applied sciences

MDPI

Tutorial

Programming Real-Time Sound in Python

Yuri De Pra † **and Federico Fontana** *,†

HCI Lab, Department of Mathematics, Computer Science and Physics, University of Udine, 33100 Udine, Italy; yuri.depra@uniud.it

* Correspondence: federico.fontana@uniud.it; Tel.: +39-0432-558-432

† These authors contributed equally to this work.

Received: 24 April 2020; Accepted: 16 June 2020; Published: 19 June 2020

Abstract: For its versatility, Python has become one of the most popular programming languages. In spite of its possibility to straightforwardly link native code with powerful libraries for scientific computing, the use of Python for real-time sound applications development is often neglected in favor of alternative programming languages, which are tailored to the digital music domain. This article introduces Python as a real-time software programming tool to interested readers, including Python developers who are new to the real time or, conversely, sound programmers who have not yet taken this language into consideration. Cython and Numba are proposed as libraries supporting agile development of efficient software running at machine level. Moreover, it is shown that refactoring few critical parts of the program under these libraries can dramatically improve the performances of a sound algorithm. Such improvements can be directly benchmarked within Python, thanks to the existence of appropriate code parsing resources. After introducing a simple sound processing example, two algorithms that are known from the literature are coded to show how Python can be effectively employed to program sound software. Finally, issues of efficiency are mainly discussed in terms of latency of the resulting applications. Overall, such issues suggest that the use of real-time Python should be limited to the prototyping phase, where the benefits of language flexibility prevail on low latency requirements, for instance, needed during computer music live performances.

Keywords: real time; sound processing; Python; Cython; Numba; code refactoring

1. Introduction

Among the many definitions, the development of computer music applications has been qualified as "the expression of compositional or signal processing ideas" [1]. In fact, a computer music language should be oriented, in particular, to the development of real-time software. Early computer music developers used to program on their personal computer in C or C++ due to the efficiency of the resulting machine code. This necessity has sometimes caused excessive dependency of the application on the characteristics of the programming language itself. In particular, the high technicality of C and C++ has often discouraged computer musicians to approach problems, requiring solving skills which are domain of computer scientists and engineers instead. To reconcile this divide, abstractions have been proposed leading to specialized languages, such as Csound, Max, Pure Data, SuperCollider, Chuck, Faust, and not only [2]. These abstractions embrace the imperative, functional, and visual programming paradigm. On the one hand, they allow computer musicians to create software with lower effort; on the other hand, the long-term life of their sound applications obviously depends on the continued support of such peculiar languages on standard operating systems and graphic user interfaces. Because of the limited business moved by this niche market, and due to many such languages being maintained (sometimes even for free) by programmers who are also computer musicians, this community has unfortunately been suffering probably more than others from the lack of a systematic, durable approach to the development and maintenance of sound software [3].

Python is continuously increasing in popularity, thanks also of a non-commercial license of use [4]. Its community maintains a lot of official packages, including libraries (e.g., Numpy, Scipy, Matplotlib) providing scientific computing tools that are comparable to those equipping dedicated software, such as Matlab and R. Thanks to a fast learning curve, its rapid prototyping features and intuitive readability of the code, the Python community now also includes users paying particular attention to the interaction aspects of their software, such as academics [5,6] employing Python as a teaching-by-examples tool [7]. Despite this quest for interactivity, the use of Python in real-time applications is testified by exceptions: RTGraph, for instance, instantaneously processes physiological signals and then displays the results through the Qt framework [8].

This limitation is common for interpreted software. Unlike C and other compiled languages, Python, in fact, generates bytecode, which is interpreted by a virtual machine operating at application level. However, this is not the only way to run an application. In particular, Python puts available more than one tool to compile the code, hence affording performances that are only accessible at the processor level. For instance, the Numba library through its just-in-time compiler speeds up numerical iterations, such as those needed by vectorial operations, called by Numpy. Furthermore, chunks of C code can be embedded within a Python program [9] while using the Cython library, which includes a static compiler accepting instructions belonging to several compiled languages as part of a program written in Python. Using Cython, it is also possible to declare static variables, as C programmers do for substantially reducing the time to bind them at runtime. This way, algorithms translated from e.g., Matlab or already in C can be compiled through Cython, furthermore preserving the static memory space. Later, they can be called from the Python environment as standard modules, however computed at machine level. In both Numba and Cython, the refactoring of a few lines of code is often sufficient for optimizing the most computationally intensive parts of an algorithm, with a dramatic speedup of the application. In the sound processing domain, this optimization is often limited to the loops that are indefinitely iterated at the sample rate.

This paper explains how real-time software can be developed in Python without renouncing to all of those features that have made this language a primary option, especially in artificial intelligence research. Specifically, the opportunity to include off-the-shelf machine learning algorithms in a real-time software project has nowadays become even more attractive, since artificial intelligence has recently taken a prominent role in the design and online tuning of digital audio filters and sound effects [10,11]. For this reason, we expect this paper—which is *not* a research paper, and substantially extends material that was presented at a national computer music conference [12]—to be of interest for computer musicians who are planning to import e.g., a new machine learning-based parametric control strategy in their preferred sound processing algorithms, and in general for Python programmers wishing to enrich their background in real-time software.

1.1. Related Work

The number of computer music programming languages is notable. Each language differs in terms of abstraction, coding approach, learning curve, portability across architectures, and operating systems. In parallel, all make real-time sound programming easier. In particular, those in the Music-N tradition [13] support dynamic instantiation of graphs of *unit generators* (UG's), the basic building blocks of a sound synthesis and processing algorithm. Csound [14] structures the code in two parts: an instrument file contains the UG's, while a score file controls them along time through note and other event parameters. Pure Data [15] and Max [16] allow for the visual programming of UG networks and related control messaging. SuperCollider [1] implements a client-server architecture that enables interactive sound synthesis and algorithmic composition through live coding, also interpreting different languages thanks to the flexibility of the client. ChucK supports deterministic concurrency and multiple control rates, providing live coding and enabling live performances [17]. Faust follows the functional programming paradigm, and builds applications and plugins for various real-time

sound environments thanks to the automatic translation into C++ code. It also provides a powerful online editor, enabling agile code development [18].

Existing sound applications in Python mainly focus on the analysis and presentation of audio, as well as on music retrieval [19–21]. Most such applications have no strict temporal requirements and, hence, are conventionally written for the interpreter: examples of this approach to sound programming can be found in e.g., [22], where latency figures are also detailed. Concerning sound manipulation, computer musicians often rely on the pyo library [23], a client-server architecture that allows combining its UG's together into processing networks. Hence, the abstraction from the signal level is realized through pyo also in Python, making the creation of sound generation and effect chains possible as most sound programming languages do.

On the other hand, the low-level development of sound algorithms is not trivial when working with UG's, as they encapsulate the signal processing by definition. Additionally, because of the existing excellence in this sound programming paradigm, our paper puts the accent to coding at signal level. Addressing such a level in Python requires to profile and refactor usually few signal processing instructions. The advantages of code refactoring go beyond sound applications; in fact, refactoring can be applied to contexts, including, among others, real-time data collection, systems control, and automation.

1.2. On Real-Time Processing

By definition, real-time processes produce an output within a given time. Concerning sound processing, this time is nominally inversely proportional to the audio sampling rate. Our test environment is an Intel-i5 laptop computer running Windows 10, connected to a RME Babyface Pro external USB audio interface. Contrarily to hardware/software systems specifically oriented to real-time audio [24], in such a standard architecture a sound process can be stopped by the operating system (OS) scheduler for too long or too many times within an allowed time window, hence becoming ultimately unable to regularly refill the audio output buffer at sample rate, with consequent sound glitches and distortions in the output. A common workaround to this problem, which is known as buffer underflow, consists of increasing the size of the audio buffer. Because a sound process normally produces samples much faster than the audio sample rate, this workaround decreases the probability for the audio interface to find the buffer empty.

However, longer buffer size comes along with a proportionally higher latency of the output. Latency can be a negligible issue in feed-forward sound interactions; conversely, it can annihilate a closed-loop musical perception-and-action, where typically no more than 10 milliseconds are allowed for a computer music setup to respond to musicians. Thus, a buffer size must compromise between probability of audio artifacts and tolerable latency, and only the aforementioned systems can reliably fulfill the low-latency needs of live electronic music interactions.

Irrespective of the buffer size, higher latency is generally beneficial for sound quality, since the OS scheduler in that case can stop the sound process occasionally for a longer while, e.g., to handle an external interrupt. However, the developer should always avoid to include unbounded time operations in a sound processing thread. Rather, input/output (I/O) instructions, graphic functions, and, in general, all procedures in charge of the interaction with the system, should be implemented by parallel threads sharing lock-free data structures with the sound processing thread. In this regard, practical general suggestions on real-time programming can be found in thematic discussions on the Internet [25]. At any rate, Python is not designed to support the servicing of audio threads within deterministic temporal constraints. For this reason, Python applications should not be programmed with the purpose to guarantee real-time sounds at low latency, regardless of the performances reported for our test environment in the following sections.

1.3. Structure of the Paper

The paper is structured, as follows: Section 2 explains real-time software interpretation in Python through a simple example. Section 3 introduces programming and code profiling with Numba and Cython. Section 4 applies the above concepts on two sound algorithms that can be profiled for their running in real time. Such examples have been put available on GitHub, (https://github.com/yuridepra88/RealtimeAudioPython) along with the scripts that have been used to benchmark the algorithms. Section 5 discusses the results in front of the constraints that are imposed by the OS to real-time process running. Section 6 concludes the paper. Finally, Appendix A contains companion code listings.

2. Interpreted Approach

Real-time program development first of all needs to manage sound I/O through a low-level application programming interface. Concerning Python, the library PyAudio [26], among others, provides bindings for portaudio, an open-source cross-platform audio device. As most low-level libraries do, PyAudio allows for operating sample-by-sample on audio chunks whose size is set by the user. Typical chunks range between 64 and 2048 samples. PyAudio provides a blocking mode, enabling synchronous read and write operations, as well as a non-blocking mode managing the same operations through a callback by a separate thread. On top of I/O, Python provides libraries supporting the agile development of virtual sound processors: the module *scipy.signal* within the Scipy library, for instance, contains some standard signal processing tools. An exhaustive list of Python libraries supporting audio analysis and processing can be found in [27]. Further examples of the interpreted approach to sound programming are presented in [22].

In Listing A1, the basic structure of a callback procedure enabling the asynchronous processing of an audio chunk at sample rate is reported. Each time the procedure is called, one chunk is read from the audio buffer and then assigned to the array *data*, containing accessible sound samples. The functions *pcm2float(byte[])* and *float2pcm(float[])* convert each sample from raw bytes to $[-1., 1.]$-normalized floats and vice versa. Finally, the function *process(data[])* encapsulates the processing algorithm.

Building upon this structure, a simple procedure is exemplified implementing the following low-pass digital filter [28]:

$$y[n] = \alpha x[n] + (1 - \alpha)y[n - 1], \quad 0 < \alpha < 1\,. \tag{1}$$

Figure 1 shows a simple graphical interface for this low-pass filter control. The interface displays also the Fast Fourier transform computed at runtime on each chunk by the *scipy.fft(float[])* method, provided by Scipy. An example that implements different Butterworth filters meanwhile providing parameter controls to the user is available on our GitHub repository.

Listing A2 shows the respective implementation: each time the *process(data)* function is called, the samples in the array are sequentially processed by the algorithm to form one output chunk; moreover, the last output sample is stored in the variable *last_sample* for processing the first sample of a new chunk when *process(data)* is called next. The variable *last_sample* must belong to the global scope since carrying a value between subsequent function calls. As opposed to other languages, global variables in Python must be declared at the beginning of a function using the *global* identifier if they are later written within an instruction appearing inside the same function. Otherwise, a local variable with the same name is automatically generated. Conversely, the variable *alpha* does not need such declaration; in fact this variable is read inside the function and, concurrently, written by the control thread that assigns a value depending on the slider position visible in Figure 1. These considerations are crucial not only to prevent from incorrect use of the variable scope. In fact, as explained in the next Section, global variables must be correctly refactored, depending on the optimization tool.

Alternatively, Python offers the possibility to program the UG's as objects, hence inherently encapsulating every UG state as part of the corresponding object variables. By guaranteeing code

modularity through the unit generator abstraction, the UG-based/object-oriented approach essentially removes the need to manage global variables, with major advantages when sets of identical UG's, such as oscillators or filters, must be first instantiated and then put in communication with each other. As an example, we uploaded in GitHub the object class *OscSine()*, allowing for multiple instances of a simple sinusoidal oscillator. Similarly to the previous simple low-pass filter, this example will be refactored and, hence, proposed again in the next sections. However, coherently with the initial aim of exploring low-level sound programming, we will give emphasis to procedural instead of object-based examples. Some arguments are in favor of this choice. For instance, especially in the case of nonlinear systems such as those being presented in Section 4, the UG-based approach shows limits as soon as a sound algorithm improvement requires to concatenate existing UG's in the form of a delay-free loop [29]. In such a case, programming a new object lumping together such UG's can be much more time-consuming and error-prone than adapting an existing procedure to compute the delay-free loop. Unfortunately, the interpreter fails to compute sounds in time, even at audio sample rate as soon as the processing algorithm falls outside simple cases. Although on the one hand dynamic interpretation allows faster development and reduces programming errors, on the other hand it slows down the computation. Among the causes of this slow down, dynamic typing has been recognized to prevent the interpreter from achieving the real time. In this regard, the Python library *line_profiler* [30] can be used to profile and analyse code performances: such library measures the code execution time line-by-line. An example of interpreted code profiling is reported in Listing A3, where the cost of incrementing a variable and computing a *sin()* function are measured. This example will be proposed again in the next section, to benchmark the refactored code.

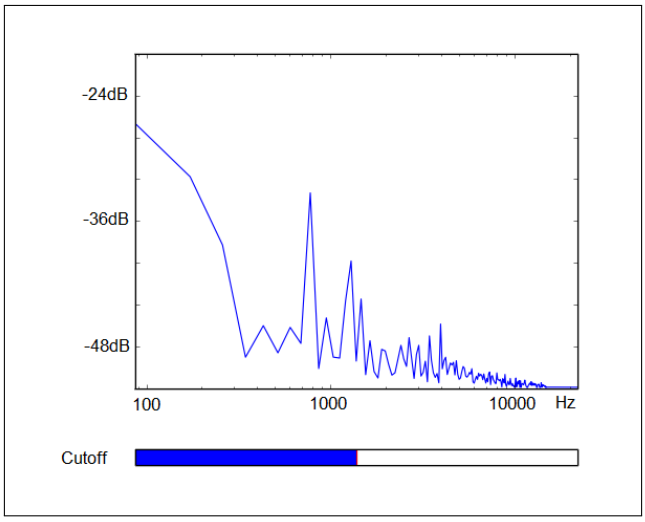

Figure 1. Real-time spectrum of the output signal and low-pass filter cutoff parameter control.

3. Code Speedup

Because od the dynamic interpretation of the bytecode, Python does not allow for declaring static variables. On the other hand static variables speed up access to the data, since they are not allocated in the local memory of a function every time it is called. In practice, the conversion into static of any possible variable that is used intensively in a program can bring substantial performance benefits. As part of their optimization features, Numba and Cython allow to declare static variables.

3.1. Numba

Numba is a just-in-time (JIT) compiler for vectorial computing in Python [31]. Using Numba, it is possible to speed up functions containing instructions that can be computed more efficiently at machine instead of application level, such as those applying to NumPy objects. In this case, the instructions are sent to the JIT compiler by adding the decorator *@jit* before declaring a function.

At this point, the bytecode is translated in Low Level Virtual Machine Intermediate Representation (LLVM IR), an architecture-independent language that is similar to assembly. Numba provides two operating modes, that are assigned during the decoration: the *nopython* mode (i.e., *@jit(nopython=True)*) compiles the code completely, whereas the *object* mode tries to compile the loops while leaving to the interpreter the instructions outside them. The performance improvement resulting from the *object* operating mode is limited and, in general, not sufficient for achieving the real time.

Numba is easy to use; furthermore, it requires almost no changes of the Python code. On the other hand it translates only a fraction of the Python instruction set, even if its expressive power is increasing release after release. Most importantly, the functions compiled in *nopython* mode cannot access Python objects at runtime; thus, data can be exchanged between such functions and a Python program only in form of input and return arguments. This limitation asks for refactoring the communication among functions and restructuring the code into simple functions, only containing critical instructions (i.e., arithmetic loops).

Numba is instructed about static variables through the decorator, that must explicitly refer to the input and return arguments as Listing A4 shows. Without this information, the compiler in some cases could be unable to determine the variable scope and, consequently, force the *object* mode. In the specific case a wrapper function named *wrap_numba_global()* is needed to declare the global variable *a*, and then pass it to the function *numba_global(a)* as an argument. Otherwise, *numba_global* cannot access its content since belonging to the runtime space. Listing A4 shows also the time profiling concerning the computation of the *sin()* function applied to iterated variable after Numba refactoring. *wrap_numba_global()* is called twice: in the first call the JIT compiler optimizes the code, hence slowing down the execution; from the second call on, the optimized cycle is computed in as fast as 7.45 ns.

3.2. Cython

Cython extends Python by allowing explicit type declarations and direct compilation of C code [9]. Almost all of the Python libraries for scientific computing use Cython to minimize the computational burden, also because most algorithms provided by these libraries were already optimized on pre-existing pieces of C, C++ or even Fortran code. In fact, Cython is able to wrap functions written in different compiled languages. Unlike Numba, Cython is an ahead-of-time (AOT) compiler: during the so-called *cythonization*, C code is generated from additional (.pyx) files written by the developer during refactoring. Such files contain Python code (that is optimized automatically) and/or function wrappers of external routines in other languages. The efficiency of the cythonization can be measured as usual through code profiling: by just adding an option to the cythonization command, Cython even offers pictorial highlighting of the instructions proportionally to their computational cost.

Listing A5 repeats the refactoring example seen in Listing A4, this time in Cython. Before the functions declaration, the global variable is defined static using the annotation *cdef*. Although Cython reduces the processing time by about 50% the interpretation of the same procedure, Numba generates a faster code at the cost of the initial optimization.

On the other hand, when compared to Numba, Cython abstracts less layers, hence allowing easier debugging and optimization of the code. However, the creation of additional files (at least one) is needed. As an example, Listing A6 revisits the simple low-pass filter implementation seen in Section 2. After cythonization, which makes available the function *lp* by compiling *low_pass.pyx*, the resulting *low_pass* module can be launched from a Python program:

- **c_low_pass.c** contains the procedure within the C function *lp*.
- **low_pass.pyx** binds the C file to the corresponding Python functions. From it, Cython compiles the corresponding module *low_pass*.
- **test_lp.py** calls the *low_pass* module functions.

For the sake of comparison, the Cython and Numba refactored code of the *OscSine()* object class have been uploaded in GitHub, together with time profiling.

3.3. Comparisons & Benchmarking

For what we have seen, Numba and Cython target two different approaches to code refactoring. Below, the most important features and differences are briefed.

- In general, code refactoring is easier and faster in Numba than Cython.
- Numba performs well if arithmetic operations are repeatedly computed without communicating with Python objects at runtime.
- Cython code is compiled once (AOT), while Numba optimizes a script at each new execution (JIT).
- Cython requires writing C instructions.
- Cython allows for debugging and profiling the code after translation.
- Cython allows embedding existing code written in C or other languages.

The computational performances of cythonized functions encoding standard algorithms, such as Fibonacci, FASTA, binary-tree visits in different programming languages can be benchmarked, thus measuring processor and memory occupation. For instance, the Computer Language Benchmark Game [32] compares the processing time differences between interpreted Python and C code, highlighting a dramatic performance drop of some Python implementations. Using Pybenchmarks [33] instead, several pairwise comparisons among Python algorithm implementations (e.g., Python3 vs. Numba or Numba vs. Cython) can be performed. In general, Numba requires less computational resources than Cython. On a final note, Numba and Cython are not the only libraries that are available for code optimization and speedup. A thorough discussion about compiled Python can be found in [34], whereas examples of C functions written for processing sounds using the portaudio device can be found in [35]. Together, such documentary resources provide a broad picture of the pros and cons coming from using interpreted or compiled Python instead of C/C++.

4. Applications to Virtual Analog

Virtual analog is a branch of sound processing that deals with the modeling and real-time simulation of analog effects [36]. We implemented two algorithms that consistently load the processor, respectively, modeling a ring modulator and a voltage-controlled filter. The code was benchmarked using the *line_profiler* and the *memory_profiler* [37] libraries concerning the processing time and memory occupation, respectively.

4.1. Ring Modulator

The diode-based ring modulator model is known to be computationally demanding when realized in the digital domain. A popular model [29] of this musical circuit, see Algorithm 1, asks for computing three non-linear and six linear difference equations at every incoming pair of signal samples, v_m and v_c. Non-linearity is expressed by the voltage-to-current characteristic $g(v) = 0.17v^4$ if $v > 0$, and zero otherwise.

To avoid instability, the digital input needs to be oversampled by a factor of at least 25 times the standard audio rate. The Python interpreter falls short in achieving real time as it computes the above equations at about one thirtieth of the audio rate, hence hundreds of times as slow as it should, as it can be seen from Table 1. As expected, an analysis made with *line_profiler* shows that most of the time is spent to repeatedly iterate over the instructions computing the circuit equations.

Numba hits the goal instead, enabling reliable real-time processing, as reported in Table 1. The related refactoring (the pseudo-code is shown in Listing A7) consists of the creation of a wrap function *compute_numba()*, passing the global variables to the compiled function *compute_numba_low()* that computes the circuit equations. Moreover, the compiler is instructed about the static character of the input and return arguments.

The easiest Cython refactoring of the program consists of moving the computing function *compute()* into an external *.pyx* file, and, furthermore, declaring the static variables sent to compilation using the *cdef* annotation. The rest of the code is left to standard interpretation. Cython enables real-time

processing almost as efficiently as Numba. The corresponding pseudo-code for Cython is shown in Listing A8. The code chunks appearing in Listings A9, A10, and A11, show the refactoring using an external C file. This file contains the instructions describing the circuit equations. The translation in C is straightforward, since no additional libraries are needed for computing the related mathematical expressions. The final software architecture consists of *c_diode.c* and *diode.pyx*, forming the algorithm library, and the main file *test_diode.py* that calls the external functions importing the *diode* module. As expected, the manual transcription in C of the equations yields almost the same performance as the previous cythonization.

Algorithm 1: Ring modulator core processing loop.

repeat

read v_m, v_c

compute $v_1 = \frac{R_m}{1+CF_sR_m}\left(\frac{v_m}{R_m} + i_1 - \frac{g(v_4)}{2} + \frac{g(v_5)}{2} - \frac{g(v_6)}{2} + \frac{g(v_7)}{2}\right) + v_1$

compute $v_2 = \frac{R_a}{1+CF_sR_a}\left(i_2 + \frac{g(v_4)}{2} - \frac{g(v_5)}{2} - \frac{g(v_6)}{2} + \frac{g(v_7)}{2}\right) + v_2$

compute $v_3 = \frac{R_i}{1+C_pF_sR_i}\left(g(v_4) + g(v_5) - g(v_6) - g(v_7)\right) + v_3$

compute $v_4 = \frac{v_1}{2} - \frac{v_2}{2} - v_3 - v_c$

compute $v_5 = -\frac{v_1}{2} + \frac{v_2}{2} - v_3 - v_c$

compute $v_6 = \frac{v_1}{2} + \frac{v_2}{2} + v_3 + v_c$

compute $v_7 = -\frac{v_1}{2} - \frac{v_2}{2} + v_3 + v_c$

compute $i_1 = -\frac{1}{LF_s}v_1 + i_1$

compute $i_2 = -\frac{1}{LF_s}v_2 + i_2$

until ever.

Table 1. Diode ring modulator: mean processing time and memory occupation of different implementations; chunk_size = 512; processing time averaged over 1000 chunks. (Real-time limit: 22.6 µs/sample).

	Processing Time	**Memory Occupation**
Interpreted Python	630.2 µs/sample	118 KB
Cython annotations	1.98 µs/sample	230 KB
Cython + native C	1.98 µs/sample	301 KB
Numba	1.93 µs/sample	219 KB

Table 1 also reports the memory requirements. In particular, Numba and Cython require more space than interpretation. This space considers only the running application; the overall Python environment needs almost 147 MB of memory.

4.2. Voltage-Controlled Filter

As another example, the voltage-controlled filter aboard the EMS VCS3 sound synthesizer is presented [38]. In this case, see Algorithm 2, the core part of the algorithm consists of iterating the computation of five non-linear and five linear equations until the loop converges to a fixed point solution; at this point, the state of the filter is updated and a new input sample v_{IN} can be processed:

The translation of this core is as straightforward as the development of the wrapper, which can be prepared similarly to the previous example.

For the sake of performance comparison, a Numba and an annotated Cython version of the same algorithm were also prepared. Cython is about five times as fast as Numba, as it can be seen in Table 2. This difference highlights the computational advantage for the C application of keeping the filter state within global variables instead of continuously passing their values through arguments, as Numba

does instead. Moreover, the benefits of importing few C instructions accelerate Cython by about fifty times as much. In fact, the manual reprogramming of the hyperbolic tangent function using the *math.c* library translates in a notably more efficient implementation of the same code.

Algorithm 2: Voltage-controlled filter core processing loop.

repeat
  read v_{IN}
  repeat
    compute $v^*_{OUT} = v_{OUT}$
    compute $u_1 = \tanh\big((v^*_{OUT} - v_{IN})/2V_T\big)$
    compute $v_{C1} = I_0(u_2 + u_1)/(4CF_S) + s_1$
    compute $u_2 = \tanh\big((v_{C2} - v_{C1})/2V_T\big)$
    compute $v_{C2} = I_0(u_3 - u_2)/(4CF_S) + s_2$
    compute $u_3 = \tanh\big((v_{C3} - v_{C2})/2V_T\big)$
    compute $v_{C3} = I_0(u_4 - u_3)/(4CF_S) + s_3$
    compute $u_4 = \tanh\big((v_{C4} - v_{C3})/2V_T\big)$
    compute $v_{C4} = I_0(u_5 - u_4)/(4CF_S) + s_4$
    compute $u_5 = \tanh(v_{C4}/6\gamma)$
    compute $v_{OUT} = (K + 1/2)v_{C4}$
  until $|v_{OUT} - v^*_{OUT}| < 10^{-4}|v^*_{OUT}|$
  compute $s_1 = v_{C1}/(2F_S) + s_1$
  compute $s_2 = v_{C2}/(2F_S) + s_2$
  compute $s_3 = v_{C3}/(2F_S) + s_3$
  compute $s_4 = v_{C4}/(2F_S) + s_4$
until ever.

In any case, Table 2 reports the computation times for the program when it is in running state. As anticipated in Section 1.2, the same program can take much longer to deliver sound samples to the audio interface in the multitasking environment. This fact is discussed in more detail in the next section.

Table 2. Voltage-controlled filter: mean processing time and memory occupation of different implementations; chunk_size = 512; processing time averaged over 1000 chunks. (Real-time limit: 22.6 µs/sample).

	Processing Time	**Memory Occupation**
Interpreted Python	199.3 µs/sample	<1 KB
Cython annotations	6.5 µs/sample	16 KB
Cython + native C	0.14 µs/sample	12 KB
Numba	1.36 µs/sample	12 KB

5. Discussion

Table 3 reports statistics for the voltage-controlled filter implemented using the Cython + native C approach under different OS load conditions. In all cases, a chunk size of 64 samples was set. The real-time limit for this chunk is equal to 64 × 22.6 µs ≈ 1.45 ms. According to the values reported in Table 2, the process takes 64 × 0.14 = 8.96 µs processor time for each chunk. However, in the best case, a chunk was computed in about 37 µs in our test environment. The extension of this

extra time depends on the priority assigned to the real-time process by the OS. Using the highest load that we were able to reproduce, the consequent extra time (31.6 ms in Table 3) exceeded the real-time limit for this chunk size by no less than 20 times.

Table 3. Statistics for the voltage-controlled filter running at different OS loads, with audio chunks of 64 samples: audio application only, audio application with I/O operations in the Python GUI, and audio application with copy of a directory tree in background.

Load	Audio Only			GUI Operations			File Copy		
	Idle	Process	Total	Idle	Process	Total	Idle	Process	Total
Min (ms)	0.005	0.039	0.044	0.005	0.037	0.042	0.005	0.038	0.044
Max (ms)	36.5	8.7	36.8	33.9	31.6	40.9	27.9	29.9	29.9
Average (ms)	1.31	0.14	1.44	1.32	0.13	1.44	1.35	0.09	1.45
Violations (%)	11%	0.17%	11.2%	11%	0.8%	11.8%	11%	0.15%	11.2%

Furthermore, Table 3 in the last row reports the percentage of violations of the real-time limit, measured as the time between two subsequent callbacks (idle time). Because no buffer underflow and consequent sound artifacts occur in consequence of such violations, certainly the system alternated audio chunks preemption with prioritization of non real-time processes. In fact, portaudio creates a further buffering level hence smoothing out delays due to algorithmic and/or OS bottlenecks, at the cost of introducing additional latency—about 50 ms in our tests. Figure 2 shows that, in practice, few longer delays interleaved with several short idle times. Such delays happen when the Python process waits for the next chunk of samples to become available from the audio driver.

The total latency of the system depends on multiple factors: the audio interface hardware, the audio drivers, the buffering levels introduced by the system, and the chunk size. In our test environment, latency grew up to about 500 ms when the voltage-controlled filter was computed using the internal sound card instead of the Babyface. More generally, Python does not acquire special privileges from the OS or address possible priority inversion problems, and even an audio thread must contend with non-real-time Python threads for the interpreter lock. Furthermore, real-time Python programmers should always check the virtual memory occupation of their software, as page faults can have a huge impact on real-time performance. Unless these issues are rigorously managed, uninterrupted audio, in principle, cannot be guaranteed by a Python application.

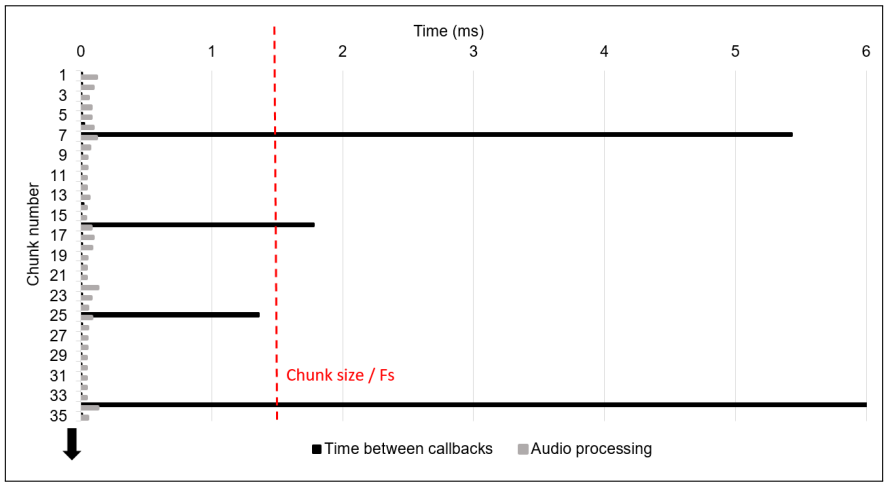

Figure 2. Timing of audio callbacks and audio processing in the voltage-controlled filter algorithm.

6. Conclusions

In this paper, we have made a short tour to show the potential and limits of Python for real-time sound programming, starting with a simple digital filter until more complex virtual analog models.

If the interpreter alone has limited application, conversely refactoring critical parts of the code for libraries, like Numba or Cython, allows for also processing complex algorithms in real time. In particular:

- those parts of the code which deal with environment setup, type conversions, user interaction, signal analysis and interface development can be left to the interpreter;
- procedures that do not need to intensively interact with the program workspace can be compiled with Numba;
- procedures, which, conversely, must often interact with the program workspace in terms of e.g., memory allocation or signal resampling and filtering, can be compiled with Cython; and,
- finally, already existing procedures written in C or other languages can also be efficiently cythonized.

Tests of two computationally-intensive sound algorithms were made under different OS loads, leading to accurate audio outputs with a latency of about 50 ms. Lower latency values were out of reach, due to the Python threading mechanism and its interaction with the OS scheduler. While this limit prevents from using the resulting applications in critical interactive music contexts, on the other hand Python certainly has potential for integrating sets of computationally-intensive algorithms in effective real-time simulation frameworks.

Author Contributions: Conceptualization, Y.D.P.; methodology, Y.D.P.; software, Y.D.P.; validation, Y.D.P.; formal analysis, Y.D.P.; investigation, Y.D.P.; resources, Y.D.P. and F.F.; data curation, Y.D.P.; writing–original draft preparation, Y.D.P.; writing–review and editing, Y.D.P. and F.F.; visualization, Y.D.P. and F.F.; supervision, F.F.; project administration, Y.D.P. and F.F.; funding acquisition, Y.D.P. and F.F. All authors have read and agreed to the published version of the manuscript.

Funding: The APC was funded by Electrolux Professional SpA.

Acknowledgments: We especially thank Anonymous Reviewer #1 for carefully reading this manuscript during every review round and contributing to its final form. Yuri De Pra's was funded with a scholarship from the Research Hub by Electrolux Professional SpA.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. Code Listings

Listing A1: Audio I/O – Callback function

```
def callback(i_d, frame_count, t_info, f):
  data = wavefile.readframes(frame_count)
  samples = pcm2float(data)
  y = process(data)
  out = float2pcm(y)
  return (out, pyaudio.paContinue)
```

Listing A2: Low-pass filter implementation

```
def process(data):
  global last_sample

  b = 1-alpha
  y= np.arange(CHUNK_SIZE, dtype=float)

  y[0] = alpha*data[0] + b*last_sample
  for i in range(1,CHUNK_SIZE-1):
      y[i] = alpha*data[i] + b*y[i-1]
  last_sample = y[CHUNK_SIZE-1]
  return y
```

Listing A3: Global variable profiling–interpreted Python

```
Function: try_global
Hit time unit: 3.41*10^-7 s
Total time: 0.1037 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1         1          3.0      3.0      0.0  a = 0
     1                                           def try_global():
     2                                           global a
     3         1          3.0      3.0      0.0  i = 0
     4   100,001      85,289.0      0.9     28.0  while i < 100000:
     5   100,000      92,719.0      0.9     30.5  i += 1
     6   100,000     126,083.0      1.3     41.5  a = math.sin(i)
```

Listing A4: Global variable profiling–Numba

```
from numba import jit, float32
import math
a=0.
@jit(float32(float32),nopython=True)
def numba_global(a):
        i=0
        while i < 100000:
                i+=1
                a = math.sin(i)
        return a

# Execution and benchmark of the refactored function
Function: wrap_numba_global
Hit time unit: 3.41*10^-7 s
Total time: 0.0474 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def wrap_w_numba_global():
     2                                           global a
     3         1     129,328.0 129,328.0     93.0  wrap_numba_global()
     4         1          3.0      3.0      0.0  a = 1.
     5         1       2183.0   2183.0      1.6  wrap_numba_global()
```

Listing A5: Global variable profiling–Cython

```
Function: cython_try_global
Hit time unit: 3.41*10^-7 s
Total time: 0.0526 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1         1          3.0      3.0      0.0  cdef float a = 0
     2                                           def cython_try_global():
     3                                           global a
     4         1          4.0      4.0      0.0  cdef int i = 0
     4         1          1.0      1.0      0.0  while i < 100000:
     8   100,000      55,087.0      0.6     37.4  i += 1
     9   100,000      92,154.0      0.9     62.6  a =math.sin(i)
```

Listing A6: Low-pass filter implementation–Cython

```
==========================
c_low_pass.c
==========================
double last_sample = 0;
double b = 0;

void lp(double* data, int samples, double* a){
   b = 1 - a;
   data[0] = a*data[0] + b*last_sample;
   for (int i = 1; i < samples; i++) {
          data[i] = a*data[i] + b*data[i-1];
   }
   last_sample = data[samples-1];
   return;
}

==========================
low_pass.pyx
==========================
import cython
...
#declare interface to C code
cdef extern void lp(double* data, int samples, double* a)

# c-array that contains samples
cdef double *cdata

#function compiled as module
def process(np.ndarray[double] data, int samples, double alfa):
  ...
  lp(cdata, samples, a)
  ...
  return data

==========================
test_lp.py
==========================
import low_pass
...
# ''a'' is the user parameter
def process(data):
    return low_pass.process(data, CHUNK_SIZE, a)
```

Listing A7: Ring modulator–Numba

```
#numba decorator with static type declaration and mode
@jit (types.Tuple((returned_values_types))(parameters_types), nopython=True)
def compute_numba_low(c_data, m_data, u1,u2,u3,u4,u5,u6,u7,i1,i2):

    for i in range (FRAMES_PER_BUFFER*over_f):
        equation1
        ...
        equation9
        data[i] = equation2_result

    return data,u1,u2,u3,u4,u5,u6,u7,i1,i2

#numba wrapper
def compute_numba(c_data, m_data):
   global u1,u2,u3,u4,u5,u6,u7,i1,i2,zi

   resample of two buffers to oversampling rate using 0-padding

   data,u1,u2,u3,u4,u5,u6,u7,i1,i2 = compute_numba_low(c_data,m_data,
                                          u1,u2,u3,u4,u5,u6,u7,i1,i2)
   LPF on returned data

   return data[::over_f] #take one sample each over_f samples
```

Listing A8: Ring modulator–Cython

```
============================
diode_cython.pyx
============================
#this file needs to be compiled

#static type declaration of global variables

cdef double u1 = 0.
...

# parameter type def. e.g.,(np.ndarray[double, ndim=1, mode="c"]
c_data not None,...)
def compute_c(c_data, m_data, FRAMES_PER_BUFFER, T, ovr_f, a, b, zi):

    global u1,u2,u3,u4,u5,u6,u7,i1,i2

    resample of two buffers to oversampling rate using 0-padding

    samples = FRAMES_PER_BUFFER*ovr_f
    for i in range (0,samples):
        equation1
        ...
        equation9
        c_data[i] = equation2_result

    LPF on returned data

    return c_data[::ovr_f] #take one sample each ovr_f samples
```

Listing A9: Ring modulator–Cython (core procedure)

```
============================
diode.pyx
============================
#this file needs to be compiled

#declare interface to external C code
cdef extern void ring(double* c,double* m, int s, float T)

declaration of C arrays and memory space allocation

# parameter type def. e.g.,(np.ndarray[double] c_data,...)
def process(c_data, m_data, FRAMES_PER_BUFFER, T, ovr_f, a, b, zi):

   cdef unsigned int i

   resample of two buffers to oversampling rate using 0-padding

   move data from python np.arrray to C memory located

   #call the C function
   compute_c(cdata,mdata,FRAMES_PER_BUFFER*ovr_f, T)

   read modified data into np.array

   LPF on returned data

   return c_data[::ovr_f] #take one sample each ovr_f samples
```

Listing A10: Ring modulator–Cython (binding)

```
============================
c_diode.c
============================
// declare global variables
double u1 = 0;
...

void ring(double* carr_in, double* mod_in, int samples, float T){
    int i = 0;
    for (i = 0; i < samples; i++) {
        equation1
        ...
        equation9
        carr_in[i] = equation2_result
  }
  return;
}
```

Listing A11: Ring modulator–Cython (main)

```
============================
test_diode.py
============================
import diode
...
ovr_f = 30 #oversampling factor
T = 1./(s_rate*ovr_f)
b,a = butt_low(sample_rate/ovr_f, s_rate, 5)
zi = signal.lfilter_zi(b,a)

def process(c_in, m_in):
  global zi
  return diode.process(c_in, m_in, CHUNK_SIZE,T, ovr_f, a, b, zi)
```

References

1. McCartney, J. Rethinking the Computer Music Language: SuperCollider. *Comput. Music J.* **2002**, *26*, 61–68. [CrossRef]
2. Adams, A.T.; Latulipe, C. Survey of Audio Programming Tools. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*; CHI EA '13; ACM: New York, NY, USA, 2013; pp. 781–786. [CrossRef]
3. Morreale, F.; McPherson, A.P. Design for longevity: Ongoing use of instruments from NIME 2010–14. In Proceedings of the International Conference on New Interfaces for Musical Expression (NIME'17), Copenhagen, Denmark, 15–19 May 2017; pp. 192–197.
4. Krishan Kumar, S.D. Programming Languages: A Survey. *Int. J. Recent Innov. Trends Comput. Commun.* **2017**, *5*, 307–313.
5. Guo, P. Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. 2014. Available online: https://cacm.acm.org/blogs/blog-cacm/176450 (accessed on 18 June 2020).
6. Mason, M.; Cooper, G.; Raadt, M. Trends in Introductory Programming Courses in Australian Universities Languages, Environments and Pedagogy. In Proceedings of the Australasian Computing Education Conference (ACE2012), Melbourne, Australia, 31 January–3 February 2012; CRPIT; Raadt, M., Carbone, A., Eds.; ACS: Melbourne, Australia; Volume 123, pp. 33–42.
7. Downey, A.B. *Think DSP: Digital Signal Processing in Python*, 1st ed.; O'Reilly Media, Inc.: Newton, MA, USA, 2016.
8. Sepúlveda, S.; Reyes, P.; Weinstein, A. Visualizing physiological signals in real-time. In Proceedings of the 14th Python in Science Conference (SciPy 2015), Austin, TX, USA, 6–12 July 2015; pp. 190–194. [CrossRef]
9. Behnel, S.; Bradshaw, R.; Citro, C.; Dalcin, L.; Seljebotn, D.S.; Smith, K. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* **2011**, *13*, 31–39. [CrossRef]
10. Wright, A.; Damskägg, E.P.; Juvela, L.; Välimäki, V. Real-Time Guitar Amplifier Emulation with Deep Learning. *Appl. Sci.* **2020**, *10*, 766. [CrossRef]
11. Martínez Ramírez, M.A.; Benetos, E.; Reiss, J.D. Deep Learning for Black-Box Modeling of Audio Effects. *Appl. Sci.* **2020**, *10*, 638. [CrossRef]
12. De Pra, Y.; Fontana, F.; Simonato, M. Development of real-time audio applications using Python. In Proceedings of the XXII Colloquium of Musical Informatics, Udine, Italy, 22–23 February 2018; pp. 226–231.
13. Mathews, M.V.; Miller, J.E.; Moore, F.R.; Pierce, J.R.; Risset, J.C. *The Technology of Computer Music*; MIT Press: Cambridge, MA, USA, 1969; Volume 5.
14. Vercoe, B.; Ellis, D. Real-time CSound: Software Synthesis with Sensing and Control. In Proceedings of the International Computer Music Conference, Glasgow, Scotland, 10–15 September 1990; pp. 209–211.
15. Puckette, M. Pure Data: another integrated computer music environment. In Proceedings of the International Computer Music Conference, Hong Kong, China, 19–24 August 1996; pp. 37–41.
16. Cycling '74. Max/MSP Website. 2020. Available online: https://cycling74.com/products/max/ (accessed on 18 June 2020).

17. Wang, G.; Cook, P.R. ChucK: A Concurrent, On-the-Fly, Audio Programming Language. In Proceedings of the International Computer Music Conference, Singapore, 29 September–4 October 2003; pp. 219–226.
18. Fober, D.; Orlarey, Y.; Letz, S. Faust Architectures Design and Osc Support. In Proceedings of the International Conference on Digital Audio Effects (DAFx-11), Paris, France, 19–23 September 2011; pp. 231–236.
19. Glover, J.C.; Lazzarini, V.; Timoney, J. Python for Audio Signal Processing. In Proceedings of the Linux Audio Conference 2011, Maynooth, Ireland, 6–8 May 2011; pp. 107–114.
20. McFee, B.; Raffel, C.; Liang, D.; Ellis, D.; McVicar, M.; Battenberg, E.; Nieto, O. librosa: Audio and Music Signal Analysis in Python. In Proceedings of the 14th Python in Science Conference, Austin, TX, USA, 6–12 July 2015; pp. 18–25. [CrossRef]
21. Bellini, D. Expressive Digital Signal Processing (DSP) Package for Python. 2016. Available online: https://github.com/danilobellini/audiolazy (accessed on 18 June 2020).
22. Wickert, M. Real-Time Digital Signal Processing using pyaudio_helper and the ipywidgets. In Proceedings of the 17th Python in Science Conference, Austin, TX, USA, 9–15 July 2018; pp. 91–98. [CrossRef]
23. Belanger, O. Pyo, the Python DSP Toolbox. In Proceedings of the 24th ACM International Conference on Multimedia, Amsterdam, The Netherlands, 15–19 October 2016; ACM: New York, NY, USA, 2016; pp. 1214–1217. [CrossRef]
24. ELK Audio. Audio Latency Demystified. 2020. Available online: https://elk.audio/audio-latency-demystified-part-ii/ (accessed on 18 June 2020).
25. Bencina, R. Real-Time Audio Programming. 2011. Available online: http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing (accessed on 18 June 2020).
26. Pham, H. PyAudio Website. 2006. Available online: https://people.csail.mit.edu/hubert/pyaudio (accessed on 18 June 2020).
27. Stöter, F. Repository of Python Scientific Audio Packages. 2020. Available online: https://github.com/faroit/awesome-python-scientific-audio (accessed on 18 June 2020).
28. Mitra, S.K. *Digital Signal Processing. A Computer-Based Approach*; McGraw-Hill: New York, NY, USA, 1998.
29. Fontana, F.; Bozzo, E. Explicit Fixed-Point Computation of Nonlinear Delay-Free Loop Filter Networks. *IEEE/ACM Trans. Audio Speech Lang. Process.* **2018**, *26*, 1884–1896. [CrossRef]
30. Kern, R. Line Profiler Code Repository. 2020. Available online: https://github.com/rkern/line_profiler (accessed on 18 June 2020).
31. Lam, S.K.; Pitrou, A.; Seibert, S. Numba: A LLVM-based python jit compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, Austin, TX, USA, 11 November 2015; pp. 1–6.
32. The Computer Language Benchmarks Game. Benchmark Python vs. C++. 2020. Available online: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gpp-python3.html (accessed on 18 June 2020).
33. Milde, D. Benchmark Numba vs. Cython. 2020. Available online: https://pybenchmarks.org/u64q/numba.php (accessed on 18 June 2020).
34. Gorelick, M.; Ozsvald, I. *High Performance Python: Practical Performant Programming for Humans*; O'Reilly Media, Inc.: Newton, MA, USA, 2014.
35. Ringle, E. Portaudio. 2020. Available online: https://github.com/EddieRingle/portaudio/blob/master/examples/ (accessed on 18 June 2020).
36. Zölzer, U. *DAFX: Digital Audio Effects*; John Wiley & Sons: Hoboken, NJ, USA, 2011.
37. Pedregosa, F. Memory Profiler Code Repository. 2020. Available online: https://github.com/pythonprofilers/memory_profiler (accessed on 18 June 2020).
38. Fontana, F.; Civolani, M. Modeling of the EMS VCS3 Voltage-Controlled Filter as a Nonlinear Filter Network. *IEEE Trans. Audio Speech Lang. Process.* **2010**, *18*, 760–772. [CrossRef]