pynth: Intuitive Sound Synthesis in Python

Matej Bevec
Email: matejbevec98@gmail.com

***Abstract*—Many existing sound synthesis libraries suffer from a burdensome API, which slows down the prototyping process and may discourage inexperienced users. In this report, we introduce a new python library — *pynth*, which utilizes various python's advantageous features to create an extensible sound synthesis framework with a minimal and intuitive interface. We briefly describe the library architecture, its main classes and provide usage examples. We provide an evaluation in terms of performance and discuss possible limitations.**

I. Introduction

Programmatic sound synthesis refers to creating sounds, soundscapes or music using a programming interface, usually in the form of a specialized high-level library that allows a user to combine simple elements to create complex sounds. This has many use cases, such as procedurally generating sounds for software and videogames, creating plugins to aid in music production, building signal-processing pipelines or even creating interactive live performances.

Numerous toolkits for programmatic sound synthesis are available, but many, such as the java library JSyn, suffer from a complicated API with burdensome syntax and require substantial boilerplate code. This slows down the prototyping proccess and may even discourage lesser experienced potential users without significant programming expertise.

According to PYPL, Python is currently the world's most popular programming language. This is in part due to its intuitive and minimal user interface, which allows programmers to translate their ideas to working code with minimal friction (or in other words, to "think in code"). As such, it has become the go-to choice for prototyping and experimental code for many users. Additionally, python is approachable to lesser-experienced users and has been popular in communities that only intersect computer science.

Following this philosophy, the proposed library utilizes poweful pythonic concepts, such as custom operation definitions (*magic functions*) and vector arithmetic (*numpy*), to provide a significantly more convenient and approachable user interface. *Pynth* adopts some similar principles to machine learning auto-differentiation libraries such as *PyTorch*. As a user constructs an expression with a functional-like syntax, a computation graph is constructed in the background. When needed, the graph is evaluated to produce the result from given inputs.

II. Related work

JSyn [2] is a sound synthesis library for the Java ecosystem. It is based on the unit generator (modular) paradigm and is, in terms of architecture, a loose model for our library. Similarly, it features various oscillators, arithmetic operations and filters. A rough C++ equivalent is the Synthesis Toolkit (STK) [1], which has been available since 1995. Both of this libraries, aside from not being in the python ecosystem, suffer from the issues described above.

III. Architecture

Pynth uses a traditional unit generator model. The basic building block of a *pynth* synthesizer is a **Module**. A module is connected to other modules via its inputs and its output. Together, modules form the computation graph.

Waves are the inputs to the sistem and generate digital signals. The signal can either be a (discretized) continuous function such as **Sin**, a sampled sequence (e.g. audio recording) or a constant value. Other modules have inputs. They transform the input signals and output the results.

The signal "flowing" through the computation graph is temporally segmented into segments, which we refer to as **chunks** — fixed length numpy arrays of samples. When a single node (module) is evaluated, it collects the required data (chunks) from its input modules and computes an output.

Before the computation starts, all the nodes are topologically sorted. Then at each time step, one chunk is evaluted for every node, in topological order. This ensures that, for each node, its input data is ready before it is asked to produce an output.

Note that cycles (i.e. feedback loops), such as the one in Figure 4, necessarily break the topological order. In this case, we make the decision to evaluate all nodes in a loop once every time step, and threat possible uncomputed inputs as inputs from the previous time step or as a zero-valued signal, if the former are not available.

Depending on the application, different chunk lengths might be optimal. A live configuration will require a stream of short chunks, while a large chunk size might suffice for other applications. The chunk size tradeoff is described in Section VI.

IV. Modules

All modules in pynth subclass the **Module** class. Additionally, most can be classified into one of three categories. **Wave** modules generate signals without an input (except a possible trigger control). **TwoOp** modules are arithmetic or logic operations between two input signals. Finally, **Filter** modules apply an LTI filter to an input signal and may be modulated by control signals.

TABLE I
CURRENTLY AVAILABLE MODULE CLASSES IN *pynth*, THEIR TYPES (SUPERCLASSES) AND BRIEF DESCRIPTIONS.

Module	Type	Description
Module		Base class.
Wave	Module	A wave generator. Sample, constant or function.
Envelope	Wave	An AD envelope generator.
Input	Wave	Generate constant value with function call.
Ramp	Wave	Interpolate between values over time.
Pulse	Wave	A single pulse of any length.
Pulses	Wave	Periodic pulses with given period and width.
Sin	Wave	A sine oscillator.
Triangle	Wave	A triangular oscillator.
Saw	Wave	A sawtooth oscillator.
Square	Wave	A square oscillator.
WhiteNoise	Wave	A white noise generator.
TwoOp	Module	Base class for arithmetic and logic operations.
Add (+)	TwoOp	Signal addition.
Sub (-)	TwoOp	Signal subtraction.
Mul (*)	TwoOp	Signal multiplication.
Div (/)	TwoOp	Signal division.
Pow (**)	TwoOp	Signal potentiation.
Compare (<, >)	TwoOp	Output 1 if A > B or vice versa.
Filter	Module	Base class for LTI digital filters.
MovingAvg	Filter	Moving-average lowpass filter.
Butter	Filter	Butterworth filter with provided parameters.
EQ	Filter	Window-function-based configurable EQ.
Lowpass	Filter	Lowpass with modulated cutoff and resonance.
Highpass	Filter	Highpass with modulated cutoff and resonance.
Bipol	Module	Convert unipolar (0, 1) signal to bipolar (-1, 1)
Unipol	Module	Convert bipolar (-1, 1) signal to unipolar (0, 1)
Delay (»)	Module	Delay input signal by given amount.
Sequencer	Module	Advance in sequence at every input front.
Scope	Module	Record and display input signal at playtime.

V. USAGE

A key feature of the proposed library is a minimal and "functional" interface. Many configurations that would require and order of magnitude more code in a library like JSyn take up only a few lines. First, one or more Waves (inputs) need to be instanciated. Other modules can then be seen as operations (functions) on these inputs. Note that standard python operations, like $+$ and $*$, can be used to instantiate certain modules. The result can be resolved to a sample array or played by calling $eval$ or $play$ on the output node. Pynth also provides a utility to visualize the constructed computation graph and to visualize the produced sound.

Playback can be performed in *offline* mode, where the entire output is evaluated (sampled) and then played or in *live* mode, when each chunk is evalued and played in real time. This mode supports user input. A special module, **Scope**, is also provided to visualize arbitrary outputs in real time (see Figure 5).

The following are some example implementations.

a) Overtones (Fig. 1): The following example implements the addition of 5 overtones to a fundemental sine wave. Note that some operators (+, -, *, /, <, >, ») can be used to create modules. This makes constructing computational graphs fast and intuitive.

```
from pynth import *

out = 0.5 * Sin(200)
for i in range(5):
    out += 0.1 * Sin(100*i)

drawgraph(out)
showsound(out, t2=0.1)
out.play(5*SR)
```

b) Vocal remover (Fig. 2): The following example loads a pop track with centrally-mixed vocals and removes the vocals by subtracting the left and right channel and adjusting for lost low frequencies.

```
sample, sr = librosa.load("file.wav", mono=False)
l = Wave(sample[0, :])
r = Wave(sample[1, :])
SR = sr

out = 0.5 * (0.5*l - 0.5*r)
out += 0.5 * MovingAvg(0.5*l + 0.5*r, M=50)
```

c) Wind (Fig. 3): The following is a basic implementation of a procedural wind generator. White noise is fed through a lowpass filter. The filter's cutoff frequency and resonance are both modulated by sums of low-freqency sine waves to produce pseudo-random patterns.

```
noise = WhiteNoise()
cutmod = Unipol(.5*Sin(1/3) + .5*Sin(1/5)) * 0.1
resmod = Unipol((.5*Sin(1) + .5*Sin(.8)) >> 0.3)
resmod = resmod * 0.4 + 0.1
out = Scope(Lowpass(noise, cutmod, resmod))
```

d) Karplus-Strong (Fig. 4): The following is an implementation of Karplus-Strong string pluck emulator, where white noise is delayed and summed with the original signal in a feedback loop. Note that the output can be "frozen" (i.e. recorded to an array) to be reused as a sample.

```
noise = Wave(np.random.rand(4000)-0.5)
delay = Delay(None, delay=0.005)
add = noise + delay
delay.ins["a"] = 0.95 * MovingAvg(add, M=10)
out = add

frozen = Wave(add.eval(2*SR))
```

e) User input, envelopes and scopes (Fig. 5): The following is a simple demonstration of pynth's live operation with user input. Whenever the L key is pressed, an enveloped saw wave is triggered. Note that the signal can be passed through special **Scope** modules, which produce live visualizations upon playback.

```
import keyboard

control = Input()
env = Envelope(Scope(control), (2000, 0, 0, 8000))
out = Scope(env)
out = Scope(out * Saw(100))

val = 0
def loop(t):
    if keyboard.is_pressed('l'):
        control.set(1)
    else:
        control.set(0)

out.play(30*SR, live=True, callback=loop)
```

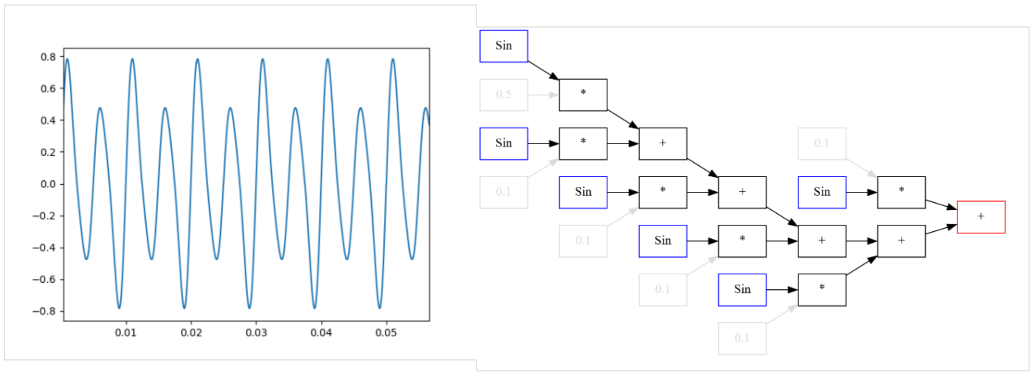

Fig. 1. Summing sine overtones in *pynth*. The left plot depics the output signal over time (in seconds). The computation graph is depicted on the right. The upcoming figures follow the same layout.

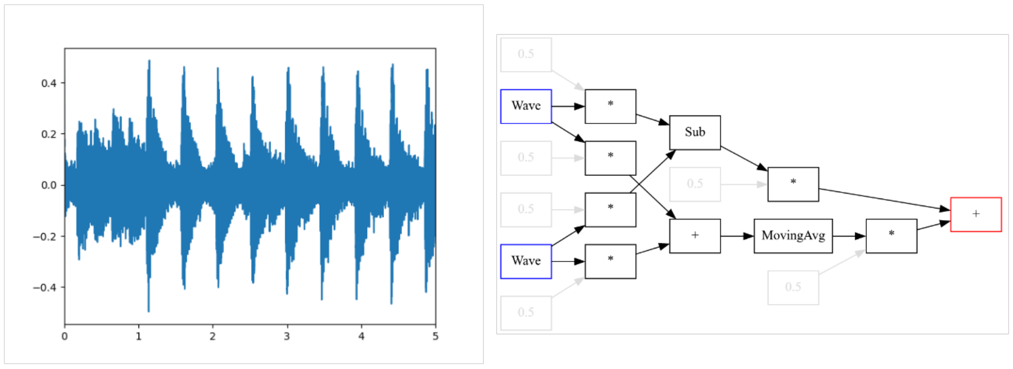

Fig. 2. Removing centrally-mixed vocals in *pynth*.

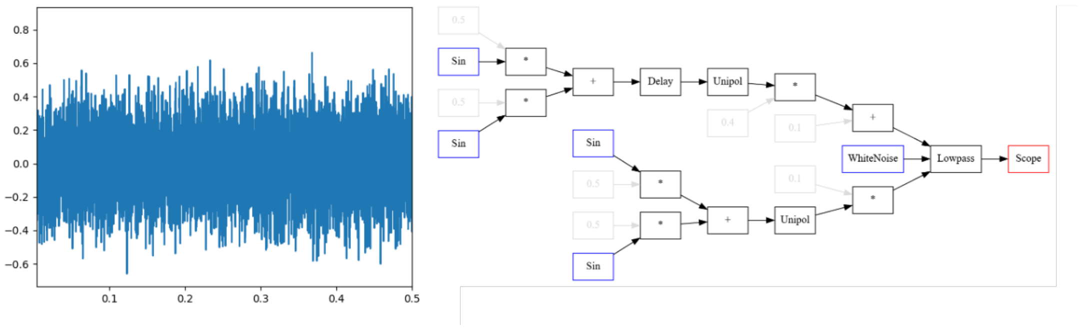

Fig. 3. Procedural wind effect in *pynth*.

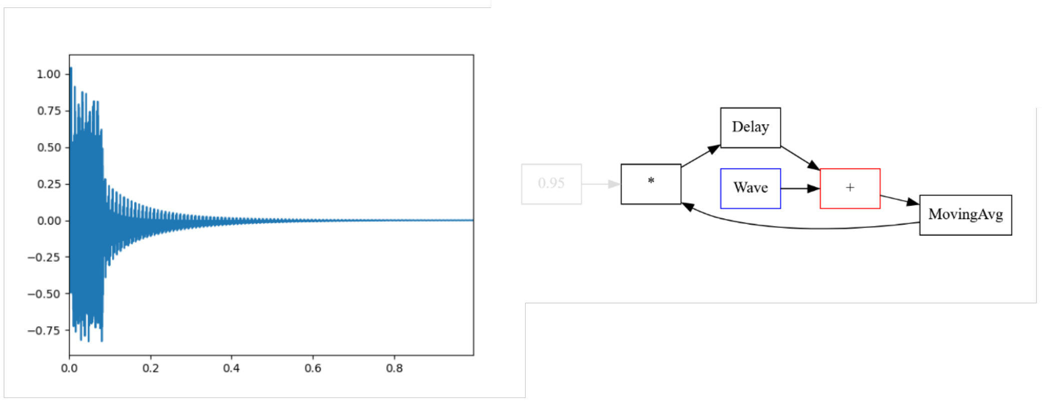

Fig. 4. Karplus-Strong string emulator in *pynth*.

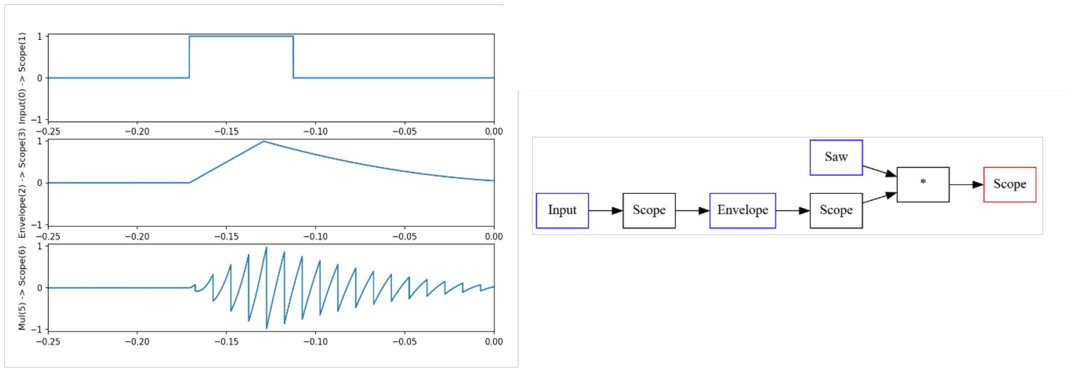

Fig. 5. User-triggered envelopes in *pynth*.

VI. Performance

As mentioned in Section I, the proposed library aims to overcome python's performance limitations (when comparing to more common choices for audio processing, like C++) by segmenting the signal into chunks and manipulating these with numpy's vector operations. Naturally then, performance depends on the chunk size. If too small, vector operation speedups contribute little, which may lead to slower-than-realtime computation. If too large, although the computation itself is fast, the introduced latency becomes the limiting factor. Additionally, due to the described implementation, feedback loops start producing undesired results. We explore this tradeoff with a performance study (tables II, III), where entire computation graphs and single modules are evaluated to produce audio *playtime* in length. We record the actual *computation time* and compute the ratio $\frac{computationtime}{playtime}$. If this ratio is below 1, the computation is fast enough for real time applications. The tests were performed on a mid-range laptop.

The observed results demonstrate that computation speed is heavily influenced by chunk size. At 10 samples per chunk, all three computation graph examples are slower-than-realtime, while at 500 samples per chunk, all are faster. However, in the more common range which still produces usable latency, like 50 samples (1.1 ms at 44Hz) or 100 samples (2.2 ms at 44Hz), is near the threshold. The overtones and vocal remover examples run in realtime, while the wind generator does not. Timing single modules (Table III) provides a possible explanation. The modules implemented as LTI filters (Delay, Lowpass, Modulated Lowpass), which the wind generator example relies on, are the slowest. In particular, filters with many coefficients, that are dynamically modulated, present a bottleneck, since the coefficients need to be recomputed at every time step according to the control signal.

TABLE II
Computation time to playtime ratio when evaluating various example setups with different chunk sizes. A ratio under one means the computation is fast enough to be performed in real time. The presented ratios are an average over 3 evaluations, each with a playtime of 3 seconds.

Chunk size Example	10	50	100	500
Overtones	**2.73**	0.52	0.27	0.06
Vocal remover	**1.65**	0.32	0.16	0.04
Wind	**13.26**	**3.07**	**1.70**	0.48

TABLE III
COMPUTATION TIME TO PLAYTIME FOR SOME SINGLE MODULES. MODULES WITH ONE OR MORE INPUTS USE THE SAME PULSES OBJECT FOR ALL INPUTS TO AVOID ADDITIONAL COMPUTATION.

Chunk size Module type	10	50	100
Sin	0.28	0.06	0.03
Saw	0.27	0.06	0.03
WhiteNoise	0.29	0.06	0.03
Ramp	0.20	0.04	0.02
TriggWave	0.71	0.15	0.08
Envelope	0.71	0.15	0.08
Add (+)	0.30	0.06	0.03
Mul (*)	0.31	0.07	0.03
Delay (»)	**2.82**	**1.35**	0.90
Lowpass	0.39	0.08	0.05
Modulated Lowpass	**8.19**	**1.96**	0.81

VII. LIMITATIONS AND FURTHER WORK

In this section we note some limitations and possible future directions that will be addressed in further work.

As discussed, the main challenge of implementing an audio signal processing library in python is performance. We address this issue by using numpy's vector arithmetic.. However, this optimization is limited with small chunks required for real-time operation. As our evaluation shows, the current implementation cannot achieve real time for certain more complex setups. We wish to address this by further optimizing module operation, specificially in the case of dynamic (modulated) filters, which appear to present a bottleneck

Another limitation arises from the described loop evaluation. Since a loop is evaluated once every time step, loops with combined delays shorted than chunk length, will not produce expected results. This may be solved by detecting loops and sufficiently approximating their output at every time step. Again, though, this presents a performance tradeoff.

Aside from addressing limitations, our work opens some further research directions. This library has been designed as a general, extensible base for further development. One Possible direction is a higher-level modular synthesis library which provides many preset modules, mimicing physical instruments.

VIII. CONCLUSION

As part of the discussed assignment, we developed a pythonic sound synthesis library with the aim to provide a more intuitive and approchable solution in the programatic synthesis space.

The conducted work was an introduction to many difficult challenges involving digital signal processing, software architecture, blocked operation and performance optimization. Although not without limitations, the end product — *pynth* — has achieved the aim of providing a minimal and intuitive interface to sound synthesis. The library is avaliable at https://github.com/MatejBevec/pynth.

REFERENCES

[1] Perry Cook and Gary Scavone. [n. d.] The synthesis toolkit in c++ (stk). https://ccrma.stanford.edu/software/stk/. ().

[2] [n. d.] Jsyn - audio synthesis api for java. http://www.softsynth.com/jsyn/. ().