Jacktools - Realtime Audio Processors as Python Classes

Fons ADRIAENSEN,
Huawei German Research Center,
Riesstrasse 25,
80992 Munich,
Germany,
fons@linuxaudio.org, fons.adriaensen@tonmeister.de

Abstract

This paper introduces a set of real-time audio processing blocks that can be used as components in Python scripts. Each of them is both a Jack client and a Python class. The full power of Python can be used to control these modules, to combine them into systems of arbitrary complexity, and to interface them to anything that can be controlled from Python. The rationale behind this approach, some of the the implementations details, and possible applications are discussed.

Keywords

Jack, Python, Audio measurements, Modular audio systems

1 Introduction

Jacktools is a set of real-time audio processing blocks using Jack for audio input and output, and wrapped as Python classes. Currently the set can be divided into two types of functionality: the first is aimed at audio measurement and more technical uses, while the second contains things like an audio file player, gain controls, equalisers, convolution matrices, etc. that can be combined into general-purpose audio systems.

The origins of Jacktools go back several years, when the author required a practical tool to test real-time implementations of audio DSP algorithms. What was needed was an easy way, without having to use a compiled language or write any low-level code, to

- generate complex and accurately defined audio test signals,
- output these via Jack to the tested module, at the same time capturing its outputs,
- analyse the captured signals and present the results in a convenient way.

Python[1], and in particular the numerical and scientific extensions (Numpy[2], Scipy[3], Matplotlib[4],...) provided the ideal environment for signal generation and analysis, only the second step was missing.

The result was JackSignal, a Python class that maps Numpy arrays to Jack ports. It took some research into Numpy's internals and some careful mixing of C and Python code, but in the end the implementation turned out to be straightforward.

JackSignal proved to be a very powerful and practical tool, and it also became clear that the same idea of combining Jack and Python could provide other useful things. The rest is history.

1.1 Overview

The complete Jacktools set contains at the moment more than sixty modules. Many of those implement proprietary algorithms developed in the context of the author's employment by Huawei Research, and unfortunately most of these can't be published.

As already mentioned, those that can be provided fall into two categories. Those intended for audio measurement include:

JackSignal Play and capture signals from/to Numpy arrays. Also provides looping and external triggering.

JackNoise Generates accurate white and pink noise.

JackNmeter Standard filters and detectors for noise measurement.

JackIecfilt IEC class 1 octave band and third octave band filters.

JackPll Phase locked loop, used to track low level drifting signals in noise. Also provides

[1]`https://www.python.org`

[2]`http://www.numpy.org/`

[3]`https://www.scipy.org/`

[4]`https://matplotlib.org/`

I,Q outputs of the phase detector as audio signals.

In the general-purpose category we have:

JackPlayer Multichannel, resampling audio file player. This will play anything that libsndfile can read.

JackGainctl Dezippered multichannel gain control, the DSP part of a fader.

JackParameq Multichannel parametric and 2nd order shelf equaliser.

JackKmeter Multichannel K-meter detector, provides RMS and peak level measurement.

JackMatrix Scalar gain matrix, can be used in many ways, e.g. signal distribution in complex audio installations.

JackMatconv Convolution matrix optimised for dense matrices of short convolutions, as used for microphone and speaker array processing.

JackZconvol General-purpose convolution matrix based on the zita-convolver library.

JackPeaklim Multichannel look-ahead peak limiter, similar to zita-dpl1.

JackAmbpan Up to 4th order Ambisonic panner.

JackAmbrot Up tp 4th order arbitrary axis Ambisonic rotation.

'Multichannel' here usually means 'up to 64 channels'. Also the various matrices can go up to 64×64.
An Ambisonic decoder, networked audio modules (compatible with zita-njbridge) and more dynamics processors are planned to be added.

1.2 Applications

The technical modules have been used to test real-time DSP code, measure speaker directivity patterns, find matched sets of microphone capsules, measure room acoustics, for long term monitoring of environmental noise, and many similar applications. The more general modules have so far been used mainly at the author's workplace, to set up complex demonstrations and listening tests using experimental algorithms. As an example, one listening test involved comparing three different Ambisonics to binaural rendering algorithms, each of them implemented as a Jacktools module, combined with several room simulation methods, again implemented in the same way. This required head motion tracking, so all the rendering had to be done in real time.

For both types of work, having everything under control of an interpreted general-purpose programming language such as Python has significant advantages. It provides not only whatever complex logic may be required, but also access to all system services, external hardware, databases, etc.

For measurements the complete process, including any off-line numerical calculations and up to the generation of a report can be automated. For the listening tests it was an easy exercise to create an ad-hoc graphical user interface providing the participants with exactly the amount of control and feedback required while hiding all the parts that they shouldn't touch or even see.

Other possible applications come to mind, e.g. artistic sound installations, and automated broadcasting systems.

2 Interfacing C++ and Python

All the real-time code for Jacktools is written in C++, so some way to interface this to the Python world is needed.

2.1 High level tools

A wide range of tools for interfacing C or C++ and Python is available. They all have a different scope and use widely diverse methods to achieve their aims.

Boost.Python [5] : 'A C++ library which enables seamless interoperability between C++ and the Python programming language'.

SWIG [6] : (Simplified Wrapper and Interface Generator) provides bindings for many languages, including Python, to C and C++.

Cython [7] : A compiler that extends the Python language and allows simple interfacing with C code.

Of these, **Boost.Python** certainly provides the highest level interface, offering a near transparent gateway between the two worlds, including

[5] `https://www.boost.org/doc/libs/1_61_0/libs/python/doc/html/index.html`
[6] `http://swig.org/`
[7] `http://cython.org/`

even Numpy's arrays (which have some peculiar traits, see below). **SWIG** is considerably simpler and more limited in scope, while **Cython** takes a completely different approach by mixing C-like and Python source code.

While **Boost.Python** would probably be able to do everything required, it would also be overkill for the relative simple functionality needed for Jacktools. In particular, we do not need nor actually want a one-to-one mapping between C++ and Python classes. The Python classes representing the various Jack clients only need to expose the functionality of the real-time code, not its implementation. Also, in Jacktools, the initiative is always at the Python side, the only exception being the C++ code performing a callback to a Python function, but this only happens after that function has been explicitly passed on by the Python code.

On the other hand, neither **SWIG** nor **Cython** seemed to offer much support for handling Numpy arrays nor for working with threads, so this would have to be done manually anyway. Given all this, the most suitable alternative seemed to use Python's C API directly. This had the added benefit of avoiding one more dependency, as well as being a most interesting exercise.

2.2 The Python C API

CPython, the only flavour of Python supported by Jacktools (there are also *Jython, IronPython and PyPy*, written resp. in Java, C# and a subset of CPython) is itself written entirely in C. All the C functions that create, destroy and manipulate Python objects are exported, available for use in extension modules, and quite well documented.

The C API [8], at a first look, seems quite overwhelming and complicated, containing probably thousands of functions. But it is not difficult to use once a few fundamental concepts are understood.

2.2.1 Reference counts

In Python everything is an object, and all objects are reference counted. Once the last reference to an object is deleted, the memory taken by the object can be reclaimed by the *garbage collector* which runs at unpredictable times. Normally all of this happens behind the scenes and automagically, which is one of the reasons why Python is so easy to use, at least regarding memory management.

When using the C API, the programmer has to take care of the reference counts, using the *Py_INCREF()* and *Py_DECREF()* functions. Failure to do this correctly will lead to memory leaks, or worse, stale pointers which will sooner or later trigger a violent crash. The rules are not difficult to understand, unless you want to use things like circular references (which are not used in Jacktools).

2.2.2 Numpy arrays

Numpy arrays are Python objects and are therefore reference counted, but they implement a second level of reference counting internally. This is because the actual data can be shared between array objects. This happens e.g. when an array is *sliced*, which is a very common thing to do in Numpy. For example if given a two-dimensional array **A** containing multichannel audio samples, we can create a vector containing the samples for channel **k** by writing **V** = **A[:,k]**. Numpy will do this without actually copying the data, it just creates a new *view* on **A**. That means that now two Numpy arrays, **A** and **V** are sharing the same data. Numpy arrays are implemented using the Python *buffer interface*, also used by Python for other array-like objects. It is here that the data sharing is implemented. To get access to the buffer of a Numpy array in C you need to call *PyObject_GetBuffer()* and after use the buffer must be released using *PyBuffer_Release()*. These two calls take care of the reference counting.

Since every Numpy array can be just a slice of another, nothing can be assumed regarding the actual placement of data elements in memory. For example, the samples in the vector **V** above may not be in consecutive memory locations. The *buffer interface* provides methods to find out the exact layout of the data elements in a Numpy array.

2.2.3 Threads

Python programs can be multi-threaded, but the interpreter is single- threaded, so only one thread can run at any time. This is implemented using the *Global Interpreter Lock* aka **GIL**. Multithreading in Python is cooperative: the interpreter will release the lock every so many bytecodes, giving other threads the opportunity to grab it and continue.

This has to be taken into account when using the C API, in two ways. First, when the C

[8] `https://docs.python.org/3/c-api/index.html`

code is actually called from Python and wants to call a blocking function, it must release the **GIL** and take it again on return. Second, when calling a Python function from C, the current thread must acquire the lock before doing so and release it when the callback returns. Apart from that, threads created in C can co-exist with Python without any problem.

3 Implementation

On the C++ side there is one base class *Jclient* which contains almost all of the code required to use Jack. It creates the Jack client and the ports, sets the callbacks, obtains the sample rate and period size, etc. It also handles the shutdown callback, and cleans up things when the client terminates. Finally it contains methods to connect or disconnect ports.

Some of the function members of *Jclient* are given a Python interface using the C API: this includes calls to obtain the current process state (more on this later), the Jack period and sample rate, and to manage ports and connections.

Each of the actual tools is a class derived from *Jclient*, implementing the process callback and any DSP code required, including methods to set parameters and obtain results. These members, and only these, are given a Python interface using the C API.

On the Python side there is also a class named *Jclient*, which is the base class for all the others. It provides access to those methods of the corresponding C++ class which have a Python interface. The actual Jacktools classes derive from this base class and again provide access to those methods of their corresponding C++ class that have a Python interface.

So the actual Python classes are defined in Python, and not in the C++ code. The C++ code only implements some methods which are used by the Python classes. It would be possible to define the Python classes directly in C++, but the current method is simpler and requires less code.

3.1 Connecting the two worlds

The remaining question is how the Python objects find their C++ counterpart when any of their methods are called. The mechanism used for this involves a Python class called *PyCapsule* which was first introduced in Python 3 and later backported to Python 2. A *PyCapsule* object is just a container for a C or C++ pointer. It allows Python code to store such pointers and hand them back to the C or C++ side whenever necessary. That is also their only possible use, as there is no way to interpret the contents of a *PyCapsule* on the Python side.

When the user's Python code creates e.g. a *JackGainctl* object, its *__init__()* calls a C++ function that creates the corresponding C++ object and returns two *PyCapsule* objects, one for the newly created C++ object and one for its *Jclient* base object. The Python object stores the first for later, and uses the second to call its base class *__init__()*. This again stores its *PyCapsule* for later use when calling C++ code.

The *PyCapsule* constructor on the C++ side also takes a pointer to a function that will be called when the last reference to the *PyCapsule* is deleted. So when the Python *JackGainctl* is deleted, this function is called and deletes the corresponding C++ object.

3.2 Process states

As explained in the previous section, a C++ object, which will be Jack client, is created whenever a corresponding Python object is created. On the C++ side things can fail e.g. when the Jack server isn't running, or later if Jack 'zombifies' the client. In those cases we still have a Python object, but one that is not usable. To handle this, all Jacktools classes share a common system which simply consists of maintaining a current state. All Jacktools classes have at least the following states:

PASSIVE : the object is an active Jack client, but the *process* callback does not access any ports. This allows to user to manually create ports. At the moment none of the published classes is using this state, all of them will create a fixed set of ports (depending on the number of inputs and outputs requested) and initialise in one of the two following states.

SILENCE : the object is an active Jack client, but the *process* callback outputs silence on all output ports. This state is typically used to further configure a processor that needs this, e.g. a convolution matrix. This is also a safe state to make port connections.

PROCESS : the object is performing its normal function as a Jack client. Some classes e.g. *JackPlayer* have additional active states.

FAILED : the object will enter this state when initialisation or becoming an active Jack client fails.

ZOMBIE : the object will enter this state when zombified by the Jack server.

In the latter two states the only remaining option is to delete the Python object, as it can not recover from these states.

The state system allows complex systems to start up cleanly without making unexpected noises, or at least to fail in a controlled way. It also allows applications that have to run unattended to check things periodically and take some recovery action when anything goes wrong.

3.3 Documentation

All the Python code for Jacktools contains documentation in the form of 'docstrings' which can be read using Python's built-in help system. Also a collection of simple example applications (some of them written for testing the Jacktools classes themselves) is provided.

4 Conclusions

In the previous sections, the Jacktools set of Jack clients implemented as Python classes has been introduced. Some of the implementation aspects and choices have been discussed. It is hoped that this may be of interest not only to potential users, but also to developers of audio software that combines the powers of C, C++ and Python. In particular, in the author's opinion, exploring Python's and Numpy's C API has been a very rewarding exercise.

The Jacktools code package will be made available shortly before the start of the conference.