": "15.0", "radius": "1.25" } }
```

PlaneWave

Plane waves differ from point sources that they do not exhibit distance-dependent attenuation and do not provide parallalax effects for moving listener positions. Because the main reproduction method in the VISR renderer at the moment is VBAP, plane waves are handled identically to point sources. This might change for alternative reproduction methods, including listener position adaptive VBAP.

Plane waves use the type "plane" and are single-channel objects.

The plane wave representation uses an object "direction" containing the attributes "az" and "el" to describe azimuth and elevation of the direction of the impinging source. The third parameter "refDist" (reference distance) encodes the relative timing of the object's audio signal: A value of 0 means that a sound event at signal time 0 is perceived at the central listener at time 0.

Example

```
{"id": 5, "channels": 5, "type": "plane", "group": 0, "priority": 0, "level": 1.
↪000000, "direction": {"az": 30.0, "el": 45.0, "refdist": 12.00 } }
```

PointSourceDiffuse

Point source with diffuseness are derived from PointSource and therefore support all attributes of the latter. In addition they define the attribute "diffuseness" that is a floating-point supposed to be in the range between 0.0 and 1.0 and describes the amount of diffuse energy relative to the point source radiation.

They are single-channel and use the type string "pointdiffuse".

Example

```
{"id": "5", "channels": "5", "type": "pointdiffuse", "group": "0", "priority": "0",
 "level": "1.0", "diffuseness": "0.35", "position": {"x": "3.0", "y": "-0.5", "z":
↪"0.25" } }
```

DiffuseSource

This source type describes a surrounding objects reproducing decorrelated signals obtained from the single object audio signal.

This object does not introduce any other attributes apart from those inherited from the base class Object. The type string is `"diffuse"`.

Example

```
{"id": 3, "channels": 3, "type": "diffuse", "group": 0, "priority": 0, "level": 1.
↪000000}
```

HoaSource

This source type represents a Ambisonics sound field of arbitrary order. It is a multichannel object where the number of channels depends on the Ambisonics order N: $ch = (N+1)^2$. The audio signals (as indexed by the `"channels"` attribute, are expected to be in ACN channel order http://ambisonics.ch/standards/channels/.

The type string is `"hoa"`.

Example

```
{"type": "hoa", "channels": "0:8", "group": 0, "id": 0, "level": 1, "order": 2,
↪"priority": 0},
```

ChannelObject

Channel objects are audio signals that are routed directly to a loudspeaker (or group of loudspeakers) specified by an id.

This type is derived from `Object` and adds the `"outputChannels"` attribute. This attribute is a string contains a list of loudspeaker ids (i.e., labels). Channel objects can contain an arbitrary number of channels. The `outputChannels` must contain an entry for each channel. This can be either a single label or a list of labels enclosed in square brackets. In the latter case, the respective channel is routed to the list of loudspeakers.

An `diffuseness` attribute controls the level of decorrelation applied, from 0.0 (no decorrelation) to 1.0 (fully replayed to the decorrelation filters). OPtional attribute, default is 0.0.

If a channel is routed to more than one loudspeaker, the levels of these loudspeakers are normalised using the same norm as the respective panner (VBAP, VBIP in case of separate high-frequency panning, or diffuse panning).

Example

Single-channel object routed to a single loudspeaker:

```
{"id": 2, "channels": "3", "type": "channel", "group": 0, "priority": 0, "level":␣
↪0.50000, "diffuseness": 0.5,
  "outputChannels": "M+030"} ]
```

Alternative syntax for single-channel syntax :

```
{"id": 2, "channels": "3", "type": "channel", "group": 0, "priority": 0, "level":␣
↪0.50000, "diffuseness": 0.5,
 "outputChannels": "[M+030]"} ]
```

Single channel routed to multiple loudspeakers:

```
{"id": 2, "channels": "3", "type": "channel", "group": 0, "priority": 0, "level":␣
↪0.50000, "diffuseness": 0.5,
 "outputChannels": "[M+030, M-030]"} ]
```

Multiple channels routed to single or multiple loudspeakers:

```
{ "id": 2, "channels": "4:8", "type": "channel", "group": 0, "priority": 0,
  "level": 0.350000, "diffuseness": 0.25,
  "outputChannels": "M+000, [M+030], [M-030, U+030], U+110"}]
```

PointSourceWithReverb

PointSourceWithReverb is a single-channel object that adds reverb to a PointSource. It uses the type string `"pointreverb"`. In addition to the Object and PointSource properties it defines an attribute "room" containing the objects `"ereflect"` (early reflections) and `"lreverb"` (late reverberation). `"ereflect"` is an array of early reflection objects, consisting of IIR coefficients (`"biquadsos"`, a point source position `"position"` using the same format as in `PointSource`, and additional level and delay information.

The maximum number of discrete reflections per reverb object is a configuration parameter of the renderer.

The `"lreverb"` object contains parameter data in fixed frequency bands that are used to synthesize reverb tails.

Example

```
{ "type": "pointreverb", "channels": "4", "group": 0, "id": 1,"level": 1,
  "position": {"x": 1.5, "y": 0.0, "z": 0.0}, "priority": 0,
  "room": {
  "ereflect": [{"biquadsos": [{"a0": "1.00000e+00", "a1": "-1.05734e+00", "a2": "5.
↪69314e-01",
                              "b0": "3.87648e-01", "b1": "0.00000e+00", "b2": "0.
↪00000e+00"},
                             ( more biquad coefficients)
                             {"a0": "1.00000e+00", "a1": "-7.20132e-02", "a2": "6.
↪48827e-01",
                              "b0": "1.00000e+00", "b1": "0.00000e+00", "b2": "0.
↪00000e+00"}],
                "delay": "0.00931", "level": "0.0603584806", "position": {"az":␣
↪337.0, "el": "-1.00000", "refdist": "1.00000"} },
              ( more early reflections )
            ],
"lreverb": {"attacktime": "0.01321, 0.01321, 0.01321, 0.01321, 0.01321, 0.01321, 0.
↪01321, 0.01321, 0.01321",
            "decayconst": "-4.50698, -5.02028, -5.75817, -5.36509, -5.42654, -5.
↪62316, -5.75298, -6.41075, -11.13465",
            "level": "0.02522, 0.01052, 0.01657, 0.02744, 0.02058, 0.01679, 0.
↪01698, 0.01433, 0.00041", "delay": "0.00931" } }
```

9.2 Predefined object-based rendering primitives and renderers

9.3 Object-Based Reverberation

9.4 The loudspeaker configuration format

Loudspeaker configurations are used to tell a renderer about the loudspeaker positions and other properties of the setup. It is used primarily in the loudspeaker renderers, including binaural renderering that use a virtual loudspeaker setup internally.

This section describes the format of a loudspeaker configuration and explains the helper functions provided with a VISR installation to create configuration files.

9.4.1 Configuration file example

A loudspeaker configuration has to be specified in an XML file.

An example is given below.

```
<panningConfiguration>
  <loudspeaker id="M+000" channel="1" eq="highpass">
    <cart x="1.0" y="0.0" z="0"/>
  </loudspeaker>
  <loudspeaker id="M-030" channel="2" eq="highpass">
    <polar az="-30.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="M+030" channel="3" eq="highpass">
    <polar az="30.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="M-110" channel="4" eq="highpass">
    <polar az="-110.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="M+110" channel="5" eq="highpass">
    <polar az="110.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U-030" channel="6" eq="highpass">
    <polar az="-30.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U+030" channel="7" eq="highpass">
    <polar az="30.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U-110" channel="8" eq="highpass">
    <polar az="-110.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U+110" channel="9" eq="highpass">
    <polar az="110.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <virtualspeaker id="VoS">
    <polar az="0.0" el="-90.0" r="1.0"/>
    <route lspId="M+000" gainDB="-13.9794"/>
    <route lspId="M+030" gainDB="-13.9794"/>
    <route lspId="M-030" gainDB="-13.9794"/>
    <route lspId="M+110" gainDB="-13.9794"/>
    <route lspId="M-110" gainDB="-13.9794"/>
  </virtualspeaker>
  <triplet l1="VoS" l2="M+110" l3="M-110"/>
  <triplet l1="M-030" l2="VoS" l3="M-110"/>
  <triplet l1="M-030" l2="VoS" l3="M+000"/>
  <triplet l1="M-030" l2="U-030" l3="M+000"/>
```

(continues on next page)

(continued from previous page)

```
  <triplet l1="M+030" l2="VoS" l3="M+000"/>
  <triplet l1="M+030" l2="VoS" l3="M+110"/>
  <triplet l1="U+030" l2="U-030" l3="M+000"/>
  <triplet l1="U+030" l2="M+030" l3="M+000"/>
  <triplet l1="U-110" l2="M-030" l3="U-030"/>
  <triplet l1="U-110" l2="M-030" l3="M-110"/>
  <triplet l1="U+110" l2="U-110" l3="M-110"/>
  <triplet l1="U+110" l2="M+110" l3="M-110"/>
  <triplet l1="U+030" l2="U-110" l3="U-030"/>
  <triplet l1="U+030" l2="U+110" l3="U-110"/>
  <triplet l1="U+030" l2="U+110" l3="M+110"/>
  <triplet l1="U+030" l2="M+030" l3="M+110"/>
  <subwoofer assignedLoudspeakers="M+000, M-030, M+030, M-110, M+110, U-030, U+030,
↪ U-110, U+110"
          channel="10" delay="0" eq="lowpass" gainDB="0"
          weights="1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0"
          />
  <outputEqConfiguration numberOfBiquads="1" type="iir">
      <filterSpec name="lowpass">
         <biquad a1="-1.9688283" a2="0.96907117" b0="6.0729856e-05" b1="0.
↪00012145971" b2="6.0729856e-05"/>
      </filterSpec>
      <filterSpec name="highpass">
         <biquad a1="-1.9688283" a2="0.96907117" b0="-0.98447486" b1="1.9689497"␣
↪b2="-0.98447486"/>
      </filterSpec>
  </outputEqConfiguration>
</panningConfiguration>
```

9.4.2 Predefined configuration files

The VISR package comes with a number of preconfigured loudspeaker configurations. They are contained in the directory `$VISR_ROOT/config/`.

The subdirectory `config/generic` contains standard configurations, mainly from the ITU-R BS2051 standard.

The supported configurations are:

Name	File name	Number of loudspeakers (upper, horizontal, lower)	Dimension	Virtual loudspeakers (azimuth, elevation)	Comment
System A	bs2051-0+2+0	0, 2, 0	2D	(180,0)	Stereo
System B	bs2051-0+5+0	0, 5, 0	2D	—	5.1
System C	bs2051-2+5+0	2, 5, 0	3D	(0,90), (0,-90)	
System D	bs2051-3+7+0	3, 7, 0	3D	(0,90), (0,-90)	
System E	bs2051-4+5+0	4, 5, 0	3D	(0,90), (0,-90)	
System F	bs2051-4+5+1	4, 5, 1	3D	(0,90), (0,-90)	
System G	bs2051-4+9+0	4, 9, 0	3D	(0,90), (0,-90)	
System H	bs2051-9+10+3	9, 10, 3	3D	(0,-90)	22.2 (NHK)

For all configurations, versions with and without subwoofer channels are provided. The version with subwoofer has no suffix, the version without has `-no-subwoofer` appended to the file name. In general, the version with subwoofer should be preferred and the generated loudspeaker outputs are identical. The output channel mapping is identical for both cases. In some cases the subwoofer channels are embedded into the block of loudspeaker output channels. In the corresponding configurations without subwoofer, these channels are not used.

For the stereo configuration, an additional configuration `bs2051-0+2+0-rear-fading.xml` is provided in which sound sources are faded out as they approach 180°. In all other cases, the energy from virtual loudspeakers (as denoted in the table above) is distributed to neighboring real loudspeakers.

The subdirectories `config/isvr`, `config/surrey`, and `config/bbc` contain examples of actual listening rooms. The functions to generated these configurations are contained in the subdirectories `scripts/` within these folders.

9.4.3 Generation functions

To ease the creation of generation functions, the VISR framework provides several Python functions to create the XML configuration files from a number of loudspeaker coordinates and additional optional parameters. These functions are contained in the Python module `loudspeakerconfig`. If the VISR framework was installed through a binary installer and Python was configured as described in *Configuration*, then the package can be directly imported, e.g.,

```
import loudspeakerconfig
loudspeakerconfig.createArrayConfigFile( ... )
```

or

```
from loudspeakerconfig import createArrayConfigFile( ... )
```

The main function in this module is `createArrayConfigFile()`. It takes a set of loudspeaker coordinates, an output file name, and a large set of additional options:

`loudspeakerconfig.`**`createArrayConfigFile`**(*outputFileName*, *lspPositions*, *twoDconfig=False*, *sphericalPositions=False*, *channelIndices=None*, *loudspeakerLabels=None*, *triplets=None*, *distanceDelay=False*, *distanceAttenuation=False*, *lspDelays=None*, *lspGainDB=None*, *eqConfiguration=None*, *virtualLoudspeakers=[]*, *subwooferConfig=[]*, *comment=None*, *speedOfSound=340.0*)

Generate a loudspeaker configuration XML file.

Parameters

- **`outputFileName`** (*`string`*) – The file name of the XML file to be written. This can be a file name or path. The file extension (typically .xml) must be provided by the user.
- **`lspPositions`** (*`array-like, 3xL or 2xL, where L is the number of loudspeakers.`*) – Provide the loudspeaker in Cartesian coordinates, relative to the centre of the array.
- **`twoDconfig`** (*`bool, optional`*) – Whether the loudspeaker configuration is 2D or 3D (default). In the former case, the lspPositions parameter does not need to have a third row, and it is ignored if present. If twoDconfig if True, then the loudspeaker coordinates in the do not have an "z" or "el" coordinate. Likewise, the triangulation "triplets" consist only of two loudspeakers.
- **`sphericalPositions`** (*`bool, optional`*) – Specify whether the loudspeaker and virtual loudspeaker positions are written in spherical (True) or Cartesian coordinates (False). Default is Cartesian.
- **`channelIndices`** (*`array-like, optional`*) – A list of output integer channel indices, one for each real loudspeaker. Optional argument, if not provided, consecutive indices starting from 1 are assigned. If provided, the length of the array must match the number of real loudspeakers, and indices must be unique.
- **`loudspeakerLabels`** (*`array-like, optional`*) – A list of strings containing alphanumerical labels for the real loudspeakers. Labels must be unique, consist of the characters 'a-zA-Z0-9&()+:_-', one for each real loudspeaker. The labels are used to reference loudspeakers in triplets, virtual loudspeaker routings, and subwoofer configs. Optional parameter. If not provided, labels of the form 'lsp_i' with i=1,2,... are generated.
- **`triplets`** (*`array-like, optional.`*) – A loudspeaker triangulation. To be provided as a list of arrays consisting of three (or two in case of a 2D configuration) loudspeaker labels. Labels must match existing values of the loudspeakerLabels parameter. Optional parameter, to be provided only in special cases. By default, the triangulation is computed internally.
- **`distanceDelay`** (*`bool, optional`*) – Whether the loudspeaker signals are delayed such that they arrive simultaneously in the array centre. This can be used if the loudspeaker distances to the centre ar not equal. In this case the farthest loudspeaker gets a delay of 0 s, and closer loudpeakers a positive delay. The distance compensation delay is added to the `lspDelays` parameter (if present). Optional attribute. The default (False) means no distance attenuation is used.
- **`distanceAttenuation`** (*`bool, optional`*) – Whether the loudspeaker gains shall be scaled if the loudspeaker distances are not 1.0. In this case, a 1/r distance law is applied such that the farthest loudspeaker gets a scaling factor of 0 dB, and lower factors are assigned to loudspeakers closer to the centre. The gain factors are applied on top of the optional parameter `lspGainDB`, if present. Optional attribute. Default is False (no distance attenutation applied)

- **lspDelays** (*array-like, optional*) – An array of delay values to be applied tothe loudspeakers. Values are to be provided in seconds. If not provided, no delays are applied. If specified, the length of the array must match the number of real loudspeakers.
- **lspGainDB** (*array-like, optional.*) – An array of gain values (in dB) to adjust the output gains of the real loudpeakers. If provided, the length must match the number of real loudspeakers. By default, no additional gains are applied.
- **virtualLoudspeakers** (*array of dicts, optional*) – Provide a set of additional virtual/phantom/dead/imaginary loudspeaker nodes to adjust the triangulation of the array. Each entry is a dict consisting of the following key-value pairs.
  - ”id”: A alphanumeric id, following the same rules as the loudspeaker indices. Must be unique across all real and imaginary loudspeakers.
  - ”pos”: A 3- or vector containing the position in Cartesian coordinates. 2 elements are allowed for 2D setups.
  - ”routing”: Specification how the panning gains calculated for this loudspeaker are distributed to neighbouring real loudspeakers. Provided as a list of tuples (label, gain), where label is the id of a real loudspeaker and gain is a linear gain value. Optional element, if not given, the energy of the virtual loudspeaker is discarded.

  Optional argument. No virtual loudspeakers are created if not specified.
- **eqConfiguration** (*array of structures (dicts), optional*) – Define a set of EQ filters to be applied to loudspeaker and subwoofer output channels. Each entry of the list is a dict containing the following key-value pairs.
  - ”name”: A unique, nonempty id that is referenced in loudspeaker and subwoofer specifications.
  - ”filter”: A list of biquad definitions, where each element is a dictionary containing the keys ‘b’ and ‘a’ that represent the numerator and denominator of the transfer function. ‘b’ must be a 3-element numeric vector, and ‘a’ a three- or two-element numeric vector. In the latter case, the leading coefficient is assumed to be 1, i.e., a normalised transfer function.
  - ”loudspeakers”: A list of loudspeaker labels (real loudspeakers) to whom the eq is applied.
- **subwooferConfig** (*array of dicts, optional*) – A list of subwoofer specifications, where each entry is a dictionary with the following key-value pairs:
  - ”name”: A string to name the subwoofer. If not provided, a default name will be generated.
  - ”channel”: An output channel number for the subwoofer signal. Must be unique across all loudspeakers and subwoofers.
  - ”assignedSpeakers”: A list of ids of (real) loudspeakers. The signals of these loudspeakers are used in the computation of the subwoofer signal.
  - ”weights”: An optional weighting applied to the loudspeaker signals of the the assigned loudspeakers. If provided, it must be an array-like sequence with the same length as assignedSpeakers. If not given, all assigned speakers are weighted equally with factor “1.0”.
- **comment** (*string, optional*) – Optional string to be written as an XML comment at the head of the file.

Examples

A minimal example of a 3D configuration:

```
createArrayConfigFile( 'bs2051-4+5+0.xml',
                       lspPositions = lspPos,
                       twoDconfig = False,
                       sphericalPositions=True,
                       channelIndices = [1, 2, 3, 5, 6, 7, 8, 9, 10],
                       loudspeakerLabels =  ["M+030", "M-030", "M+000", "M+110
↪", "M-110",
                          "U+030", "U-030", "U+110", "U-110"  ],
                       virtualLoudspeakers = [ { "id": "VotD", "pos": [0.0, 0.
↪0,-1.0],
                                                 "routing": [ ("M+030", 0.2), ("M-
↪030", 0.2),
                                                  ("M+000", 0.2), ("M+110", 0.2), (
↪"M-110", 0.2) ] }]
```

The function *createArrayConfigFromSofa()* can be used to create configuration files from a SOFA file to be used, for example in a virtual loudspeaker renderer (*visr_bst.VirtualLoudspeakerRenderer*):

loudspeakerconfig.**createArrayConfigFromSofa**(*sofaFile*, *xmlFile=None*, *lspLabels=None*, *twoDSetup=False*, *virtualLoudspeakers=[]*)

Create a loudspeaker configuraton file from a SOFA file containing a number of emitters representing loudspeakers.

Parameters

- **sofaFile** (*string*) – A file path to a SOFA file.
- **xmlFile** (*string, optional*) – Path of the XML output file to be written. Optional argument, if not provided, the SOFA file path is used with the extension replaced by ".xml"
- **lspLabels** (*list of strings, optional*) – List of loudspeaker labels, must match the number of emitters in the SOFA files. If not provided, numbered labels are automatically generated.
- **twoDSetup** (*bool, optional*) – Flag specifying whether the aray is to be considered plane (True) or 3D (False). Optional value, dafault is False (3D).
- **virtualLoudspeakers** (*list, optional*) – A list of virtual loudspeakers to be added to the setup. Each entry must be a Python dict as decribed in the function *loudspeakerconfig.createArrayConfigFile()*.

9.4.4 Format description

The root node of the XML file is `<panningConfiguration>`. This root element supports the following optional attributes:

isInfinite Whether the loudspeakers are regarded as point sources located on the unit sphere (`false`) or as plane waves, corresponding to an infinite distance (`true`). The default value is `false`.

dimension Whether the setup is considered as a 2-dimensional configuration (value `2`) or as three-dimensional (`3`, thedefault). In the 2D case, the array is considered in the x-y plane , and the `z` or `el` attributes of the loudspeaker positions are not evaluated. In this case, the triplet specifications consist of two indices only (technically they are pairs, not triplets).

Within the `<panningConfiguration>` root element, the following elements are supported:

<loudspeaker> Represents a reproduction loudspeaker. The position is encoded either in a `<cart>` node representing the cartesian coordinates in the `x`, `y` and `z` attributes (floating point values in meter), or a `<polar>` node with the attributes `az` and `el` (azimuth and elevation, both in degree) and `r` (radius, in meter).

The `<loudspeaker>` nodes supports for a number of attributes:

- `id` A mandatory, non-empty string identification for the loudspeaker, which must be unique across all `<loudspeaker>` and `<virtualspeaker>` (see below) elements. Permitted are alpha-numeric characters, numbers, and the characters "@&()+/:_-". ID strings are case-sensitive.
- `channel` The output channel number (sound card channel) for this loudspeaker. Logical channel indices start from 1. Each channel must be assigned at most once over the set of all loudspeaker and subwoofers of the setup..
- `gainDB` or `gain` Additional gain adjustment for this loudspeaker, either in linear scale or in dB (floating-point values. The default value is 1.0 or 0 dB. `gainDB` or `gain` are mutually exclusive.
- `delay` Delay adjustment to be applied to this loudspeaker as a floating-point value in seconds. The default value is 0.0).
- `eq` An optional output equalisation filter to be applied for this loudspeaker. Specified as a non-empty string that needs to match an `filterSpec` element in the `outputEqConfiguration` element (see below). If not given, no EQ is applied to for this loudspeaker.

`<virtualspeaker>` An additional vertex added to the triangulation that does not correspond to a physical loudspeaker. Consist of a numerical `id` attribute and a position specified either as a `<cart>` or a `<polar>` node (see `<loudspeaker>` specification).

The `<virtualspeaker>` node provides the following configuration options:

- A mandatory, nonempty and unique attribute `id` that follows the same rules as for the `<loudspeaker>` elements.
- A number of `route` sub-elements that specify how the energy from this virtual loudspeaker is routed to real loudspeakers. The `route` element has the following attributes: * `lspId`: The ID of an existing real loudspeaker. * `gainDB`: A scaling factor with which the gain of the virtual loudspeaker is distributed to the real loudspeaker.

  In the above example, the routing specification is given by

```
<virtualspeaker id="VoS">
 <polar az="0.0" el="-90.0" r="1.0"/>
 <route lspId="M+000" gainDB="-13.9794"/>
 <route lspId="M+030" gainDB="-13.9794"/>
 <route lspId="M-030" gainDB="-13.9794"/>
 <route lspId="M+110" gainDB="-13.9794"/>
 <route lspId="M-110" gainDB="-13.9794"/>
</virtualspeaker>
```

  That means that the energy of the virtual speaker `"vos"` is routed to five surrounding speakers, with a scaling factor of 13.97 dB each.

`<subwoofer>` Specify a subwoofer channel. In the current implementation, the loudspeaker are weighted and mixed into an arbitray number of subwoofer channels. The attributes are:

- `assignedLoudspeakers` The loudspeaker signals (given as a sequence of logical loudspeaker IDs) that contribute to the subwoofer signals. Given as comma-separated list of loudspeaker index or loudspeaker ranges. Index sequences are similar to Matlab array definitions, except that thes commas separating the parts of the sequence are compulsory.

  Complex example:

```
assignedLoudspeakers = "1, 3,4,5:7, 2, 8:-3:1"
```

- `weights` Optional weights (linear scale) that scale the contributions of the assigned speakers to the subwoofer signal. Given as a sequence of comma-separated linear-scale gain values, Matlab ranges are also allowed. The number of elements must match the `assignedLoudspeakers` index list. Optional value, the default option assigns 1.0 for all assigned loudspeakers. Example: "0:0.2:1.0, 1, 1, 1:-0.2:0".
- `gainDB` or `gain` Additional gain adjustment for this subwoofer, either in linear scale or in dB (floating-point valus, default 1.0 / 0 dB). Applied on top of the `weight` attributes to the summed subwoofer signal. See the `<loudspeaker>` specification.

- `delay` Delay adjustment for this (floating-point value in seconds, default 0.0). See the `<loudspeaker>` specification.

`<triplet>` Loudspeaker triplet specified by the attributes `l1`, `l2`, and `l3`. The values of `l1`, `l2`, and `l3` must correspond to IDs of existing real or virtual loudspeakers. In case of a 2D setup, only `l1` and `l2` are evaluated.

Note: At the time being, triplet specifications must be generated externally and placed in the configuration file. This is typically done by creating a Delaunay triangulation on the sphere, which can be done in Matlab or Python.

Future versions of the loudspeaker renderer might perform the triangulation internally, or might not require a conventional triangulation at all. In these cases, is it possible that the renderer ignores or internally adapts the specified triplets.

`outputEqConfiguration` This optional element must occur at most once. It provides a global specification for equalisation filters for loudspeakers and subwoofers.

```
<outputEqConfiguration  type="iir" numberOfBiquads="1">
  <filterSpec name="lowpass">
    <biquad a1="-1.9688283" a2="0.96907117" b0="6.0729856e-05" b1="0.
↪00012145971" b2="6.0729856e-05"/>
  </filterSpec>
  <filterSpec name="highpass">
    <biquad a1="-1.9688283" a2="0.96907117" b0="-0.98447486" b1="1.9689497" b2=
↪"-0.98447486"/>
  </filterSpec>
</outputEqConfiguration>
```

The attributes are:

- `type`: The type of the output filters. At the moment, only IIR filters provide as second-order sections (biquads) are supported. Thus, the value `"iir"` must be set.
- `numberOfBiquads`: This value is specific to the `"iir"` filter type.

The filters are described in `filterSpec` elements. These are identifed by a `name` attribute, which must be an non-empty string unique across all `filterSpec` elements. For the type `iir`, a `filterSpec` element consists of at most `numberOfBiquad` nodes of type `biquad`, which represent the coefficients of one second-order IIR (biquad) section. This is done through the attributes `a1`, `a2`, `b0`, `b1`, `b2` that represent the coefficients of the normalised transfer function

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

CHAPTER

TEN

VISR COMPONENT REFERENCE

10.1 Standard rendering component library (`rcl`)

Note: This section shows how the documentation for components will look like in the future. The content is encoded in docstrings in the Python files of the class definitions for Python components, and in the Pythin bindings in case of C++ components. The documentation is extracted and formatted using `sphinx.autodoc`.

10.1.1 Module overview

VISR default component library.

At the moment, this module contains atomic components (whereas the composite ones are in signalflows) for historic reasons. In the future, however, this will change do a topical orgainisation.

10.1.2 Class `rcl.Add`

class rcl.**Add**(*self: rcl.Add*, *context: visr.SignalFlowContext*, *name: str*, *parent: visr.CompositeComponent=None*, *width: int*, *numInputs: int*) → None

Component for adding multichannel multichannel audio signals.

input_<i>" Audio input signals to be added, numbered from 0..'numInputs'-1. The width is determined by the constructor parameter *width*."

output Multichannel audio output signal, width determined by paramter **width**.

Constructor, creates an instance of Add.

Parameters

- **context** (*visr.SignalFlowContext*) – Common audio processing parameters (e.g., sampling rate and block size)
- **name** (*string*) – Name of the component.
- **parent** (*visr.CompositeComponent or None*) – The composite component that contains the present object, or None for a top-level component.
- **numInputs** (*int*) – The number of input ports (addends)

process(*self: rcl.Add*) → None

10.1.3 Class `rcl.BiquadIirFilter`

class rcl.**BiquadIirFilter**(**args*, ***kwargs*)

Multichannel IIR filtering component based on second-order IIR sections (biquads).

Audio ports: input: Multichannel audio signal, the witdth is determined by the constructor parameter *numberOfChannels*. output: Multichannel output signal, width is determined by the constructor parameter *numberOfChannels*.

Parameter ports:

eqInput: Optional parameter input port for receiving updated EQ settings of type `pml.BiquadMatrixParam`
This port is activated by the constructor parameter *controlInputs* (default: `True`)

Overloaded function.

1. __init__(self: rcl.BiquadIirFilter, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, numberOfBiquads: int, controlInput: bool=True) -> None

Constructor that initialises all biquad IIR sections the default value (flat EQ).

Parameters

- **`context`** – (visr.SignalFlowContext) Common audio processing parameters (e.g., sampling rate and block size)
- **`name`** – (string) Name of the component.
- **`parent`** – (`visr.CompositeComponent` or None) The composite component that contains the present object, or None in case of a top-level component.
- **`numberOfChannels`** – (int) The number of individual audio channels processed.
- **`numberOfBiquads`** – (int) The number of second-order sections processed per channels.

2. __init__(self: rcl.BiquadIirFilter, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, numberOfBiquads: int, initialBiquad: rbbl.BiquadCoefficientFloat, controlInput: bool=True) -> None

Constructor initialising all biquad IIR sections to the same given value.

3. __init__(self: rcl.BiquadIirFilter, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, numberOfBiquads: int, initialBiquads: rbbl.BiquadCoefficientListFloat, controlInput: bool=True) -> None

Constructor initialising all channels to the same sequence of biquad IIR sections

4. __init__(self: rcl.BiquadIirFilter, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, numberOfBiquads: int, initialBiquads: rbbl.BiquadCoefficientMatrixFloat, controlInput: bool=True) -> None

Constructor initialising the biquad IIR sections to individual values.

10.1.4 Class `rcl.DelayMatrix`

class `rcl.`**DelayMatrix**(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: rcl.DelayMatrix, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent=None) -> None

2. __init__(self: rcl.DelayMatrix, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent=None, numberOfInputs: int, numberOfOutputs: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str, methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayMatrix.ControlPortConfig=ControlPortConfig.No, initialDelay: float=0.0, initialGain: float=1.0) -> None

3. __init__(self: rcl.DelayMatrix, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent=None, numberOfInputs: int, numberOfOutputs: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str, methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayMatrix.ControlPortConfig=ControlPortConfig.No, initialDelay: efl.BasicMatrixFloat=0.0, initialGain: efl.BasicMatrixFloat=1.0) -> None

setup(**args*, ***kwargs*)
Overloaded function.

1. setup(self: rcl.DelayMatrix, numberOfInputs: int, numberOfOutputs: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str, methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayMatrix.ControlPortConfig=ControlPortConfig.No, initialDelay: float=0.0, initialGain: float=1.0) -> None

2. setup(self: rcl.DelayMatrix, numberOfInputs: int, numberOfOutputs: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str, methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayMatrix.ControlPortConfig=ControlPortConfig.No, initialDelays: efl.BasicMatrixFloat, initialGains: efl.BasicMatrixFloat) -> None

10.1.5 Class `rcl.DelayVector`

class rcl.**DelayVector**(**args*, ***kwargs*)
Overloaded function.

1. __init__(self: rcl.DelayVector, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent=None) -> None

2. __init__(self: rcl.DelayVector, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str, methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayVector.ControlPortConfig=ControlPortConfig.No, initialDelay: float=0.0, initialGain: float=1.0) -> None

3. __init__(self: rcl.DelayVector, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str='lagrangeOrder3', methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayVector.ControlPortConfig=ControlPortConfig.No, initialDelay: efl.BasicVectorFloat=<pml.VectorParameterFloat object at 0x61376e2d0>, initialGain: efl.BasicVectorFloat=<pml.VectorParameterFloat object at 0x61376e308>) -> None

4. __init__(self: rcl.DelayVector, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str='lagrangeOrder3', methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayVector.ControlPortConfig=ControlPortConfig.No, initialDelay: numpy.ndarray[float32], initialGain: numpy.ndarray[float32]) -> None

Constructor taking Python lists or NumPy arrays as initial gain and delay values.

5. __init__(self: rcl.DelayVector, context: visr.SignalFlowContext, name: str, parent: visr.CompositeComponent, numberOfChannels: int, interpolationSteps: int=1024, maxDelay: float=3.0, interpolationType: str='lagrangeOrder3', methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add, controlInputs: rcl.DelayVector.ControlPortConfig=ControlPortConfig.No, initialDelay: List[float], initialGain: List[float]) -> None

Constructor taking Python lists or NumPy arrays as initial gain and delay values.

setup(*self: rcl.DelayVector*, *numberOfChannels: int*, *interpolationSteps: int=1024*, *maxDelay: float=3.0*, *interpolationType: str*, *methodDelayPolicy: rcl.DelayMatrix.MethodDelayPolicy=MethodDelayPolicy.Add*, *controlInputs: rcl.DelayVector.ControlPortConfig=ControlPortConfig.No*, *initialDelay: float=0.0*, *initialGain: float=1.0*) → None

10.2 The Binaural Synthesis Toolkit (VISR-BST)

10.2.1 Tutorial

Note: This tutorial is based on the AES e-Brief:

Franck, A., Costantini, G., Pike, C., and Fazi, F. M., "An Open Realtime Binaural Synthesis Toolkit for Audio Research," in Proc. Audio Eng. Soc. 144th Conv., Milano, Italy, 2018, Engineering Brief."

Binaural synthesis has gained fundamental importance both as a practical sound reproduction method and as a tool in audio research. Binaural rendering requires significant implementation effort, especially if head movement tracking or dynamic sound scenes are required, thus impeding audio research. For this reason we propose the Binaural Synthesis Toolkit (BST), a portable, open source, and extensible software package for binaural synthesis. In this paper we present the design of the BST and the three rendering approaches currently implemented. In contrast to most other software, the BST can easily be adapted and extended by users. The Binaural Synthesis Toolkit is released as an open software package as a flexible solution for binaural reproduction and to foster reproducible research in this field.

Introduction

Binaural synthesis aims at recreating spatial audio by recreating binaural signals at the listener's ears *[B1]*, using either headphones or loudspeakers. While binaural technology is an area of active research for a long time, the shift of music consumption towards mobile listening, object-based content, as well as the increasing importance of augmented and virtual reality (AR/VR) applications emphasize the increasing significance of binaural reproduction. In addition, binaural techniques are an important tool in many areas of audio research and development, from basic perceptual experiments to auralization of acoustic environments or the evaluation of spatial sound reproduction *[B2]*.

Regardless of the application, synthesizing binaural content invariably comprises a number of software building blocks, e.g., HRTF/BRIR selection and/or interpolation, signal filtering, modeling and applying interaural time and level differences, etc. *[B3]*. Dynamic binaural synthesis significantly increases the plausibility of reproduction by including dynamic cues as head movements *[B4]*, but requires both a real-time implementation and additional DSP functionality as time-variant delays, dynamic filter updates, and filter crossfading techniques. This implies a considerable implementation effort for research based on binaural synthesis, increases the likelihood of errors due to implementation effects, and makes it difficult to reproduce or evaluate the research of others. This argument is in line with the increasing importance of software in (audio) research, e.g., *[B5]*, and the generally growing awareness of reproducible research, e.g., *[B6]*.

For this reason we introduce the Binaural Synthesis Toolkit (BST) as an open source, portable, and extensible software library for real-time and offline binaural synthesis. Our intention is to provide baseline implementations for main binaural rendering schemes as well as DSP building blocks that enable the modification of existing renderers as well as the implementation of new rendering approaches.

The objective of its paper is to describe the architecture of the BST to enable its use as well as its adaptation by the audio community. In essence, the BST is a set of processing components implemented within the VISR — an open software rendering framework for audio rendering *[B7]* — and preconfigured renderers built upon these components. At the moment, three renderers are provided, namely HRIR-based dynamic synthesis, e.g., *[B4]*, virtual loudspeaker synthesis (also termed room scanning *[B8]*), and binaural rendering based on higher order Ambisonics (HOA), e.g., *[B9]*. BRIR/HRIR data can be provided in the AES69-2015 format (SOFA) *[B10]*

(http://www.sofaconventions.org), allowing for arbitrary HRIR/BRIR measurement grids, and enabling the use of a wide range of impulse response datasets.

Compared to existing software projects supporting binaural synthesis, for instance Spat *[B9]* or the SoundScape Renderer (SSR) *[B11]*, the BST offers several new possibilities. On the one hand, its modular structure is designed for easy adaptation and extension rather than providing a fixed functionality. To a large extent, this is achieved by the Python language interface of the underlying VISR, and the fact that most high-level BST components are implemented in Python. On the other hand, while most other software projects are mainly for real-time use, BST components can be used both in real-time and in fully parametrizable offline simulations using the same code base. This makes the BST an effective tool for algorithm development. Again, these capabilities are enabled mainly by the Python language integration of the BST and the underlying VISR.

This paper is structured as follows. The underlying VISR framework and how its features influence the design and the uses of the BST is briefly outlined in Sec. *Introduction*. Section *Preconfigured Binaural Renderers* discusses the three rendering approaches currently implemented in the BST, their use, configuration, and optional features. The main building blocks of these renderers, which can also be used to adapt the BST or to implement different synthesis methods, are outlined in section Sec. *Rendering Building Blocks*. Section *Application Examples* shows practical examples of using the BST, while Sec. *Conclusion* summarizes the paper.

The VISR Framework

The binaural synthesis toolkit is based on the VISR (Versatile Interactive Scene Renderer) framework, an open-source, portable, and extensible software for audio processing *[B7]*. It is being developed as part of the S3A project (http://www.s3a-spatialaudio.org) *[B12]*. At the moment, it is supported on Linux (Intel and Raspberry Pi), Mac OS X, and Windows. VISR is a general-purpose audio processing software with emphasis on, but not limited to, multichannel and object-based audio. This section outlines the main features of the VISR framework and the implications on design and usage of the BST.

Component-Based Design

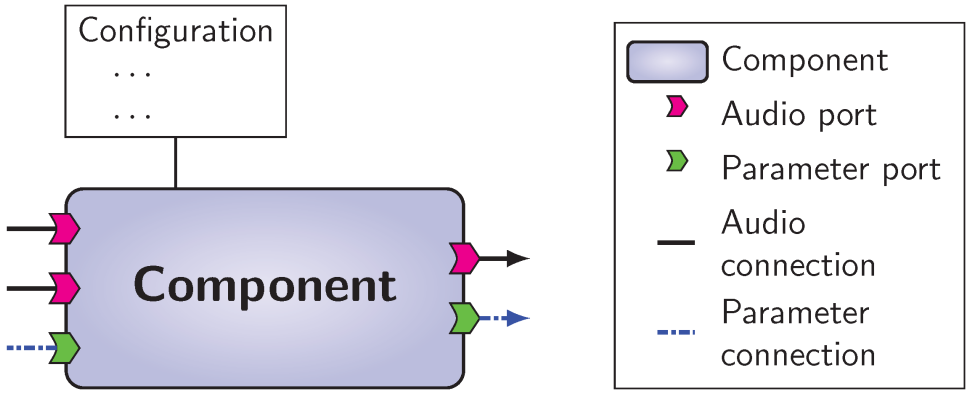

Fig. 1: General interface of a VISR component.

VISR is a software framework, which means that it enables a systematic reuse of functionalities and is designed for extension by users. To this, all processing tasks are implemented within *components*, software entities that communicate with other components and the external environment through a defined, common interface. Fig. *General interface of a VISR component.* depicts the general structure of a component.

Ports

Data inputs and outputs to components are represented by *ports*. They enable configurable, directional flow of information between components or with the outside environment. There are two distinct types of ports: audio and parameter ports. Audio ports accept or create multichannel audio signals with an arbitrary, configurable number

of single/mono audio signal waveforms, which is referred as the *width* of the port. Audio ports are configured with unique name, a width and a sample type such as `float` or `int16`.

Parameter ports, on the other hand, convey control and parameter information between components or from and to the external environment. Parameter data is significantly more diverse than audio data. For example, parameter data used in the BST includes vectors of gain or delay values, FIR or IIR filter coefficients, audio object metadata, and structures to represent the listener's orientation. In addition to the data type, there are also different communication semantics for parameters. For example, data can change in each iteration of the audio processing, be updated only sporadically, or communicated through messages queues. In VISR, these semantics are termed *communication protocols* and form an additional property of a parameter port. The semantics described above are implemented by the communication protocols `SharedData`, `DoubleBuffering`, and `MessageQueue`, respectively. Several parameter types feature additional configuration data, such as the dimension of a matrix parameter. In the VISR framework, such options are passed in `ParameterConfig` objects. This allows extensive type checking, for instance to ensure that only matrix parameter of matching dimensions are connected. Combining these features, a parameter port is described by these properties: a unique name, a parameter type, a communication protocol type and an optional parameter configuration object.

Hierarchical Signal Flows

To create and reuse more complex functionality out of existing building blocks, VISR signal flows can be structured hierarchically. To this end, there are two different kinds of components in VISR, *atomic* and *composite*. They have the same external interface, that means that they can be used in the same way. Figures *Atomic VISR component.* and *Composite VISR component.* schematically depict these two types.

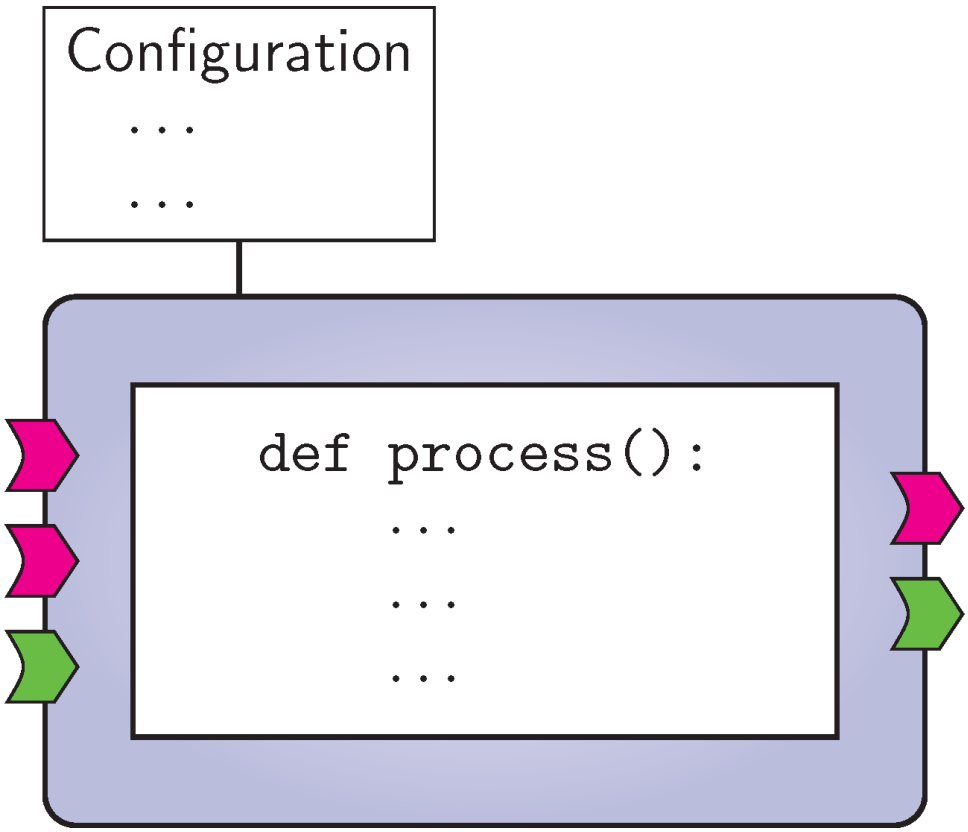

Fig. 2: Atomic VISR component.

Atomic components implement processing task in program code, e.g., in C++ or Python. They feature a constructor which may take a variety of configuration options to tailor the behaviour of the component and to initialize its state. The operation of an atomic component is implemented in the `process` method.

In contrast, a *composite component* contains a set of interconnected components (atomic or composite) to define its behaviour. This allows the specification of more complex signal flows in terms of existing functionality, but

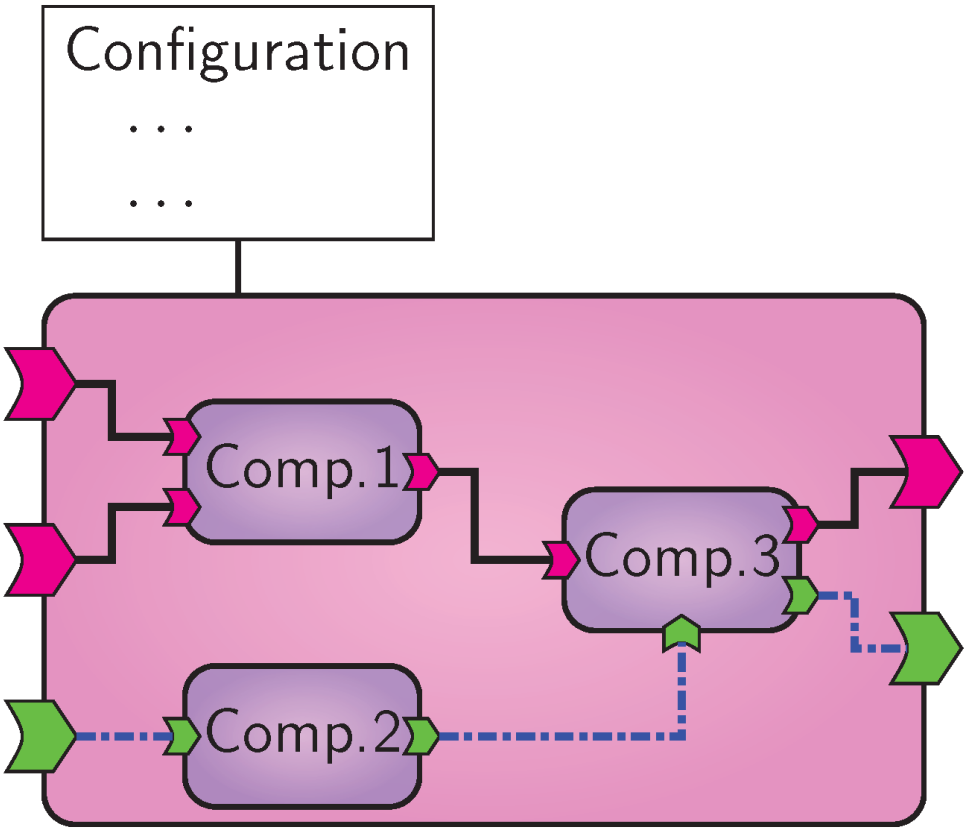

Fig. 3: Composite VISR component.

also the reuse of such complex signal flows. As their atomic counterparts, they may take a rich set of constructor options. These can control which contained components are constructed, how they are configured, and how they are connected. It is worth noting that nested components do not impair computational efficiency because the hierarchy is flattened at initialisation time and therefore not visible to the runtime engine.

This hierarchical structure has far-reaching consequences for the design and the use of the BST. Firstly, it allows for an easy use of existing VISR components in BST components. Secondly, BST renderers can easily be augmented by additional functionality, such as receiving and decoding of object metadata or headtracking information, by wrapping it into a new composite component. Thirdly, BST functionality can be conveniently integrated into larger audio applications implemented in the VISR, for instance the transaural loudspeaker array rendering described in Sec. *Transaural Loudspeaker Array*.

Standard Component Library

The runtime component library (`rcl`) of the VISR framework contains a number of components for general-purpose DSP and object-based audio operations. They are typically implemented in C++ and therefore relatively efficient. The rcl library includes arithmetic operations on multichannel signals, gain vectors and matrices, delay lines, FIR and IIR filtering blocks, but also network senders and receivers and components for decoding and handling of object audio metadata.

Runtime Engine

A key objective of the VISR framework is to enable users to focus on their processing task – performed in a component – while automating tedious tasks, such as error checking, communication between components, or interfacing audio hardware, as far as possible. The rendering runtime library (rrl) serves this purpose. Starting from a top-level component, it is only necessary to construct an object of type `AudioSignalFlow` for this component. All operations from consistency checking to the initialization of memory buffers and data structures

for rendering is performed by this object. The audiointerfaces library provides abstractions for different audio interface APIs (such as Jack, PortAudio, or ASIO). Realtime rendering is started by connecting the SignalFlow object to an audiointerfaces object.

Python interface

While the core of the VISR framework is implemented in C++, it provides a full application programming interface (API) for the Python programming language. This is to enable users to adapt or extend signal flows more productively, using an interpreted language with a more accessible, readable syntax and enabling the use of rich libraries for numeric computing and DSP, such as NumPy and SciPy *[B13]*. The Python API can be used in three principal ways:

Configuring and running signal flows Components can be created and configured from the interactive Python interpreters or script files. This makes this task fully programmable and removes the need for external scripts to configure renders. In the same way, audio interfaces can be configured, instantiated and started from within Python, enabling realtime rendering from within an interactive interpreter.

Extending and creating signal flow As described above, complex signal flows are typically created as composite components. This can be done in Python by deriving a class from the base class `visr.CompositeComponent`. The behavior of the signal flow is defined in the class' constructor by creating external ports, contained components, and their interconnections. Instances of this class can be used for realtime rendering from the Python interpreter, as described above, or from a standalone application. Most of the BST renderer signal flows are implemented in this way, ensuring readability and easy extension by users.

Adding atomic functionality In the same way as composites, atomic components can be implemented by deriving from `visr.AtomicComponent`. This involves implementing the constructor set up the component and the `process()` method that performs the run-time processing. The resulting objects can be embedded in either Python or C++ composite components (via a helper class `PythonWrapper`). In the BST toolkit, the controller components that define the logic of the specific binaural rendering approaches are implemented as atomic components in Python. This language choice allows for rapid prototyping, comprehensible code, and easy adaptation.

Offline Rendering

By virtue of the Python integration, signal flows implemented as components are not limited to realtime rendering, but can also be executed in an offline programming environment. Because the top-level audio and parameter ports of a component can be accessed externally, dynamic rendering features such as moving objects or head movements can be simulated in a deterministic way. In the majority of uses, this is most conveniently performed in an interactive Python environment. Applications of this feature range from regression tests of atomic components or complex composite signal flows, performance simulations, to offline rendering of complete sound scenes. A full characterization of the offline rendering support is beyond the scope of this paper, interested readers are referred to *[B7]*.

Use in multiple software environments

In addition to realtime rendering and offline Python scripting, VISR components can also be embedded into audio software environments such as digital audio workstations (DAWs) plugins, or Max/MSP or Pd externals. This means that parts of the BST can be used from these applications, creating new tools and integrating into the workflow of more researchers and creatives. Support libraries are provided to ease this task by reducing the amount of code required for this embedding. Again, a full discussion of this paper is beyond the scope of this paper, see *[B7]* for a discussion.

Preconfigured Binaural Renderers

The VISR Binaural Synthesis Toolkit contains prepackaged, configurable renderers (or signal flows) for three major binaural synthesis strategies. They are implemented as composite VISR components in Python. On the one hand, this means that they can be readily used either as standalone binaural renderers or as part of larger audio processing schemes. On the other hand, the use of Python allows for easy modification and extension. This section describes the general structure and the configuration options of these three renderers. Features common to all approaches, such as an option to include headphone transfer function (HPTF) compensation filters, are not described here. Entities such as components, ports, or configuration options are set in a monospaced font, e.g., `input`.

Dynamic HRIR-Based Synthesis

This approach renders sound objects, typically point sources or plane waves represented by position metadata and a mono signal, using spatial datasets of head-related impulse responses (HRIR) or, equivalently, head-related transfer functions (HRTFs), e.g., *[B3][B4]*. This approach is widely used in audio research and for practical reproduction and is well-suited for object-based audio as well as AR/VR applications. In its basic form, it synthesizes sound scenes under freefield conditions, and is therefore often augmented by a reverberation engine, e.g., *[B14]*.

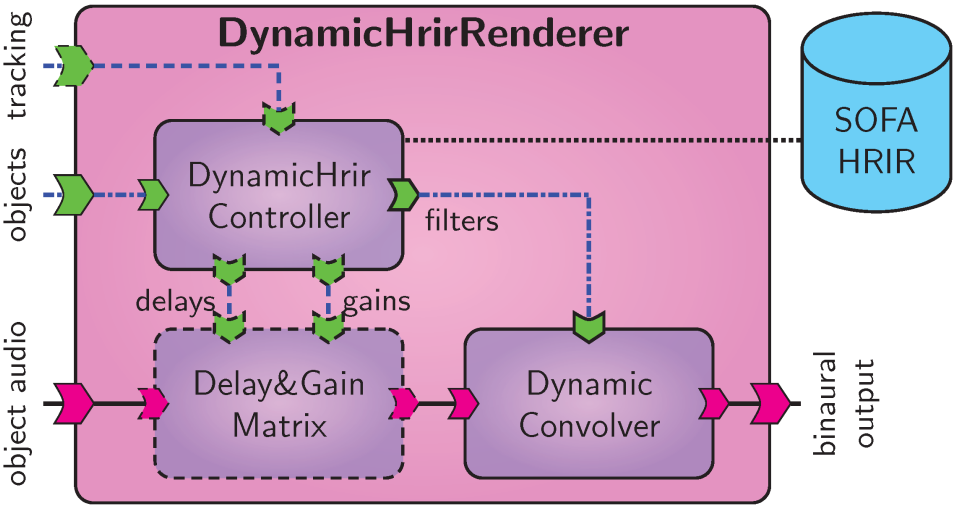

Fig. 4: Dynamic binaural synthesis rendering component. Optional parts are dashed.

The signal flow of the BST dynamic HRIR renderer is shown in *Dynamic binaural synthesis rendering component. Optional parts are dashed.*. It is implemented as a composite VISR component named `DynamicHrirRenderer`. The logic of the synthesis is encapsulated in the atomic component `DynamicHrirController`. It receives object metadata and determines a pair of HRIR filters for each object, which are transmitted to the DSP components. The controller is initialized with a spatial HRIR dataset and the corresponding grid locations of the IRs. Audio object metadata, including positions and levels, are received through the parameter input port `objects`. If present, the listener's head orientation is received through the optional input `tracking` and incorporated in the HRIR calculation. At the moment, two HRIR calculation methods are supported: nearest-neighbour selection and barycentric interpolation using a Delaunay triangulation, e.g., *[B15]*. The generated IRs, one per sound object and ear, are transmitted through the output port `filters` to the `DynamicConvolver` component, which performs time-variant MIMO (multiple-input, multiple-output) FFT-based convolution of the object audio signals and combines them into binaural output. Depending on the configuration, the `DynamicConvolver` uses crossfading to reduce audible artifacts when changing filters.

The HRTF dataset, including the measurement position grid, can be provided in the AES69:2015 (SOFA) format *[B10]*. Optionally the rendering component accepts preprocessed datasets where the HRIRs are time-aligned and the onset delays are kept apart (e.g., in the `Data.Delay` field of the SOFA format). Applying the delays separately can improve HRIR interpolation quality, reduces audible effects when updating filters, and can therefore enable the use of coarser HRIR datasets, e.g., *[B3]*. Because the pure delay part of the HRIRs dominates the ITD cue of the synthesized binaural signal, this also provides a means to use alternative ITD models instead of

the measured IR delays, or to implement ITD individualization *[B16]*. In the same way, the filter gain may be calculated separately, for instance to simulate near-field source effects *[B14]*. In either of these cases, the dynamic delay and/or gain coefficients are calculated in the `DynamicHrirController`, and the optional component `DelayGainMatrix` is instantiated to apply these values prior to the convolution. As described in Sec. *Delay Lines: Vectors and Matrices*, the delay/gain components support configurable fractional delay filtering and smooth parameter updates, thus ensuring audio quality and reducing audible artifacts in dynamic rendering scenarios.

Higher Order Ambisonics-Based Synthesis

A second approach, termed HOA (Higher Order Ambisonics)-based synthesis, is based on a spherical harmonics (SH) representation of a spatial sound scene, i.e., higher-order B-format. First-order B-format binaural synthesis has been proposed, e.g., in *[B17][B18]*, and extended to higher Ambisonic orders, e.g., *[B19]*. This scene-based rendering approach forms, for example, the basis of the spatial audio synthesis in Facebook's Audio360 Google's Resonance Audio SDK https://github.com/resonance-audio/resonance-audio-web-sdk.

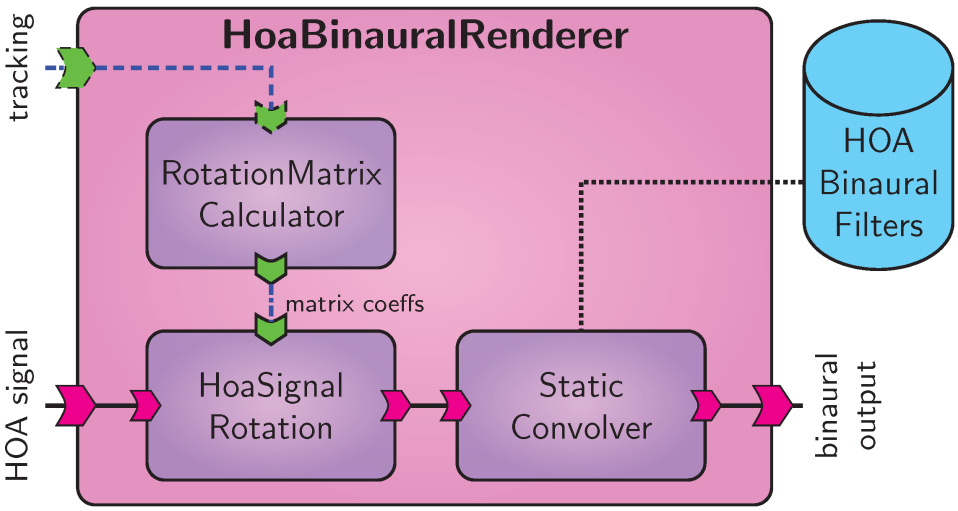

Fig. 5: Binaural synthesis based on HOA.

The signal flow of the generic HOA synthesis renderer is depicted in Fig. *Binaural synthesis based on HOA.*. The component `HoaBinauralRenderer` accepts a HOA input signal, i.e., higher-order B-format, of a selectable order L, consisting of $(L+1)^2$ channels. If head tracking is enabled, the component `RotationMatrixCalculator` computes a rotation matrix for spherical harmonics using a recurrent formula *[B20]*. These coefficients are applied to the B-format signal in the gain matrix component `HoaSignalRotation`, effectively rotating the sound field to compensate for the listener's orientation. The component of this signal are filtered with a bank of $(L+1)^2$ static FIR filters for each ear *[B19]* in the `StaticConvolver` component, which also performs an ear-wise summation of the filtered signals to yield the binaural output signal.

Figure *HOA binaural synthesis of object-based scenes.* shows a variant of the HOA synthesis that operates on an object-based scene. Component `ObjectToHoaCoefficients` transforms the object metadata into a set of SH coefficients for each object. If head tracking is active, the component `HoaCoefficientRotation` receives orientation data from the optional `tracking` input and applies a rotation matrix to the HOA coefficients. The `HoaSignalEncoder` component uses these data to encode the object signals into a B-format signal. This representation is transformed to a binaural signal in the same way as in the generic HOA binaural renderer. The advantage of this approach is that the rotation is performed on the SH coefficients, that is a much lower rate than the audio sampling frequency, and is consequently more efficient.

Virtual Loudspeaker Rendering/ Binaural Room Scanning

The third principal approach implemented in the BST, denoted as virtual loudspeaker rendering, uses binaural room impulse responses of a multi-loudspeaker setup in a given acoustic environment to recreate the listening

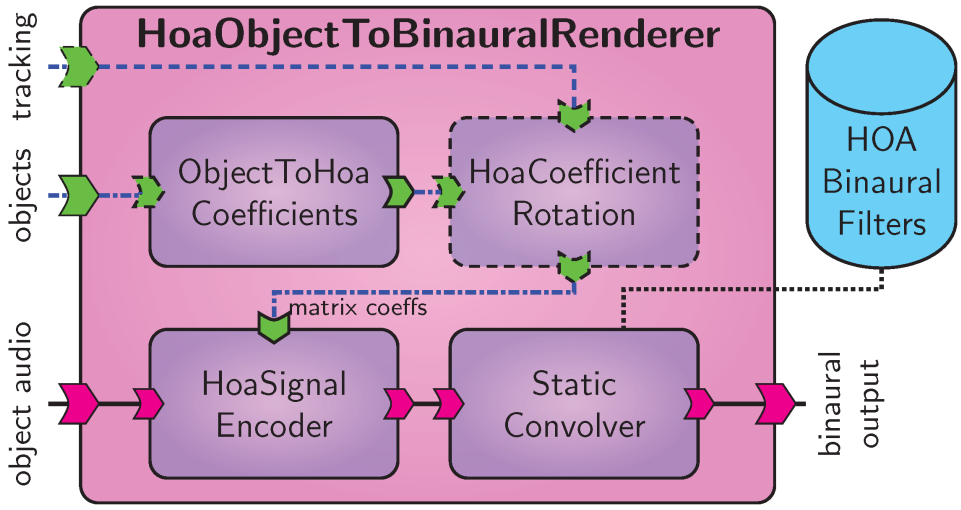

Fig. 6: HOA binaural synthesis of object-based scenes.

experience in that room. For this reason it is also referred to as *binaural room scanning [B8][B4]*. Headtracking can be used to incorporate the listener's orientation by switching or interpolating between BRIR data provided for a grid of orientations. While the BST supports both 2D and 3D grids, exiting datasets, e.g., *[B21][B22]* are typically restricted to orientations in the horizontal plane because of the measurement effort and data size. In contrast to the aforementioned methods, this approach does not operate on audio objects but on loudspeaker signals. It is therefore used to reproduce channel-based content or to transform the output of loudspeaker-based rendering methods to binaural, e.g., *[B23]*.

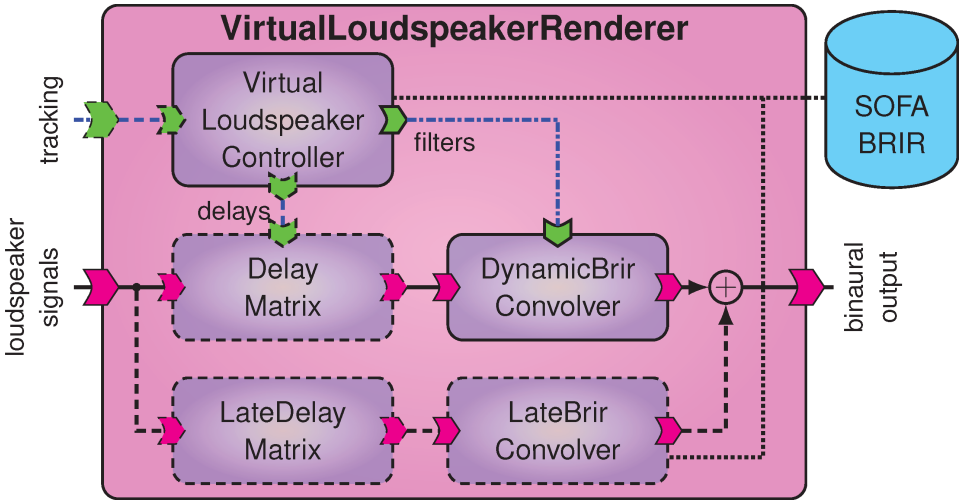

Fig. 7: Virtual loudspeaker renderer.

The signal flow of the `Virtual-Loudspeaker-Renderer` is displayed Fig. *Virtual loudspeaker renderer.*. As in the dynamic HRIR renderer (*Dynamic HRIR-Based Synthesis*), the logic is implemented in a controller component, `VirtualLoudspeakerController`. It loads the BRIR dataset from a SOFA file and uses the optional parameter input port `tracking` to select and potentially interpolate the room impulse responses according to the head rotation. The resulting BRIRs are sent to the `DynamicBrirConvolution` component, where the L loudspeaker signals are convolved, optionally crossfaded, and summed to form the binaural output.

If the onset delays are extracted from the BRIRs, they are dynamically computed in the controller and applied in the optional `DynamicDelayMatrix` component. As in case dynamic HRIR synthesis, this can help to improve filter interpolation, reduce switching artifacts, and allow for coarser BRIR datasets. To reduce memory requirements and the computational load due to filter interpolation and switching, the late part of the BRIRs can optionally be rendered statically, i.e., independent of the head rotation, as described in *[B21]*. In this case, the loudspeaker signals are processed through an additional branch consisting of a fixed delay matrix

`LateDelayMatrix`, which applies the time offset (mixing time) of the late reverberation tail, and the static convolver `LateConvolutionEngine`. The result is combined with the dynamically convolved early part to form the binaural output signal.

Rendering Building Blocks

After describing the ready-made binaural synthesis approaches provided in the BST, this section explains the main DSP building blocks used to implement these algorithms in more detail. On the one hand, this is to provide more insight into the workings of these renderers. On the other hand, these components can also be used to adapt or extend the BST, or to implement alternative synthesis approaches.

As outlined in Sec. *The VISR Framework*, most of the present generic DSP functionality is implemented as C++ components in the VISR `rcl` library. Python language interfaces are provided to enable their use in Python components. They are typically application-independent and therefore highly configurable. For instance, the widths (i.e., the number of individual signals) of input and output ports can be changed. Other parameters, such as gains, delay values, or filters, can be either set statically or updated during runtime. To this end, parameter ports to update these values can be activated in the component's initialization, typically using an argument `controlInputs` with a component-specific enumeration type. In a `DelayVector` object, for example, the setting `controlInputs = rcl.DelayVector.PortConfigConfig.Delay` activates the parameter input port to set delay values at runtime.

Convolution Kernels

Convolution with FIR filters representing HRIR or BRIR data is an essential operation in binaural synthesis. The VISR framework provides different components for multiple-input, multiple-output FIR filtering using fast convolution techniques.

The most basic, `rcl.FirFilterMatrix`, enables arbitrary sets of filtering operations between individual signals of its variable-width input and output ports. To this end, so-called *routings* — lists of elements formed by an input index, an output index and a filter id — can be provided either during initialisation or at runtime. In this way, widely different filtering operations can be performed by the same component. Examples are multichannel channel-wise filtering, dense MIMO filter matrix, or application-specific topologies such as filtering a set of object signals to a left and right HRIR each, and summing the results ear-wise. FFT and inverse FFT transforms are reused for multiple filtering operations where possible, improving efficiency compared to simple channel-wise convolution. Depending on the configuration, both the routing points and the FIR filters can be exchanged at runtime, and the changes are performed instantaneously.

To avoid artifacts due to such filter switching operations, the component `rcl.CrossFadingFirFilterMatrix` extends this filter matrix by a crossfading implementation to enable smooth transitions. To this end, an additional configuration parameter `interpolationSteps` is added to specify the duration of the transition to a new filter. At the moment, this operation is performed in the time domain, thus incurring significant increase in computational complexity. This can be partly alleviated by frequency-domain filter exchange strategies, e.g., *[B24]*, which can be implemented without changing the component's interface.

Gain Vectors and Matrices

Gains are used, for example, to apply audio object levels or distance attenuation, or to perform matrix operations as rotations on HOA signals. The VISR `rcl` library provides three component types for vector and matrix delays. `rcl.GainVector` applies individual gains to an arbitrary-width audio signal. `rcl.GainMatrix` performs a matrixing operation between an audio input and an audio output signal of arbitrary, possibly different widths using a dense matrix of gain coefficients. If a significant part of the gains are zero, `rcl.GainMatrixSparse` performs this matrixing more efficiently. To this end, `rcl.GainMatrixSparse` is configured with a set of *routings*, similar to those used with the convolution components, to describe the location of the nonzero gains.

All gain components can be configured with initial gains values to allow static operation with fixed coefficients. Optionally, a parameter input `gain` can be activated to receive run-time updates. In this case, the gains are

smoothly and linearly changed to their new value within a transition period defined by the optional parameter `interpolationSteps` (in samples). This ensures click-free operation.

Delay Lines: Vectors and Matrices

Time delays are ubiquitous in binaural synthesis, with uses ranging from modeling of propagation delays, simulation of Doppler effects, the separate application of HRIR/BRIR onset delays, or the incorporation of analytic ITD models. The VISR provides two components, `rcl.DelayVector` and `rcl.DelayMatrix` for applying channel-wise and matrix time delays to arbitrary-width signals. For time-variant delay operation, an optional parameter input `delay` is instantiated to receive runtime updates. For artifact-free operation, the time delays are transitioned smoothly to the new values, based on a `interpolationsteps` configuration parameter as described above. In addition, fractional delay (FD) filtering is essential for maintaining audio quality in time-variant delay lines. To this end, a FD algorithm can be selected using the `interpolationType` parameter. Currently the delay components support nearest-neighbor interpolation and Lagrange interpolation of arbitrary order based on an efficient linear-complexity implementation *[B25]*. Since signal delays are often applied in combination with a scaling, i.e., gain, and because these operations can be combined efficiently, the VISR delay components support an optional `gain` parameter input port that can be activated if required.

Head Tracking Data Receivers

For real-time dynamic rendering, data from head tracking devices must be supplied to the `tracking` input port. To this end we provide an extensible library of tracking data receivers. These are implemented in Python for conciseness and to make adaptation to other devices easier. Tracking receiver components transform the device-specific tracking information into the parameter type `pml.ListenerPosition` that represents a listener position (not used in the current binaural renderers) and orientation within the VISR framework. Currently, the following devices are supported: The following devices are supported at the moment. Razor AHRS headtracker (https://github.com/Razor-AHRS/razor-9dof-ahrs), MrHeadTracker (https://git.iem.at/DIY/MrHeadTracker/), HTC VIVE Tracker (https://www.vive.com/us/vive-tracker/), and Intel RealSense (https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html). The separation of the tracking data receivers and the example implementations in Python will ease the creation of components for additional tracking devices.

Application Examples

In this section we provide a number of usage examples for the Binaural Synthesis Toolkit.

Offline algorithm evaluation

Algorithm development often requires deterministic, repeatable simulation runs and the generation of objective results such as waveforms, plots, or error metrics. The VISR Python integration provides an environment for simulating dynamic scenarios using the same binaural synthesis algorithms as in the real-time case in an offline, scripted fashion. To this end, one of the BST renderers (or an adapted version) is configured and instantiated in a Python script. Within this script, the renderer is executed in a block-by block fashion, providing audio object signals, object metadata and (optionally) listener orientation updates. The resulting binaural output signal can be displayed, saved as an audio file, or analyzed using the rich scientific computing libraries of Python. Thus it is easy to switch between conventional audio research and realtime rendering using the same code base.

Realtime HRIR synthesis of object-based scenes

In this use case, a complex object-based scene, e.g., *[B26]* is reproduced binaurally. The dynamic scene is stored in the Audio Definition Model (ADM) *[B27]* format and played through a specialized software player, generating a multichannel audio signal containing the object waveforms, and the object metadata as a UDP network stream. A new composite component `ObjectSceneRenderer`, depicted in Fig. *Application: Object scene rendering.*,

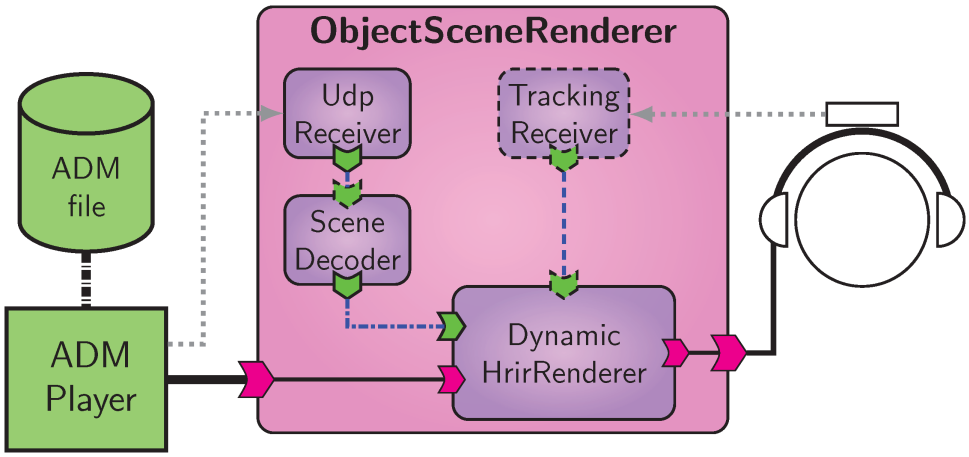

Fig. 8: Application: Object scene rendering.

is created to reproduce such content. It contains a `UdpReceiver` and a `SceneDecoder` component, both part of the VISR `rcl` library, to receive object metadata and to decode it into the internal representation. This and the multichannel object audio is passed to the `DynamicHrirRenderer`. Optionally, information from a head tracking device is decoded in the `HeadtrackingReceiver` component and passed to the `tracking` input of the renderer.

Transaural Loudspeaker Array

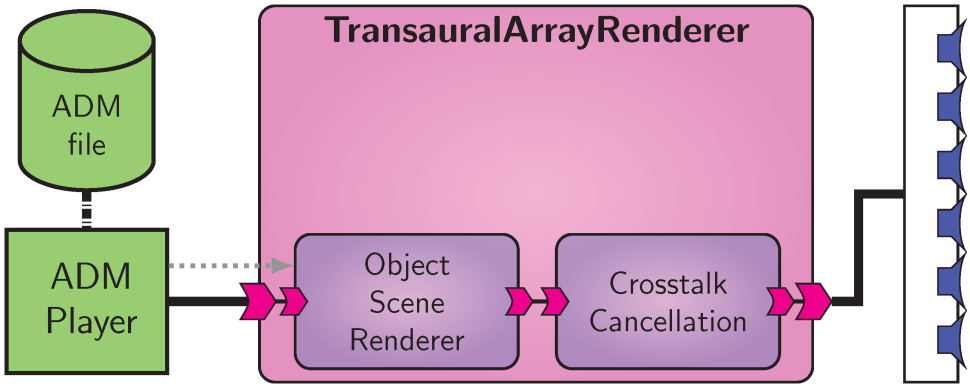

Fig. 9: Application: Transaural array rendering

A variant of the object scene rendering for transaural array reproduction e.g., *[B28]* is depicted in Fig. *Application: Transaural array rendering*. It features a new top-level composite component `TransauralArrayRendering`, which connects a `ObjectSceneRenderer` to the transaural-specific crosstalk cancellation algorithm, which is implemented using the VISR framework. The output of the latter is sent to a multichannel loudspeaker array.

These examples show how the BST can be used in practical reproduction scenarios, and how it can be embedded in more complex audio algorithms.

Conclusion

In this paper we introduced the Binaural Synthesis Toolkit, an open, portable software package for binaural rendering over headphones and transaural systems. It features an extensible, component-based design that is based on the open VISR framework. Currently, the Binaural Synthesis Toolkit provides baseline implementations for three general binaural rendering schemes: Dynamic HRTF-based synthesis, rendering based on spherical higher order Ambisonics, and binaural room scanning. A distinguishing feature, compared to existing software, is the

C++/Python interoperability of the BST and the underlying VISR framework. It facilitates accessible, readable code, ease of modification, and the ability to use the binaural rendering context in different applications or more complex processing systems.

While providing baseline rendering methods in an open, extensible toolkit will reamin the focus of the BST, future extensions might include a dynamic room model, listener position adaptation, or more sophisticated signal processing methods for convolution, filter interpolation and updating, or fractional delay filtering. We alsop aim to make the renderers and building available as easy-to use tools such as digital audio workstation (DAW) plugins or Max/MSP externals.

The intended use of the BST is as a tool for practical binaural sound reproduction and to foster reproducible research in audio.

References

10.2.2 Class reference

The renderer classes

Class `DynamicHrirRenderer`

class visr_bst.**DynamicHrirRenderer**(*context*, *name*, *parent*, *numberOfObjects*, *, *sofaFile=None*, *hrirPositions=None*, *hrirData=None*, *hrirDelays=None*, *headOrientation=None*, *headTracking=True*, *dynamicITD=True*, *dynamicILD=True*, *hrirInterpolation=True*, *filterCrossfading=False*, *interpolatingConvolver=False*, *fftImplementation='default'*)

Rendering component for dynamic binaural synthesis based on HRTFs/HRIRs.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, holds the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **numberOfObjects** (*int*) – Maximum number of audio objects
- **sofaFile** (*str, optional*) – Optional SOFA for loading loaded the HRIR and associated data (HRIR measurement positions and delays) If not provided, the information must be provided by the hrirPositions and hrirData arguments.
- **hrirPositions** (*numpy.ndarray, optional*) – Optional way to provide the measurement grid for the BRIR listener view directions. If a SOFA file is provided, this is optional and overrides the listener view data in the file. Otherwise this argument is mandatory. Dimension #grid directions x (dimension of position argument)
- **hrirData** (*numpy.ndarray, optional*) – Optional way to provide the BRIR data. Dimension: #grid directions x #ears (2) # x #loudspeakers x #ir length
- **hrirDelays** (*numpy.ndarray, optional*) – Optional BRIR delays. If a SOFA file is given, this argument overrides a potential delay setting from the file. Otherwise, no extra delays are applied unless this option is provided. Dimension: #grid directions x #ears(2) x # loudspeakers

- **headOrientation** (*array-like, optional*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic head tracking is supported. If True, a parameter input with type pml.ListenerPosition and protocol pml.DoubleBufffering is created.
- **dynamicITD** (*bool, optional*) – Whether the ITD is applied separately. That requires preprocessed HRIR data
- **dynamicILD** (*bool, optional*) – Whether the ILD is computed and applied separately. At the moment this feature is not used (apart from applying the object gains)
- **hrirInterpolation** (*bool, optional*) – Whether the controller supports interpolation between neighbouring HRTF grid points. False means nearest neighbour (no interpolation), True enables barycentric interpolation.
- **filterCrossfading** (*bool, optional*) – Use a crossfading FIR filter matrix to avoid switching artifacts.
- **fftImplementation** (*string, optional*) – The FFT implementation to use. Default value enables VISR's default FFT library for the platform.

Class HoaBinauralRenderer

class visr_bst.**HoaBinauralRenderer**(*context*, *name*, *parent*, *hoaOrder=None*, *sofaFile=None*, *decodingFilters=None*, *interpolationSteps=None*, *headOrientation=None*, *headTracking=True*, *fftImplementation='default'*)

Component to render binaural audio from plane wave and point source objects using an Higher Order Ambisonics (HOA) algorithm.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, holds the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **hoaOrder** (*int or None*) – The maximum HOA order that can be reproduced. If None, the HOA order is deduced from the first dimension of the HOA filters (possibly contained in a SOFA file).
- **sofaFile** (*string or NoneType*) – A file in SOFA format containing the decoding filters. This expects the filters in the field 'Data.IR', dimensions (hoaOrder+1)**2 x 2 x irLength. If None, then the filters must be provided in 'decodingFilters' parameter.
- **decodingFilters** (*numpy.ndarray or NoneType*) – Alternative way to provide the HOA decoding filters.
- **interpolationSteps** (*int, optional*) – Number of samples to transition to new object positions after an update.
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic head tracking is active.

- **fftImplementation** (*string, optional*) – The FFT library to be used in the filtering. THe default uses VISR's default implementation for the present platform.

Class HoaObjectToBinauralRenderer

class visr_bst.**HoaObjectToBinauralRenderer**(*context*, *name*, *parent*, *numberOfObjects*, *maxHoaOrder=None*, *sofaFile=None*, *decodingFilters=None*, *interpolationSteps=None*, *headOrientation=None*, *headTracking=True*, *objectChannelAllocation=False*, *fftImplementation='default'*)

Component to render binaural audio from plane wave and point source objects using an Higher Order Ambisonics (HOA) algorithm.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, holds the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **numberOfObjects** (*int*) – The number of audio objects to be rendered.
- **maxHoaOrder** (*int or None*) – The maximum HOA order that can be reproduced. If None, the HOA order is deduced from the first dimension of the HOA filters (possibly contained in a SOFA file).
- **sofaFile** (*string or NoneType*) –
- **decodingFilters** (*numpy.ndarray or NoneType*) – Alternative way to provide the HOA decoding filters.
- **interpolationSteps** (*int*) –
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic head tracking is active.
- **objectChannelAllocation** (*bool*) – Whether the processing resources are allocated from a pool of resources (True), or whether fixed processing resources statically tied to the audio signal channels are used. Not implemented at the moment, so leave the default value (False).
- **fftImplementation** (*string, optional*) – The FFT library to be used in the filtering. THe default uses VISR's default implementation for the present platform.

class visr_bst.**VirtualLoudspeakerRenderer**(*context*, *name*, *parent*, ***, *sofaFile=None*, *hrirPositions=None*, *hrirData=None*, *hrirDelays=None*, *headOrientation=None*, *headTracking=True*, *dynamicITD=False*, *hrirInterpolation=False*, *irTruncationLength=None*, *filterCrossfading=False*, *interpolatingConvolver=False*, *staticLateSofaFile=None*, *staticLateFilters=None*, *staticLateDelays=None*, *fftImplementation='default'*)

Signal flow for rendering binaural output for a multichannel signal reproduced over a virtual loudspeaker array with corresponding BRIR data.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, a structure holding the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **sofaFile** (*string*) – BRIR database provided as a SOFA file. This is an alternative to the hrirPosition, hrirData (and optionally hrirDelays) argument. Default None means that hrirData and hrirPosition must be provided.
- **hrirPositions** (*numpy.ndarray*) – Optional way to provide the measurement grid for the BRIR listener view directions. If a SOFA file is provided, this is optional and overrides the listener view data in the file. Otherwise this argument is mandatory. Dimension #grid directions x (dimension of position argument)
- **hrirData** (*numpy.ndarray*) – Optional way to provide the BRIR data. Dimension: #grid directions x #ears (2) # x #loudspeakers x #ir length
- **hrirDelays** (*numpy.ndarray*) – Optional BRIR delays. If a SOFA file is given, this argument overrides a potential delay setting from the file. Otherwise, no extra delays are applied unless this option is provided. Dimension: #grid directions x #ears(2) x # loudspeakers
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic headTracking is active. If True, an control input "tracking" is created.
- **dynamicITD** (*bool*) – Whether the delay part of th BRIRs is applied separately to the (delay-free) BRIRs.
- **hrirInterpolation** (*bool*) – Whether BRIRs are interpolated for the current head orientation. If False, a nearest-neighbour interpolation is used.
- **irTruncationLength** (*int*) – Maximum number of samples of the BRIR impulse responses. Functional only if the BRIR is provided in a SOFA file.
- **filterCrossfading** (*bool*) – Whether dynamic BRIR changes are crossfaded (True) or switched immediately (False)
- **interpolatingConvolver** (*bool*) – Whether the interpolating convolver option is used. If True, the convolver stores all BRIR filters, and the controller sends only interpolation coefficient messages to select the BRIR filters and their interpolation ratios.
- **staticLateSofaFile** (*string, optional*) – Name of a file containing a static (i.e., head orientation-independent) late part of the BRIRs. Optional argument, might be used as an alternative to the staticLateFilters argument, but these options are mutually exclusive. If neither is given, no static late part is used. The fields 'Data.IR' and the 'Data.Delay' are used.
- **staticLateFilters** (*numpy.ndarray, optional*) – Matrix containing a static, head position-independent part of the BRIRs. This option is mutually exclusive to staticLateSofaFile. If none of these is given, no separate static late part is rendered. Dimension: 2 x #numberOfLoudspeakers x firLength

- **staticLateDelays** (*numpy.ndarray, optional*) – Time delay of the late static BRIRs per loudspeaker. Optional attribute, only used if late static BRIR coefficients are provided. Dimension: 2 x #loudspeakers
- **fftImplementation** (*string*) – The FFT implementation to be used in the convolver. the default value selects the system default.

class visr_bst.**ObjectToVirtualLoudspeakerRenderer**(*context*, *name*, *parent*, *, *numberOfObjects*, *sofaFile=None*, *hrirPositions=None*, *hrirData=None*, *hrirDelays=None*, *headOrientation=None*, *headTracking=True*, *dynamicITD=False*, *hrirInterpolation=False*, *irTruncationLength=None*, *filterCrossfading=False*, *interpolatingConvolver=False*, *staticLateSofaFile=None*, *staticLateFilters=None*, *staticLateDelays=None*, *fftImplementation='default'*, *loudspeakerConfiguration=None*, *loudspeakerRouting=None*, *objectRendererOptions={}*)

Signal flow for rendering an object-based scene over a virtual loudspeaker binaural renderer.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, a structure holding the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **sofaFile** (*string*) – BRIR database provided as a SOFA file. This is an alternative to the hrirPosition, hrirData (and optionally hrirDelays) argument. Default None means that hrirData and hrirPosition must be provided.
- **hrirPositions** (*numpy.ndarray*) – Optional way to provide the measurement grid for the BRIR listener view directions. If a SOFA file is provided, this is optional and overrides the listener view data in the file. Otherwise this argument is mandatory. Dimension #grid directions x (dimension of position argument)
- **hrirData** (*numpy.ndarray*) – Optional way to provide the BRIR data. Dimension: #grid directions x #ears (2) # x #loudspeakers x #ir length
- **hrirDelays** (*numpy.ndarray*) – Optional BRIR delays. If a SOFA file is given, this argument overrides a potential delay setting from the file. Otherwise, no extra delays are applied unless this option is provided. Dimension: #grid directions x #ears(2) x # loudspeakers
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic headTracking is active. If True, an control input "tracking" is created.

- **dynamicITD** (*bool*) – Whether the delay part of th BRIRs is applied separately to the (delay-free) BRIRs.
- **hrirInterpolation** (*bool*) – Whether BRIRs are interpolated for the current head orientation. If False, a nearest-neighbour interpolation is used.
- **irTruncationLength** (*int*) – Maximum number of samples of the BRIR impulse responses. Functional only if the BRIR is provided in a SOFA file.
- **filterCrossfading** (*bool*) – Whether dynamic BRIR changes are crossfaded (True) or switched immediately (False)
- **interpolatingConvolver** (*bool*) – Whether the interpolating convolver option is used. If True, the convolver stores all BRIR filters, and the controller sends only interpolation coefficient messages to select the BRIR filters and their interpolation ratios.
- **staticLateSofaFile** (*string, optional*) – Name of a file containing a static (i.e., head orientation-independent) late part of the BRIRs. Optional argument, might be used as an alternative to the staticLateFilters argument, but these options are mutually exclusive. If neither is given, no static late part is used. The fields 'Data.IR' and the 'Data.Delay' are used.
- **staticLateFilters** (*numpy.ndarray, optional*) – Matrix containing a static, head position-independent part of the BRIRs. This option is mutually exclusive to staticLateSofaFile. If none of these is given, no separate static late part is rendered. Dimension: 2 x #numberOfLoudspeakers x firLength
- **staticLateDelays** (*numpy.ndarray, optional*) – Time delay of the late static BRIRs per loudspeaker. Optional attribute, only used if late static BRIR coefficients are provided. Dimension: 2 x #loudspeakers
- **fftImplementation** (*string*) – The FFT implementation to be used in the convolver. the default value selects the system default.
- **loudspeakerConfiguration** (*panning.LoudspeakerArray*) – Loudspeaker configuration object used in the ob ject renderer. Must not be None
- **loudspeakerRouting** (*array-like list of integers or None*) – Routing indices from the outputs of the object renderer to the inputs of the binaural virtual loudspeaker renderer. If empty, the outputs of the object renderer are connected to the first inputs of the virt. lsp renderer.
- **objectRendererOptions** (*dict*) – Keyword arguments passed to the object renderer (rcl.CoreRenderer). This may involve all optional arguments for this class apart from loudspeakerConfiguration, numberOfInputs, and numberOfOutputs. If provided, these paremters are overwritten by the values determined from the binaural renderer's configuration.

Realtime rendering classes

class visr_bst.**RealtimeDynamicHrirRenderer**(*context*, *name*, *parent*, ***, *numberOfObjects*, *sofaFile=None*, *hrirPositions=None*, *hrirData=None*, *hrirDelays=None*, *headOrientation=None*, *dynamicITD=False*, *dynamicILD=False*, *hrirInterpolation=False*, *filterCrossfading=False*, *fftImplementation='default'*, *headTrackingReceiver=None*, *headTrackingPositionalArguments=None*, *headTrackingKeywordArguments=None*, *sceneReceiveUdpPort=None*)

VISR component for realtime audio rendering of object-based scenes using a 'dynamic HRIR' approach.

It contains a DynamicHrirRenderer component, but optionally adds a receiver component for head tracking devices and real-time receipt of object metadata from UDP network packets.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, holds the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **numberOfObjects** (*int*) – Maximum number of audio objects
- **sofaFile** (*str, optional*) – Optional SOFA for loading loaded the HRIR and associated data (HRIR measurement positions and delays) If not provided, the information must be provided by the hrirPositions and hrirData arguments.
- **hrirPositions** (*numpy.ndarray, optional*) – Optional way to provide the measurement grid for the BRIR listener view directions. If a SOFA file is provided, this is optional and overrides the listener view data in the file. Otherwise this argument is mandatory. Dimension #grid directions x (dimension of position argument)
- **hrirData** (*numpy.ndarray, optional*) – Optional way to provide the BRIR data. Dimension: #grid directions x #ears (2) # x #loudspeakers x #ir length
- **hrirDelays** (*numpy.ndarray, optional*) – Optional BRIR delays. If a SOFA file is given, this argument overrides a potential delay setting from the file. Otherwise, no extra delays are applied unless this option is provided. Dimension: #grid directions x #ears(2) x # loudspeakers
- **headOrientation** (*array-like, optional*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **dynamicITD** (*bool, optional*) – Whether the ITD is applied separately. That requires preprocessed HRIR data
- **dynamicILD** (*bool, optional*) – Whether the ILD is computed and applied separately. At the moment this feature is not used (apart from applying the object gains)
- **hrirInterpolation** (*bool, optional*) – Whether the controller supports interpolation between neighbouring HRTF grid points. False means nearest neighbour (no interpolation), True enables barycentric interpolation.
- **filterCrossfading** (*bool, optional*) – Use a crossfading FIR filter matrix to avoid switching artifacts.
- **fftImplementation** (*string, optional*) – The FFT implementation to use. Default value enables VISR's default FFT library for the platform.
- **headTrackingReceiver** (*class type, optional*) – Class of the head tracking recveiver, None (default value) disables dynamic head tracking.
- **headTrackingPositionalArguments** (*tuple optional*) – Positional arguments passed to the constructor of the head tracking receiver object. Must be a tuple. If there is only a single argument, a trailing comma must be added.
- **headTrackingKeywordArguments** (*dict, optional*) – Keyword arguments passed to the constructor of the head tracking receiver. Must be a dictionary (dict)
- **sceneReceiveUdpPort** (*int, optional*) – A UDP port number where scene object metadata (in the S3A JSON format) is to be received). If not given (default), no

network receiver is instantiated, and the object exposes a top-level parameter input port "objectVectorInput"

class visr_bst.**RealtimeHoaObjectToBinauralRenderer**(*context*, *name*, *parent*, *, *numberOfObjects*, *maxHoaOrder*, *sofaFile=None*, *decodingFilters=None*, *interpolationSteps=None*, *headTracking=True*, *headOrientation=None*, *objectChannelAllocation=False*, *fftImplementation='default'*, *headTrackingReceiver=None*, *headTrackingPositionalArguments=None*, *headTrackingKeywordArguments=None*, *sceneReceiveUdpPort=None*)

VISR component for realtime audio rendering of object-based scenes using a Higher-order Ambisonics encoding of point source/plane wave objects and binaural rendering of the soiundfield representation.

It contains a HoaObjectToRenderer component, but optionally adds a receiver component for head tracking devices and real-time receipt of object metadata from UDP network packets.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, holds the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **numberOfObjects** (*int*) – The number of audio objects to be rendered.
- **maxHoaOrder** (*int*) – HOA order used for encoding the point source and plane wave objects.
- **sofaFile** (*string, optional*) – A SOFA file containing the HOA decoding filters. These are expects as a 2 x (maxHoaIrder+1)^2 array in the field Data.IR
- **decodingFilters** (*numpy.ndarray, optional*) – Alternative way to provide the HOA decoding filters. Expects a 2 x (maxHoaIrder+1)^2 matrix containing FIR coefficients.
- **interpolationSteps** (*int, optional*) – Number of samples to transition to new object positions after an update.
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic head tracking is active.
- **objectChannelAllocation** (*bool*) – Whether the processing resources are allocated from a pool of resources (True), or whether fixed processing resources statically tied to the audio signal channels are used. Not implemented at the moment, so leave the default value (False).
- **fftImplementation** (*string, optional*) – The FFT implementation to use. Default value enables VISR's default FFT library for the platform.
- **headTrackingReceiver** (*class type, optional*) – Class of the head tracking recveiver, None (default value) disables dynamic head tracking.

- **headTrackingPositionalArguments** (*tuple optional*) – Positional arguments passed to the constructor of the head tracking receiver object. Must be a tuple. If there is only a single argument, a trailing comma must be added.
- **headTrackingKeywordArguments** (*dict, optional*) – Keyword arguments passed to the constructor of the head tracking receiver. Must be a dictionary (dict)
- **sceneReceiveUdpPort** (*int, optional*) – A UDP port number where scene object metadata (in the S3A JSON format) is to be received). If not given (default), no network receiver is instantiated, and the object exposes a top-level parameter input port "objectVectorInput"

class visr_bst.**RealtimeHoaBinauralRenderer**(*context*, *name*, *parent*, *, *hoaOrder=None*, *sofaFile=None*, *decodingFilters=None*, *interpolationSteps=None*, *headTracking=True*, *headOrientation=None*, *fftImplementation='default'*, *headTrackingReceiver=None*, *headTrackingPositionalArguments=None*, *headTrackingKeywordArguments=None*)

VISR component for realtime audio rendering of Higher-order Ambisonics (HOA) audio.

It contains a HoaObjectToRenderer component, but optionally adds a receiver component for head tracking devices

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, holds the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **hoaOrder** (*optional, int or None*) – HOA order used for encoding the point source and plane wave objects. If not provided, the order is determined from the number of decoding filters (either passed as a matrix or in a SOFA file)
- **sofaFile** (*string, optional*) – A SOFA file containing the HOA decoding filters. These are expects as a 2 x (maxHoaIrder+1)^2 array in the field Data.IR
- **decodingFilters** (*numpy.ndarray, optional*) – Alternative way to provide the HOA decoding filters. Expects a 2 x (maxHoaIrder+1)^2 matrix containing FIR coefficients.
- **interpolationSteps** (*int, optional*) – Number of samples to transition to new object positions after an update.
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction
- **headTracking** (*bool*) – Whether dynamic head tracking is active.
- **fftImplementation** (*string, optional*) – The FFT implementation to use. Default value enables VISR's default FFT library for the platform.
- **headTrackingReceiver** (*class type, optional*) – Class of the head tracking recveiver, None (default value) disables dynamic head tracking.

- **headTrackingPositionalArguments** (*tuple optional*) – Positional arguments passed to the constructor of the head tracking receiver object. Must be a tuple. If there is only a single argument, a trailing comma must be added.
- **headTrackingKeywordArguments** (*dict, optional*) – Keyword arguments passed to the constructor of the head tracking receiver. Must be a dictionary (dict)

class visr_bst.**RealtimeVirtualLoudspeakerRenderer**(*context*, *name*, *parent*, ***, *sofaFile=None*, *hrirPositions=None*, *hrirData=None*, *hrirDelays=None*, *headOrientation=None*, *dynamicITD=True*, *hrirInterpolation=True*, *irTruncationLength=None*, *filterCrossfading=False*, *interpolatingConvolver=False*, *staticLateSofaFile=None*, *staticLateFilters=None*, *staticLateDelays=None*, *headTrackingReceiver=None*, *headTrackingPositionalArguments=None*, *headTrackingKeywordArguments=None*, *fftImplementation='default'*)

Binaural renderer to transform a set of loudspeaker signals into a binaural output.

This class extends visr_bst.VirtualLoudspeakerRenderer by a configurable head tracking receiver, making it suitable for realtime use.

Constructor.

Parameters

- **context** (*visr.SignalFlowContext*) – Standard visr.Component construction argument, a structure holding the block size and the sampling frequency
- **name** (*string*) – Name of the component, Standard visr.Component construction argument
- **parent** (*visr.CompositeComponent*) – Containing component if there is one, None if this is a top-level component of the signal flow.
- **sofaFile** (*string*) – BRIR database provided as a SOFA file. This is an alternative to the hrirPosition, hrirData (and optionally hrirDelays) argument. Default None means that hrirData and hrirPosition must be provided.
- **hrirPositions** (*numpy.ndarray*) – Optional way to provide the measurement grid for the BRIR listener view directions. If a SOFA file is provided, this is optional and overrides the listener view data in the file. Otherwise this argument is mandatory. Dimension #grid directions x (dimension of position argument)
- **hrirData** (*numpy.ndarray*) – Optional way to provide the BRIR data. Dimension: #grid directions x #ears (2) # x #loudspeakers x #ir length
- **hrirDelays** (*numpy.ndarray*) – Optional BRIR delays. If a SOFA file is given, this argument overrides a potential delay setting from the file. Otherwise, no extra delays are applied unless this option is provided. Dimension: #grid directions x #ears(2) x # loudspeakers
- **headOrientation** (*array-like*) – Head orientation in spherical coordinates (2- or 3-element vector or list). Either a static orientation (when no tracking is used), or the initial view direction

- **headTracking** (*bool*) – Whether dynamic headTracking is active. If True, an control input "tracking" is created.
- **dynamicITD** (*bool*) – Whether the delay part of th BRIRs is applied separately to the (delay-free) BRIRs.
- **hrirInterpolation** (*bool*) – Whether BRIRs are interpolated for the current head orientation. If False, a nearest-neighbour interpolation is used.
- **irTruncationLength** (*int*) – Maximum number of samples of the BRIR impulse responses. Functional only if the BRIR is provided in a SOFA file.
- **filterCrossfading** (*bool*) – Whether dynamic BRIR changes are crossfaded (True) or switched immediately (False)
- **interpolatingConvolver** (*bool*) – Whether the interpolating convolver option is used. If True, the convolver stores all BRIR filters, and the controller sends only interpolation coefficient messages to select the BRIR filters and their interpolation ratios.
- **staticLateSofaFile** (*string, optional*) – Name of a file containing a static (i.e., head orientation-independent) late part of the BRIRs. Optional argument, might be used as an alternative to the staticLateFilters argument, but these options are mutually exclusive. If neither is given, no static late part is used. The fields 'Data.IR' and the 'Data.Delay' are used.
- **staticLateFilters** (*numpy.ndarray, optional*) – Matrix containing a static, head position-independent part of the BRIRs. This option is mutually exclusive to staticLateSofaFile. If none of these is given, no separate static late part is rendered. Dimension: 2 x #numberOfLoudspeakers x firLength
- **staticLateDelays** (*numpy.ndarray, optional*) – Time delay of the late static BRIRs per loudspeaker. Optional attribute, only used if late static BRIR coefficients are provided. Dimension: 2 x #loudspeakers
- **fftImplementation** (*string*) – The FFT implementation to be used in the convolver. the default value selects the system default.

10.3 Dynamic range control library

CHAPTER

ELEVEN

OLD CONTENTS

11.1 Examples

11.2 Tutorials

The contents of these files will be removed or moved to other parts of the documentation.

BIBLIOGRAPHY

[T1] Greg Wilson, D. A. Aruliah, C Brown, and others. Best practices for scientific computing. *PLoS Biology*, 12(1):e1001745, January 2014.

[T2] Chris Cannam, Lu'ıs Figueira, and Mark D. Plumbley. Sound software: towards software reuse in audio and music research. In *IEEE Int. Conf. Acoust., Speech Signal Process.*, 2745–2748. March 2012.

[T3] Philip Coleman, Andreas Franck, Jon Francombe, and others. An audio-visual system for object-based audio: From recording to listening. *IEEE Transactions on Multimedia*, 2018.

[T4] Yann Orlarey, Dominique Fober, and Stéphane Letz. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632, 2004.

[T5] Richard Charles Boulanger, editor. *The Csound Book*. MIT Press, 2000.

[T6] Scott Wilson, David Cottle, and Nick Collins. *The SuperCollider Book*. MIT Press, 2011.

[T7] Thibaut Carpentier, Markus Noisternig, and Olivier Warusfel. Twenty years of ircam Spat: looking back, looking forward. In *Proc. Int. Computer Music Conf.* Denton, TX, USA, September 2015.

[T8] Xavier Amatriain, Pau Arumi, and David Garcia. A framework for efficient and rapid development of cross-platform audio applications. *Multimedia Systems*, 14(1):15–32, June 2008. URL: https://doi.org/10.1007/s00530-007-0109-6.

[T9] Matthias Geier, Torben Hohn, and Sascha Spors. An open-source C++ framework for multithreaded realtime multichannel audio applications. In *Proc. Linux Audio Conf.*, 183–188. Stanford, California, USA, April 2012.

[T10] Travis E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, May 2007.

[T11] Visr download web site. http://cvssp.org/data/s3a/public/VISR/, August 2018.

[T12] Philip Mackensen, Uwe Felderhof, Günther Theile, and others. Binaural room scanning — a new tool for acoustic and psychoacoustic research. *J. Acoust. Soc. Amer.*, 105(2):1343–1344, 1999.

[T13] Chris Pike and Michael Romanov. An impulse response dataset for dynamic data-based auralization of advanced sound systems. In *Proc. Audio Eng. Soc. 142nd Conv.* Berlin, Germany, May 2017. Engineering Brief. URL: http://www.aes.org/e-lib/browse.cfm?elib=18709.

[T14] Andreas Franck, Giacomo Costantini, Chris Pike, and Filippo Maria Fazi. An open realtime binaural synthesis toolkit for audio research. In *Proc. Audio Eng. Soc. 144th Conv.* Milano, Italy, May 2018. Engineering Brief.

[T15] Richard James Hughes, Andreas Franck, Trevor J. Cox, Ben G. Shirley, and Filippo Maria Fazi. Dual frequency band amplitude panning for multichannel audio systems. In *Proc. AES Int. Conf Sound Reproduction*. Tokyo, Japan, August 2018.

[T16] Steven Diamond and Stephen Boyd. CVXPY: a Python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17(83):1–5, 2016. http://www.cvxpy.org.

[T17] ITU. Itu-r bs.2051-1, advanced sound system for programme production. 2017.

[T18] Andreas Franck, Wenwu Wang, and Filippo Maria Fazi. Sparse, $\ell _1$-optimal multi-loudspeaker panning and its relation to vector base amplitude panning. *IEEE Transactions on Audio, Speech, and Language Processing*, 25(5):996–1010, May 2017.

[B1] Rozenn Nicol. *Binaural Technology*. AES Monographs. AES, New York, NY, 2010.

[B2] Alexander Lindau and Stefan Weinzierl. Assessing the plausibility of virtual acoustic environments. *Acta Acoustica United with Acoustica*, 98(5):804–810, September 2012.

[B3] Jean-Marc Jot, Véronique Larcher, and Olivier Warusfel. Digital signal processing issues in the context of binaural and transaural stereophony. In *Proc. Audio Eng. Soc. 98th Conv.* Paris, France, February 1995.

[B4] V. Ralph Algazi and Richard O.he Duda. Headphone-based spatial sound. *IEEE Signal Processing Magazine*, 28(1):33–42, January 2011.

[B5] Chris Cannam, Lu'ıs Figueira, and Mark D. Plumbley. Sound software: towards software reuse in audio and music research. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, 2745–2748. Kyoto, Japan, March 2012.

[B6] Greg Wilson, D. A. Aruliah, C Brown, and others. Best practices for scientific computing. *PLoS Biology*, 12(1):e1001745, January 2014.

[B7] Andreas Franck and Filippo Maria Fazi. VISR – a versatile open software framework for audio signal processing. In *Proc. Audio Eng. Soc. 2018 Int. Conf. Spatial Reproduction*. Tokyo, Japan, August 2018.

[B8] Philip Mackensen, Uwe Felderhof, Günther Theile, and others. Binaural room scanning — a new tool for acoustic and psychoacoustic research. *J. Acoust. Soc. Amer.*, 105(2):1343–1344, 1999.

[B9] Jérôme Daniel, Jean-Bernard Rault, and Jean-Dominique Polack. Ambisonics encoding of other audio formats for multiple listening conditions. In *Proc. Audio Eng. Soc. 105th Conv.* San Francisco, CA, USA, September 1998. URL: http://www.aes.org/e-lib/browse.cfm?elib=8385.

[B10] Audio Engineering Society. Aes69:2015 AES standard for file exchange – Spatial acoustic data file format. 2015.

[B11] Matthias Geier and Sascha Spors. Spatial audio with the SoundScape renderer. In *Proc. 27th Tonmeistertagung / VDT Int. Conv.* Cologne, Germany, November 2012.

[B12] Philip Coleman, Andreas Franck, Teofilo de Campos, and others. An audio-visual system for object-based audio: From recording to listening. *IEEE Transactions on Multimedia*, 2018. Accepted for publication.

[B13] Travis E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, May 2007.

[B14] Jean-Marc Jot, Adam Philp, and Martin Walsh. Binaural simulation of complex acoustic scenes for interactive audio. In *Proc. Audio Eng. Soc. 121st Conv.* San Francisco, CA, USA, October 2006. URL: http://www.aes.org/e-lib/browse.cfm?elib=13784.

[B15] Hannes Gamper. Selection and interpolation of head-related transfer functions for rendering moving virtual sound sources. In *Proc. 16th Int. Conf. Digital Audio Effects*. Maynooth, Ireland, September 2013.

[B16] Alexander Lindau, Jorgos Estrella, and Stefan Weinzierl. Individualization of dynamic binaural synthesis by real time manipulation of ITD. In *Proc. Audio Eng. Soc. 128th Conv.* London, UK, May 2010. URL: http://www.aes.org/e-lib/browse.cfm?elib=15385.

[B17] Adam McKeag and David S. McGrath. Sound field format to binaural decoder with head tracking. In *Proc. Audio Eng. Soc. 6th Australian Regional Conv.* Melbourne, Australia, August 1996. URL: http://www.aes.org/e-lib/browse.cfm?elib=7477.

[B18] Jean-Marc Jot, Scott Wardle, and Veronique Larcher. Approaches to binaural synthesis. In *Proc. AES 105th Conv.* San Francisco, CA, USA, September 1998. URL: http://www.aes.org/e-lib/browse.cfm?elib=8319.

[B19] Benjamin Bernschutz, A. Vazquez Giner, C. Porschmann, and J. Arend. Binaural reproduction of plane waves with reduced modal order. *Acta Acoustica United with Acoustica*, 100(5):972–983, September 2014. URL: https://doi.org/10.3813/aaa.918777.

[B20] Joseph Ivanic and Klaus Ruedenberg. Rotation matrices for real spherical harmonics. direct determination by recursion. *J. Phys. Chem*, 100(15):6342–6347, April 1996.

[B21] Chris Pike, Frank Melchior, and Tony Tew. Assessing the plausibility of non-individualised dynamic binaural synthesis in a small room. In *AES 55th Int. Conf. Spatial Audio*. Helsinki, Finland, August 2014.

[B22] Chris Pike and Michael Romanov. An impulse response dataset for dynamic data-based auralization of advanced sound systems. In *Proc. Audio Eng. Soc. 142nd Conv.* Berlin, Germany, May 2017. Engineering Brief. URL: http://www.aes.org/e-lib/browse.cfm?elib=18709.

[B23] Mikko-Ville Laitinen and Ville Pulkki. Binaural reproduction for directional audio coding. In *Proc. IEEE Workshop Applicat. Signal Process. Audio Acoust.*, 337–340. New Paltz, NY, October 2009.

[B24] Andreas Franck. Efficient frequency-domain filter crossfading for fast convolution with application to binaural synthesis. In *Proc. Audio Eng. Soc. 55th Int. Conf. Spatial Audio*. Helsinki, Finland, August 2014.

[B25] Andreas Franck. Efficient algorithms and structures for fractional delay filtering based on Lagrange interpolation. *J. Audio Eng. Soc.*, 56(12):1036–1056, December 2008.

[B26] James Woodcock, Chris Pike, Frank Melchior, and others. Presenting the s3a object-based audio drama dataset. In *Proc. Audio Eng. Soc. 140th Conv.* Paris, France, May 2016. Engineering Brief. URL: http://www.aes.org/e-lib/browse.cfm?elib=18159.

[B27] ITU. Itu-r bs.2076-1 : audio definition model. 2017.

[B28] Marcos Simón-Gálvez and Maria Fazi, Filippo. Loudspeaker arrays for transaural reproduction. In *Proc. 22nd Int. Congr. Sound Vibration*. Florence, Italy, July 2015.

PYTHON MODULE INDEX