Proceedings of the 18th Linux Audio Conference (LAC-20), SCRIME, Université de Bordeaux, France, November 25–27, 2020

APPLICATIONS OF JUPYTER NOTEBOOKS FOR AUDIO PLUGIN DEVELOPMENT

Travis Skare
CCRMA
Stanford University, USA
travissk@ccrma.stanford.edu

Jonathan Abel
CCRMA
Stanford University, USA
abel@ccrma.stanford.edu

ABSTRACT

Notebook interfaces in computing, introduced in the late 1980s, are in active modern use by data science and machine learning communities. Related to literate computing, notebooks encourage interleaving expository text with data, code, and figures, making for intuitive presentation of results. During development, they allow for nonlinear or exploratory development, and encourage building on prior research. We consider the application of such notebooks in audio plugin development and analysis, providing short example notebooks covering scenarios in DSP tutorials, white-box testing, black-box testing, and automation of third-party tools. While noting these workflows have been supported by commercial tools for decades, we exclusively use a range of FOSS languages and tools in our samples.

1. INTRODUCTION

We consider the scenario of developing a new phaser effect plugin for digital audio workstations. This involves tasks such as researching what a phaser is, learning how it operates, perhaps exploring the underlying filters we will use, generating some sound samples from competing products, and then moving to write and debug our own plugin.

We demonstrate a set of workflows using Jupyter[1] notebooks to explore accomplishing these tasks with notebook interfaces. Many of these workflows will be immediately familiar to researchers fluent in MATLAB, which offers a similar notebook paradigm, and indeed supports all these use cases with relevant tools and toolboxes. However, in this work we concentrate on open-source tools, libraries, and languages, without loss of generality.

A study of the field of the phaser effect is orthogonal to this work, since it is only an example. For very brief context, the effect is accomplished by applying a chain of time-varying allpass filters with a feed-forward signal path. This results in a pleasant modulation-class effect commonly used on electric guitar, electric piano, synthesizers, and more. We provide sound examples in our first example notebook, and more detail on the history and approaches is available in[2, 3], or in DAFX[4][1].

Our notebooks during development of this hypothetical plugin include:

- Python–Shows combining prose and code to generate an allpass filter, and plotting the filter response.
- Julia[5]–calling C++ instances of an STK[6] biquad class via a foreign function interface and wrapper library. This may be considered a "virtual breadboard" for development; we may edit our native C++ directly and have it driven with test signals by a higher-level language.
- Python–loading an arbitrary LADSPA plugin for which we may or may not have the source, and driving it with a test signal. This can be used for black-box validation of our own plugins or studying third-party effects.
- Faust[7] (via Python and the shell)–driving external tools in the course of implementation of a phaser. Faust is a very powerful domain-specific time-domain language and lets us code up a phaser in a few lines of code–in fact the standard library includes such modulation effects as primitives! Because the language does not yet integrate with Jupyter notebooks directly via a kernel or similar, we demonstrate shelling out to the Faust tools to have a notebook act as a build automation or report generation tool

These notebooks have been uploaded as part of this work; readers may wish to browse them after scanning this paper. However the descriptions in the paper are intended to stand on their own, and a screenshot of the first notebook is provided. The direct URL at the time of writing is https://github.com/tskare/lac2020demo; a redirector has been set up at at https://bit.ly/2TqcQuG in case this changes in the future, which is not expected.

We note some source repository browsers have notebook-viewing facilities for `.ipynb` and similar formats, which is convenient; we use GitHub for this work to demonstrate. In case the viewer does not support audio widgets, `phaserdemo.mp3` is provided as a standalone file for this case.

A side note on machine learning: Machine Learning and Deep Learning are very popular topics across many domains of research. Such notebooks are a common workflow in ML and Data Science; many getting started guides use them. Because they are so prolific, we intentionally avoid discussing ML workflows in this work and aim to stay within the digital audio effect development domain.

Finally, to aid in conveying motivation and use case, a demonstration video developed for the conference presentation will be provided/linked with the repository.

1.1. Installation

Readers may follow along by installing the following software:

[1] Section 2.4.2 in the Second Edition

Python: likely already installed on your system. We use Python 3 for this work; noting that Python 2 has been officially sunset as of January 1, 2020. We would suggest that if you do not have Python installed already, consult your system administrator or package manager, as this may affect your system in a wide manner.

Conda (*Optional)*: This work was developed using Conda, a package manager that is supported on Linux, MacOS, and Windows, and allows switching between different environments for different projects. https://docs.conda.io/en/latest/

Jupyter via `conda` or `pip`: https://jupyter.org/install

Julia via Conda, your package manager of choice, or from their homepage at https://julia-lang.org. The Julia Project homepage may offer the most up-to-date version; we updated to 1.3.1 before paper submission.

STK, the Synthesis Toolkit in C++[6]. This is for following along with the second example. A mirror is available at https://github.com/thestk/stk

Julia's **CxxWrap** package via the built-in package manager (press right-bracket, `]`, in the Julia REPL and enter "`add CxxWrap`"). This is for easily wrapping C++ classes. This is only one of a handful of methods for wrapping or calling C/C++ code. This seemed to work better than the built-in libraries when running inside Jupyter, but readers are encouraged to evaluate the others for their use case.

Faust via some package managers, or the Faust Homepage at https://faust.grame.fr/.

Next, we present the four use cases.

2. USE CASE: EDUCATION (PYTHON)

In this section we explore a standard use of notebooks, presentation of interleaved prose, code, and results. We note that this use of notebook interfaces is common in other domains.

Here we explain a simple digital phaser effect. In the opening paragraphs, we list some commercial phaser effects from MXR, Electro-Harmonix, and Eventide, and fetch a Creative Commons image of an MXR Phase 90 pedal from the web (local filesystem works as well and is better for posterity).

We provide an inline audio example of the phaser, so the reader may immediately understand what our desired end result may sound like.

A screenshot of the second half of this notebook is presented in Figure 1 [after the main paper text and bibliography]. Note we interleave explanatory text, an equation, Python numerics code, and filter response plots.

There, we explain the digital allpass filter that will be a building block for our implementation. The introduction links to resources we cite here, to guide readers to deeper study. LaTeX-style equations are rendered in the Markdown prose via MathJax.

Finally, we use `SciPy`'s `freqz` implementation to obtain the frequency and phase response of the first-order digital allpass filter. We leverage the example code from the `freqz` documentation to plot the frequency and phase responses inline in the notebook.

Readers who wish to dive deeper may download the notebook and experiment. For instance, they may wish to alter the allpass parameters g_i, or extend the notebook from an introductory level to a deep-dive level by adding text discussing virtual analog considerations.

While again we emphasize this exploration is a standard use of data science notebooks, rather than a novel work, we call out the benefits of interleaving product images, underlying equations, study of the building blocks, implementation, and sound example in a single browser window. Beyond display in a browser, JupyterLab also allows exporting notebooks with code, data, results, and commentary all "baked in" to slide-style presentations, HTML, lecture-note-style PDFs, or LaTeXwhich could be integrated directly into an academic paper.

Tools also exist to host live versions of the notebook, or have multiple researchers working in the same session; these are outside the scope of this paper.

3. USE CASE: WHITE-BOX PRODUCTION C++ ANALYSIS (JULIA)

Next, we consider the case of using notebooks to provide a "report" on production C++ code.

A variety of workflows for plugin development exist. Anecdotally, we hear it is common to prototype in a high-level language such as MATLAB before porting algorithms to optimized code, usually in C++. In recent years, Mathworks has even introduced compilation direct to plugins to facilitate prototyping and experimentation directly in DAWs.

In this section we propose use of notebooks to call C++ code in development. The hypothetical code under test is considered "white box;" that is, in this section our imaginary company has developed both the notebook and the plugin code. We may be porting C++ from a Matlab prototype, and wish to make sure inputs and outputs match, or we may be building our plugin from scratch and would benefit from a test bench that drives the plugin and obtains various plots, inputs, and outputs for analysis, or sharing with our development team.

The Julia language is used without loss of generality; we note that in the next section we will call C++ from Python for a different application. While outside the scope of this paper, interested readers might consider the `cffi` module (C Foreign Function interface) in Julia, or CPython extension capabilities. Finally, C++ interpreter kernels exist for notebook computing and we could write our plugin code directly in the notebook.

Development of this notebook is fairly straightforward. We imagine a use case is that we are debugging the Biquad filter present in the Synthesis Toolkit (STK)–this may be found in `src/BiQuad.cpp` in the STK repository. Julia supports multiple ways of calling C/C++ code, including a built-in `ccall`[2], designed to be a low-overhead, no-glue method of calling C and Fortran numerics libraries. Other methods such as `Cxx` and `CxxWrap` packages may be added from the built-in package manager. We use the latter, `CxxWrap`, currently available via GitHub[3], and installable via the built-in package manager as discussed in Section1.1.

[2]https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/

[3]https://github.com/JuliaInterop/CxxWrap.jl

3.1. C++ Work Required: Wrapping the Class

The CxxWrap approach requires some prep work outside of the notebook, but this is straightforward. We write some standardized glue code then use CMake to build a `.so` shared library.

We must define a function to expose our method as a Julia module. This is as follows; not all methods are included for brevity.

```
#include "jlcxx/jlcxx.hpp"
JLCXX_MODULE define_module_biquad(
    jlcxx::Module& mod)
{
  mod.add_type<stk::BiQuad>("STKBiQuad")
    .constructor<>()
    .method("setCoefficients", &stk::BiQuad::
        setCoefficients)
    .method("sampleRateChanged", &stk::BiQuad
        ::sampleRateChanged)
    .method("setB0", &stk::BiQuad::setB0)
    .method("setB1", &stk::BiQuad::setB1)
    .method("setB2", &stk::BiQuad::setB2)
    .method("setA1", &stk::BiQuad::setA1)
    .method("setA2", &stk::BiQuad::setA2)
    .method("tickOne",&stk::BiQuad::tickOne);
}
```

Whereas most functions like `setCoefficients` are native STK functions, `tickOne` was added to work around specifying an overloaded function in the call to `add_type`. STK's `tick` has several variants and this was currently the easiest way we found to disambiguate between them. Also, this way we do not depend on familiarity with the `StkFrames` class and only deal with primitive types in the notebook. We provide our modified source in `BiQuadJulia.cpp` and associated CMake file; the only other addition for the sake of completion is a `tickOne` implementation. A minimal one:

```
float BiQuad::tickOne(float in) {
  StkFrames frames(1, 1);
  frames[0] = in;
  StkFrames framesout = tick(frames);
  return framesout[0];
}
```

On the module side, loading the library begins with the minimal:

```
module STKBiquad
  using CxxWrap
  @wrapmodule("/home/$USER/src/third_party/
      stk/src/lib/libbiquadtestlib", :
      define_module_biquad)

  function __init__()
    @initcxx
  end
end
```

Now we may drive and plot our C++ function in Julia.

Next, we explore loading shared libraries from another language, Python:

4. USE CASE: BLACK-BOX BINARY PLUGIN ANALYSIS (PYTHON)

We may wish to drive and analyze a plugin on a "virtual lab bench." Perhaps we wish to black-box test our build artifacts to validate with a test suite, or perhaps we wish to script an analysis of which third-party saturation plugins alias when running at 44.1kHz, for example.

In this section we load and drive a simple LADSPA plugin. A simple v1 plugin is launched directly as a standard library; we note a complete production workflow would instantiate and call LV2 plugins through the Liiv library. We note the existence of Python-LADSPA projects on GitHub; we did not evaluate these so that our notebook requires no dependencies beyond the built-in `ctypes`.

Here, we enumerate available LADSPA plugins from the commandline (via the `listplugins` program included with the SDK) and then in our notebook, use the `ctypes` module to load any of those plugins. We declare the LADSPA interface to `ctypes` in terms of relevant structures and functions, then may load the library and create a plugin in memory. LADSPA is fairly unique in that the plugin libraries expose only one function, which retrieves a reference to the Nth plugin in the library. The reference structure in turn contains function pointers which allow connecting control and sound buffers, reading metadata, and processing audio data.

Because a LADSPA wrapper may be more immediately useful than our notebooks, we include the wrapper code directly in Listing 1. Users may load and call into a plugin with Python code such as:

```
plugHandle = 0
# Load the second plugin in a shared library.
plugPtr = loadPlugin(
  '/myhome/dev/testplugin.so', 1)
plugInst = plugPtr[0] # dereference pointer
print("Plugin: %s by: %s, (c) %s" % (
    plugInst.Name,
    plugInst.Maker,
    plugInst.Copyright))
print('ports:')
for i in range(plugInst.PortCount):
    print("%s - %s" % (
      plugInst.PortNames[i],
      plugInst.PortDescriptors[i]))
```

As we provided type information to `ctypes`, runtime type checking is performed. LADSPA typedefs were included in the wrapper to help avoid type confusion and increase readability.

5. USE CASE: CALLING EXTERNAL TOOLS (FAUST)

A final, fourth notebook considers the case where we would like to use the exploratory, cross-media notebook paradigm but have existing tools and do not want to use C foreign function tools or write a new notebook kernel.

As a concrete case, consider that we wish to report on the architecture and results of a Faust plugin in development.

We do note the existence of `faust_python`[4] from 2015

[4] https://github.com/marcecj/faust_python

and a wrapper for Julia widgets that leverages this, currently in development over the last months[5].

Development of this notebook is perhaps the most straightforward. A notebook cell that begins with the exclamation point operator will execute that command in the shell.

```
!echo hi world
```

will output "hi world", for example. We can use this functionality to display a `.dsp` file in development, call `faust` to compile it, invoke `faust2svg` to generate the system diagram, and display that artifact with IPython's native SVG rendering support in the notebook.

This may be seen as build automation, though extending things a bit further, it could be used to combine algorithm descriptions, relevant Faust code, plots of system response, and generated audio demos. There are opportunities for significant further work here, as described in the next section. On its own, this style of notebook can demonstrate that processes spanning multiple tools may be combined and automated in place of a Makefile or script. We can glue together existing workflows quickly, and spend more time on exploration and development of our hypothetical plugin–a common goal among all these processes.

6. FURTHER WORK

The "virtual test bench" that runs LADSPA plugins would ideally be extended to use LV2 and/or VST, as development has moved to those platforms for Linux (for MacOS, AudioUnit is worth considering).

There are many opportunities for extending the Faust notebook. As mentioned, there are some open-source libraries in the field for loading plugins or wrapping Faust's compilation functionality with the Python foreign function interface. This could be investigated, toward having the full Faust development workflow available to a notebook. We could also call the excellent Faust web-based tools and compiler as an API, or have those system-local, to be able to develop, build, and test actual plugin binaries within one notebook. Especially once inline coding opportunities are added, this could be the framework for a set of interactive articles on introductory effects plugin signal processing.

7. CONCLUSIONS

We suggested the use of notebook workflows, popular in data science and machine learning communities, for subtasks involved in plugin development. Both Python and Julia were used at different times, and we shelled out to Faust to demonstrate driving tools not yet integrated in the notebook ecosystem. Markdown provides prose and equation support for all notebooks.

As a secondary tangible result, we provide generic wrapper code for loading LADSPA v1 plugins in Python.

8. ACKNOWLEDGMENTS

Thanks to the anonymous reviewers and conference organizers, especially for this unique year. And of course to the authors and contributors to all the FOSS software mentioned in these workflows. They combine to make audio development an exciting area for research, education, and hobby development.

[5] https://github.com/hrtlacek/faustWidgets

9. REFERENCES

[1] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[2] Julius O. Smith III, *Physical audio signal processing: For virtual musical instruments and audio effects*, W3K publishing, 2010.

[3] Julius O. Smith III, "An allpass approach to digital phasing and flanging," in *ICMC*, 1984.

[4] Udo Zölzer, *DAFX: digital audio effects*, John Wiley & Sons, 2011.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[6] Perry R Cook and Gary P Scavone, "The synthesis toolkit (STK).," in *ICMC*, 1999.

[7] Yann Orlarey, Dominique Fober, and Stéphane Letz, "Faust: an efficient functional approach to dsp programming," 2009.

Allpass filters

The phaser effect is obtained by combining an even number of allpass filters with a feedforward gain path. The allpass filters are formed as follows with time-varying parameter $g_i(n)$:

$$\text{Allpass}_i = \frac{g_i + z^{-1}}{1 + g_i z^{-1}}$$

Then we can plot frequency and phase response via `SciPy` and plot via `Matplotlib`:

[38]:
```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
g = 0.5
w, h = signal.freqz(b=[g, 1], a=[1, g], worN=1000)
```

[39]:
```
fig = plt.figure()
plt.title('Digital allpass frequency response')
ax1 = fig.add_subplot(111, label='freqresp')
plt.plot(w, 20 * np.log10(abs(h)), 'b')
plt.ylabel('Amplitude [dB]', color='b')
plt.ylim([-1,1])
plt.xlabel('Frequency [rad/sample]')
ax2 = ax1.twinx()
angles = np.unwrap(np.angle(h))
plt.plot(w, angles, 'g')
plt.ylabel('Angle (radians)', color='g')
plt.grid()
plt.axis('tight')
plt.show()
```

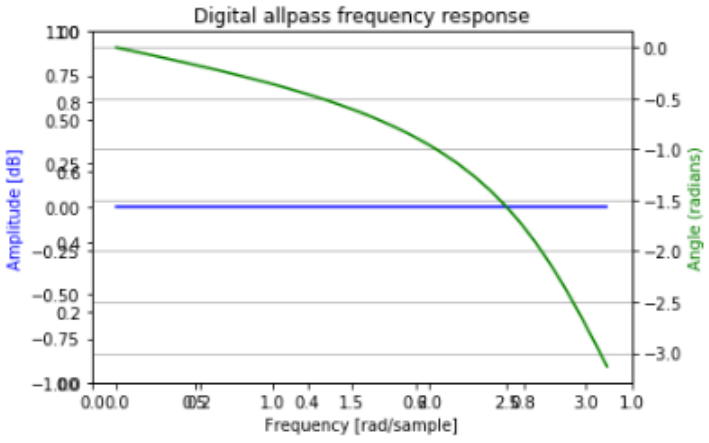

Figure 1: *A screen capture of the second half of Notebook 1, described in Sec. 2*

```
import ctypes

# Declare interfaces to the structures and functions we'll call.

# typedefs and constants
LADSPA_Data = ctypes.c_float
LADSPA_Properties = ctypes.c_int
LADSPA_Handle = ctypes.c_void_p

LADSPA_PortDescriptor = ctypes.c_int
kLADSPA_PORT_INPUT   = 0x1
kLADSPA_PORT_OUTPUT = 0x2

LADSPA_PortRangeHintDescriptor = ctypes.c_int;
# hint constants omitted so this fits on one page; please reference the .h file.

class LADSPA_PortRangeHint(ctypes.Structure):
    pass
LADSPA_PortRangeHint._fields = [
    ("HintDescriptor", LADSPA_PortRangeHintDescriptor),
    ("LowerBound", LADSPA_Data),
    ("UpperBound", LADSPA_Data)
]

class LADSPA_Descriptor(ctypes.Structure):
    pass
LADSPA_Descriptor._fields_= [
    ("UniqueID", ctypes.c_long),
    ("Label", ctypes.c_char_p),
    ("Properties", LADSPA_Properties),
    ("Name", ctypes.c_char_p),
    ("Maker", ctypes.c_char_p),
    ("Copyright", ctypes.c_char_p),
    ("PortCount", ctypes.c_ulong),
    ("PortDescriptors", ctypes.POINTER(LADSPA_PortDescriptor)),
    ("PortNames", ctypes.POINTER(ctypes.c_char_p)),
    ("PortRangeHints", ctypes.POINTER(LADSPA_PortRangeHint)),
    ("ImplementationData", ctypes.c_void_p),

    # Interface is via function pointers in the struct.
    ("instantiate", ctypes.CFUNCTYPE(LADSPA_Handle, ctypes.POINTER(LADSPA_Descriptor),
        ctypes.c_ulong)),
    ("connect_port", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle, ctypes.c_ulong)),
    ("activate", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle)),
    ("run", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle)),
    ("run_adding", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle, ctypes.c_ulong)),
    ("run_adding_gain", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle, LADSPA_Data)),
    ("deactivate", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle)),
    ("cleanup", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle))
]

# The actual library has only one function.
# The argument, |index|, can choose one of N plugins in the library.
# Indices beyond that range are NULL.
def loadPlugin(name = '/usr/lib/ladspa/delay.so', index=0):
    plugin = ctypes.CDLL(name)
    plugin.ladspa_descriptor.argtypes = [ctypes.c_ulong]
    plugin.ladspa_descriptor.restype = ctypes.POINTER(LADSPA_Descriptor)
    return plugin.ladspa_descriptor(index)
```

Listing 1: *Code to define the LADSPA interface in Python via* `ctypes`.