DawDreamer: Bridging the Gap Between Digital Audio Workstations and Python Interfaces

David Braun (braun@ccrma.stanford.edu)

Center for Computer Research in Music and Acoustics, Department of Music, Stanford University

ABSTRACT

Audio production techniques which previously only existed in GUI-constrained digital audio workstations, live-coding environments, or C++ APIs are now accessible with our new Python module called DawDreamer. DawDreamer therefore bridges the gap between real sound engineers and coders imitating them with offline batch-processing. Like contemporary modules in this domain, DawDreamer can create directed acyclic graphs of audio processors such as VSTs which generate or manipulate audio streams. DawDreamer can also dynamically compile and execute code from Faust, a powerful signal processing language which can be deployed to many platforms and microcontrollers. Our paper discusses DawDreamer's unique features in detail and potential applications across music information retrieval including source separation, transcription, parameter inference, and more. We provide fully cross-platform PyPi installers, a Linux Dockerfile, and an example Jupyter notebook for making tempo-matched audio mashups.

Homepage: https://github.com/DBraun/DawDreamer

INTRODUCTION

A digital audio workstation (DAW) is a software system which integrates most music production tasks including composing, recording, editing, adjusting effects, and exporting to audio files. An audio engineer typically uses a mouse and keyboard or expensive mixing console to carry out these tasks, making it difficult to explore efficiently the large action space of effects and their parameters. Moreover, some digital instruments and effects are platform specific, such as Audio Units on macOS or LV2 plug-ins on Linux. The ideal batch-processing audio framework with relevance to machine learning should both overcome the hurdles of mouse-and-keyboard interfaces and unify instruments and effects across all platforms.

One project in this domain is RenderMan, a Python module which served as the starting codebase for DawDreamer. RenderMan uses the JUCE framework for rendering audio from VST instruments. RenderMan played a crucial role in research on software synthesizer presets and massive audio generation, but its development has been slow to branch into other aspects of music production such as bussing.

FluidSynth is a sample-based synthesizer engine with command-line support, but its reliance on SoundFount samples limits broader applications.

Pedalboard is a new project with similarities to RenderMan and DawDreamer. It has a promising future but currently lacks support for Faust, parameter automation, efficient time-stretching and pitch-bending, and audio processor graph building (generalized bussing).

FEATURES

DawDreamer aims to address the limitations of other tools and expand the capabilities of Python interfaces which emulate DAWs by combining these key features:

- Composing graphs of audio processors
- Audio playback
- VST instruments
- VST effects (including those with multiple inputs)
- Faust effects and polyphonic instruments
- Time-stretching and looping according to Ableton Live warp markers (without writing to file system)
- Pitch-warping (without writing to file system)
- Parameter automation
- Rendering multiple processors simultaneously
- Full support on macOS, Windows, Linux, Ubuntu, Dockerfile
- Continuous integration testing on GitHub

Why Faust?

Faust (**F**unctional **AU**dio **ST**ream) is a programming language for real time signal processing. Faust's built-in libraries include functions for reverbs, compressors, oscillators, filters, ambisonics, Yamaha DX7 emulation, and more. Visit https://faustlibraries.grame.fr for more examples.

DawDreamer uses the **libfaust** backend to compile Faust code just-in-time. Elements in the Faust source code that would usually designate user interfaces such as sliders or toggles instead become parameters which can be automated according to **numpy** arrays.

This coupling between Faust user interfaces and DawDreamer also enables easy control of polyphonic Faust instruments. A developer can write Faust code with a single voice of polyphony in mind and provide MIDI notes from Python or from a MIDI file. All of the voice allocation is done automatically.

DawDreamer includes great starting-point Faust examples:
- Faust Library's Yamaha DX7 recreation
- A sidechain compressor
- A polyphonic wavetable synthesizer
- A polyphonic sampler instrument (like a drum machine)

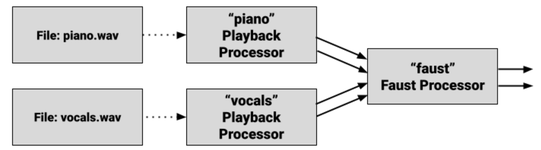
The synthesizer's wavetable and the sampler's sample can be specified with **numpy** arrays. If a VST has limits on which audio files it can use, or some parameters aren't accessible to the user, you should consider using Faust to recreate the instrument. The sampler example shows the simplicity of using MIDI-triggered ADSR envelopes and information to modulate the sample's pitch, volume, and filter cutoff.

Beyond DawDreamer, Faust code can be compiled for Windows, Linux, macOS, Android, iOS, and many microcontrollers such as Teensy, SHARC, Bela, and most recently FPGAs. It can also be exported in many project formats and languages such as JUCE, Max, vcvrack, rust, julia, soul, C, C++, and more. Researchers would be wise to not restrict themselves to VST and LV2 audio plug-ins when Faust can be deployed so widely.

CODE WALKTHROUGH

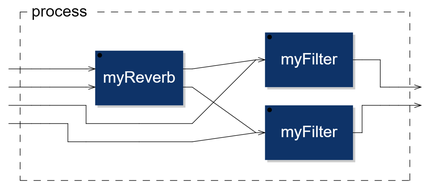
Let's load two stereo tracks (piano and vocals) from WAV files. The piano will go through a reverb effect and then be added with the vocals. The result will go through a stereo low-pass filter whose cutoff frequency changes according to **numpy** data. DawDreamer's Playback Processor will play the audio files, and a Faust Processor will do the reverb, mixing, and low-pass filter.

Overall signal flow diagram from DawDreamer's perspective, resulting in a stereo output.



The Faust Processor will take two stereo inputs and produce one stereo output.

This diagram was made by entering Faust code in the online Integrated Development Environment: https://faustide.grame.fr



Next, let's examine the Faust code (saved as "my_dsp.dsp") that produced the diagram above.

```
declare name "MyEffect";
import("stdfaust.lib");

cutoff = hslider("cutoff",  15000.,  20,  20000,  .01) : si.smoo;
myFilter= fi.lowpass(10, cutoff);

myReverb = _, _ : dm.zita_rev1 : _, _;

process = myReverb, _, _ :> myFilter, myFilter;
```

The code defined a UI parameter named "cutoff," which we will automate with **numpy**. Finally, let's look at the Python code which uses DawDreamer to process and save the audio.

```
SR, BLOCK_SIZE, DURATION = 44100, 1, 10.
# Create an engine with a sample rate and block size.
engine = dawdreamer.RenderEngine(SR, BLOCK_SIZE)

# Load audio files into 2D numpy arrays
piano = get_audio("piano.wav")
vocals = get_audio("vocals.wav")

# Create 2 audio processors for playback and one for Faust.
playback_1 = engine.make_playback_processor("piano", piano)
playback_2 = engine.make_playback_processor("vocals", vocals)
faust_processor = engine.make_faust_processor("faust")

faust_processor.set_dsp("C:/path/to/my_dsp.dsp")
# Make a sine wave of 2 Hz, and remap the output values.
automation = 15000+5000*make_sine(2., DURATION)
faust_processor.set_automation("/MyEffect/cutoff", automation)

graph = [
  (playback_1, []),
  (playback_2, []),
  (faust_processor, ["piano", "vocals"])
]

engine.load_graph(graph)
engine.render(DURATION)
audio = engine.get_audio().transpose()
scipy.io.wavfile.write('my_song.wav', SR, audio)
# Modify parameters of the processors and render again...
```

POTENTIAL USE CASES

Music Information Retrieval

A researcher of universal music source separation could use DawDreamer and generative music composition networks to create ground truth mixtures of tens of audio tracks rather than the common four (vocals, drums, bass, and other). These generated mixtures could become increasingly realistic and helpful for source separation, transcription, lyrics alignment, instrument identification, cover identification, and more.

Intelligent Music Production

DeepAFX achieved high quality automatic audio mastering through gradient approximation of a fixed series of LV2 audio effects. DeepAFx also succeeded at picking plug-in parameters to match a guitar pedal's distortion. In both cases, DawDreamer could learn the same mastering or compressor with Faust effects, but thanks to Faust, the effect could be deployed easily to more microcontrollers.

CONCLUSION

Much of music production is a series of actions taken inside a DAW environment, yet some ML researchers study musical audio as a raw series of numbers. To be fair, this domain-agnosticism helps models generalize to other domains, but it forfeits the helpful inductive biases from understanding music as the interaction of MIDI notes, sample packs, signal chains, effects, and parameter settings. Those building blocks and domain knowledge form a large part of the DNA of music. Researchers can now use DawDreamer as the physically unconstrained software engine that grows musical DNA into fully-realized audio data.

ACKNOWLEDGEMENTS

The author thanks the following people:
- Leon Fedden for starting RenderMan and making it open-source
- Julius O. Smith III and Stéphane Letz for their support with Faust
- Christian Steinmetz and Chris Donahue for feedback on manuscripts

Stanford University

CCRMA