# ScPy: A SuperCollider Extension for Performing Numerical Computation via Embedded Python

Noah Weninger Abram Hindle

August 28, 2016

**Abstract**

SuperCollider, a language for sound synthesis and algorithmic composition of audio, supports a wide range of synthesis, effect and analysis algorithms. However, available operations are limited to those implemented explicitly as Unit Generators (UGens). Since UGens are written in C/C++ and loaded as plugins during the SuperCollider server boot process, it is impossible to live code UGens, which limits the user to creating sound as a composition of existing UGens during a performance. Many of the vector operations required for efficiently creating complex audio effects are notably missing or tedious to use. To overcome this, we present ScPy, a UGen which embeds Python within SuperCollider to enable the use of the optimized vector operations provided by the NumPy library.

## 1 Introduction

Although a number of open-source projects which interface other languages with SuperCollider exist [17, 20, 22], they are mostly SuperCollider clients, which have features to control synths and send other automation messages. One notable exception is OctaveSC [11], which embeds GNU Octave within SuperCollider. However, it is designed for performing operations on control rate data matrices and suffers from performance issues which prevent it from being used on audio rate data.

One of the goals of this project is to enable more flexible experimentation with FFT and phase vocoder based operations. SuperCollider contains a total of 32 built in spectral operations, which provide the ability to produce many simple effects. There are additionally some extra operations available as user created extensions. However, many conceivable effects are impossible, including basic phase vocoder effects such as pitch shifting. Some examples of novel effects can be found in Section 4.

ScPy enables users to overcome these limitations. By embedding the Python programming language within SuperCollider, it is possible to do numerical processing operations which would be too slow to perform in real time with SuperCollider alone. Through access to the NumPy library, previously difficult

or impossible phase vocoder operations become trivial. Although our usage of this library for testing purposes has mostly focused on FFT based operations, support exists for passing arbitrary data buffers to Python, which can be used to process numerical data for many other purposes as well.

ScPy is developed and maintained as an open-source project under the GPLv3 license. The latest code, documentation, and issues are available on GitHub[1]. Contributions to the project are welcome.

# 2 Literature Review

With any New Interface for Musical Expression (NIME), it is important to evaluate how it performs and is interacted with across various dimensions. Birnbaum et al. [1] present a new model for evaluating musical instruments by determining the position of the instrument along 7 axes: *Required Expertise*, *Musical Control*, *Feedback Modalities*, *Degrees of Freedom*, *Inter-actors*, *Distribution in Space*, and the *Role of Sound*. Some of these axes represent a continuous spectrum, where as others have discrete steps. These dimensions are arranged to form a heptagon in the 2D plane, where each corner of the shape is placed at a distance to the center proportional to the score of the instrument being evaluated along the corresponding axis. The specific layout of the dimensions used in the paper additionally has properties which allow instruments of different classes to be easily distinguished. Plots of instruments tend to occupy the right side of the graph, where as installations tend to occupy the left. Although this model performs well on the instruments evaluated in the paper, it is based on subjective evaluation and it is likely that it does not adequately cover the space of all possible instruments.

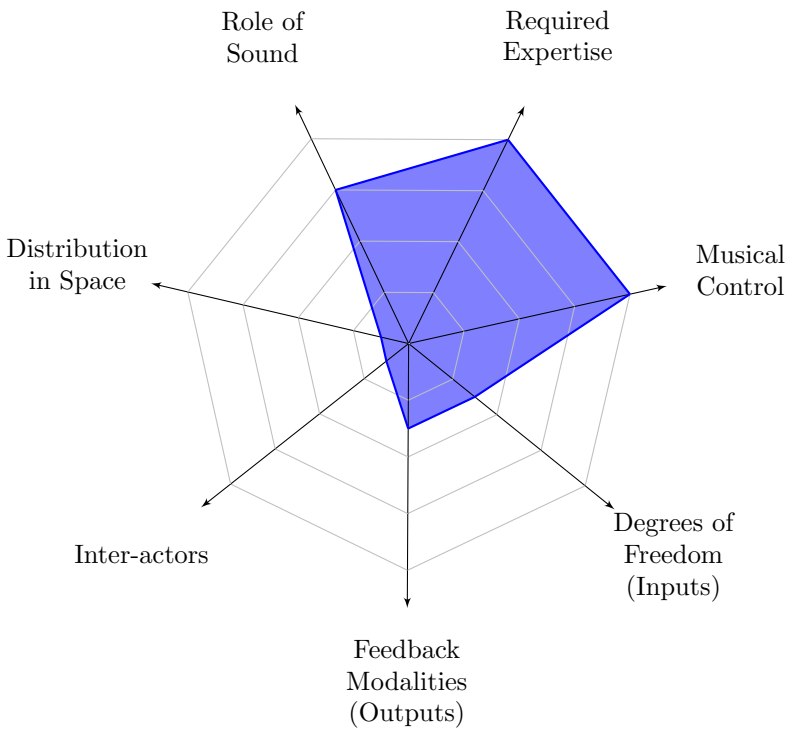
ScPy is unlike many NIMEs because it does not have the properties of any traditional instrument. However, it can still be considered a NIME because it ultimately is a tool for musical expression, just with an unconventional method of interaction. A plot of ScPy in the 7-dimensional space developed by Birnbaum et al. can be seen in Figure 1.

The phase vocoder is an essential part of spectral audio manipulation. De Götzen et al. [8] present an open source Matlab implementation along with a number of tricks to make the implementation faster or simpler. The implementation is split into two functions, `pv_analyze` and `pv_synthesize`. These functions operate as a tightly coupled set due to the phase information maintained between frames. One of the techniques presented is to perform a `fftshift` operation in order to simplify the phase relationship between neighbouring bins. Another is to zero pad frames: by adding extra zeros on each side of a frame, a larger FFT size can be used, producing better accuracy while maintaining good time resolution.

The code in this paper was used as a reference for the phase vocoder implementation in ScPy. Additionally, some of the extra techniques mentioned could improve the quality or flexibility of the implementation in the future.

[1] `https://github.com/nwoeanhinnogaehr/ScPy`

Figure 1: NIME dimension space analysis for ScPy.




Keiler and Sylvain [12] present a survey of 6 techniques for accurate analysis of stationary sounds. The first method, plain FFT, uses a Hann window and assumes the frequencies of the sinusoids at centered within each bin. Parabolic interpolation notices that the shape of a magnitude peak looks like a parabola, and thus attempts to fit a parabola to the magnitudes of bins nearby each local maxima, yielding more accurate results than a plain FFT. The triangle algorithm is similar to the parabolic interpolation method but fits a triangular function instead of a parabola. Spectral reassignment uses the derivative of the analysis window to estimate the true values of the bins as the center of gravity of the energy of each bin. The derivative algorithm also uses the derivative of the analysis window to determine both a more accurate frequency and amplitude. It can be further extended to use the first nth derivatives. The phase vocoder uses the phase information of two successive FFTs to improve the frequency resolution by calculating the phase derivative. Comparison of the techniques found that the spectral reassignment, derivative algorithm, and phase vocoder methods had the best performance. Surprisingly, those three techniques produced almost identical results, suggesting that they may have equivalent theoretical

basis.

The two other techniques that performed similarly to the phase vocoder will be interesting to analyze in more detail since they may present a solution to the *phasiness* problem of the phase vocoder due their lack of persistent state. However, they require extra processing so it is not clear if the trade-off will be worthwhile.

Audio morphing is defined as a gradual change from one sound to the other. While morphing can be achieved to some extent simply by cross fading two signals, perceptually this sounds simply like one sound is becoming quieter as the other is becoming louder. Sethares and Bucklew [21] propose density and kernel based techniques that aim to provide a more convincing morph operation where there is no clear separation between the two sounds at any point during the morphing process. The method allows the use of different kernel functions to provide different kinds of morphing. These include *Cross-fade*, *Heat equation*, and *Harmonic integrity*. The morphing technique is implemented using a phase vocoder. These techniques could be implemented in Python as a future extension to ScPy.

Laroche and Dolson [14] present an improvement to the phase vocoder which aims to prevent the "phasiness" problem where different parts of a sound lose their phase coherence, as well as provide a significant decrease in computational cost. The paper recognizes that there are two types of phase coherence that must be preserved in any STFT based algorithm to ensure that the reproduced sound is without artifacts: horizontal and vertical. The basic phase vocoder algorithm is built to preserve horizontal phase coherence — but it does so at the expense of vertical phase coherence. To remedy this, they build on prior work by presenting two new phase-locking techniques.

Similar techniques [15] enable new phase vocoder techniques that enable exotic audio effects. The core of the techniques is a peak detection process followed by peak shifting. Since these new techniques operate on peaks, the standard phase vocoder algorithm can be simplified to not require phase unwrapping, reducing computational complexity. Additionally, they produce better quality output than the standard phase vocoder algorithm because of implicit phase-locking.

The phase vocoder currently implemented in ScPy uses the standard algorithm. Future experiments could implement many of the proposed modifications to the phase vocoder to improve sound quality.

Bradford et al. have published a number of works detailing the sliding DFT [5] and associated sliding phase vocoder [6]. These new algorithms are efficient implementations for the STFT with a hop size of a single sample. Although such a small hop size would be impossible to process in real time using current hardware and the STFT, the sliding DFT makes it possible using consumer GPU hardware [4]. The sliding phase vocoder additionally has a number of advantges over traditional implementations. Quality is greatly improved (at a high computational cost). Each analysis bin can hold any frequency, essentially making the spectrum more of a generic oscillator bank than a traditional spectrum. This simplifies effects such as pitch shifting to a simple vector mul-

tiplication in the frequency domain. Additionally, since the time resolution is so fine, effects that require time modulation such as FM or AM can now be performed in the frequency domain.

Using Theano, the sliding DFT and sliding phase vocoder could be implemented easily in ScPy. It remains unknown if it will be possible to run them in real time, but even if not this future extension could add the possibility to experiment with many novel effects.

Klingbeil [13] describes the techniques used in a high quality graphical spectral editing and resynthesis program called SPEAR. The FFT analysis is done taking into account the fundamental frequency of the sound to analyze, so that the transform can be optimized to accurately capture as many partials as possible. Spectral peaks are then selected and very quiet peaks are discarded. A linear prediction method is used to follow the peaks over time. The data structure for storage of spectral data involves division of time into a number of frames which keep track of which partials are active during that time, so that the search for a specific partial can be narrowed. User interface design choices are also covered in detail.

Although the interface and granularity of SPEAR differs significantly from those in ScPy, the techniques for high quality resynthesis presented should prove useful for improving the quality of the sound output.

While the Fourier transform is by far the most commonly used technique for spectral manipulation in digital audio effects, there exist numerous other transformations which may be useful as well to achieve certain audio effects. One such technique is the Mellin transform, described in a audio effects context by De Sena and Rocchesso [9]. The Mellin transform has a scale invariance property that is analogous to the shift invariance property of the Fourier transform. It can be computed quickly by applying two extra steps before doing a Fourier transform: exponential time warping and exponential multiplication. A few simple audio effects are presented using the Mellin transform, time stretching, filtering and random phase deviation (whisperization).

As the Mellin transform can be implemented simply as an extension to the Fourier transform, it should be fairly easy to add support for it in ScPy, and it will hopefully enable the use of some different audio effects than can be achieved easily with the Fourier transform. However, the paper mentions that a short-time Mellin transform has not been attempted, so some further work may need to be done in that area first.

Collins et al. [7] present an introduction to the field of live coding, where code to generate sound is written live during a performance, often on a laptop computer. Although the use graphical programming environments such as Reaktor, Pure Data, or MAX/MSP in live performances is in some sense live coding, the paper focuses on the use of text based programming environments. Many of the pros and cons of live coding are presented. Although live coding adds some very interesting new possibilities to live music performance, it is not without it's challenges. Case studies of two live coding environments are included: Slub and SuperCollider with the just in time library.

ScPy can be used for live coding of audio effects, similarly to how the en-

vironments studied in this paper can be used to live code synths and pattens. More details about how ScPy was designed to facilitate this can be found in Section 3.

Live coding presents an interesting contrast to contemporary music with many clear differences. Nilson [19] presents a review of the challenges of live coding practice. While some argue that live coding must be performed live in front of an audience, an alternate perspective suggests that time constraints are not necessary and that the practice can refer to any improvisational music created by programming. The practice of live coding is still quite young — it is mentioned that professional violinists will typically have practiced for 10000 hours, where as an email survey of live coders revealed most had dedicated about 100 hours. Of course, the prerequisite knowledge for live coding may be higher; many live coders have extensive programming experience. Another thing mentioned is the high level of abstraction offered by live coding creates a large startup time for producing meaningful output from a blank slate. Additionally, many exercises are presented for improving one's live coding ability.

The ideas in Nilson's paper shed light on some of the challenges faced in this project. One such challenge is the amount of writing required to produce an interesting result from a blank slate. We intend to minimise this by providing many effect primitives and composition mechanisms. However, it will be important to ensure the language is accessible to ensure it can be quickly learned to a level of expertise where lack of technical knowledge is not an impediment.

Domain-Specific Languages (DSLs) are small programming languages designed for use in a specific problem domain. They often lack general purpose programming constructs and have a syntax which is very concise for describing programs in their domain. Van Deursen et al. [23] present a survey of available DSL literature, along with the pros and cons of using a DSL, example DSLs, design methodology and implementation approaches. The biggest advantage of DSLs is that they enable expression of solutions in a way that is common for the problem domain. However, they can be costly to develop and difficult to design. Implementation techniques include the classical approach of interpretation or compilation, embedded languages / domain-specific libraries, preprocessing, or compiler extension. Additionally, the references of the paper each include a short summary.

This review reveals many of the possible options for the DSL used within the project. Although we decided not to design a DSL for our project, analyzing the design and implementation techniques presented here will be helpful for future work in this area.

As Python is an easily extendible general purpose language, it has applications in many scientific fields. Glover et al. [10] present a discussion of the use of Python for audio signal processing. They note that while audio processing libraries exist for many languages, scripting languages such as Python or MATLAB see widespread adoption because of the speed at which prototypes can be built. General purpose Python libraries such as NumPy, SciPy and Matplotlib include many essential functions for audio processing, such as file IO, Fourier transforms, signal processing and interpolation. There are additionally more

specific libraries such as Simpl, a sinusoidal modelling library, and Modal, an onset detection library. Any of these libraries can easily be used within ScPy simply by installing and importing them.

Another open-source Python library which could be used in ScPy is librosa [18] by McFee et al. librosa is a music signal analysis library designed to help transition music information retrieval (MIR) researchers into using Python, as well as make MIR techniques readily available to all Python programmers. librosa's core design choices and conventions are documented in detail to inspire future library authors to follow similarly good coding practice. As well as standard audio analysis functions, librosa also includes support for effects and audio IO.

librosa has quite an extensive list of features and would likely cover all of the audio related functionality which is missing from NumPy. It can be used with ScPy simply by installing and importing it.

The ixi lang [17] is a live coding language. It is implemented as an interpreter built in SuperCollider, a choice which allows for SuperCollider and ixi lang code to be mixed together in a similar manner to ScPy. The ixi lang is built to reduce the startup time in a live coding performance and enable faster musical experimentation through syntax more concise than SuperCollider. The paper also contains user feedback on the usability of the ixi lang.

Since the ixi lang is similar to ScPy in many ways, many of the design choices made in ixi lang are also appropriate in ScPy. Additionally, lessons learned from the user feedback for ixi lang could potentially be used to improve usability of ScPy.

# 3 Methodology

During the initial planning stage of the project, we wanted to essentially provide a set of fundamental operations on spectral data which could be composed together to perform any conceivable effect. It was also a requirement that these operations would be syntactically concise to enable easy and fast experimentation. Ideally, the system would be able to handle complex effects in real-time.

The choice of SuperCollider was largely because it is one of the most widely used audio programming environments. Using SuperCollider also enabled the reuse of many parts of its codebase. Since it already has STFT based operations, it was a good target for extension purposes in that area.

Although designing a domain-specific language for this task was briefly considered, we eventually decided to use Python for a number of reasons. First, Python includes an excellent C API which makes embedding it within other languages trivial in comparison to many other options. Second, Python's syntax is easily readable and concise — it clearly adheres to our goals. Finally, Python has a massive number of community built libraries for performant scientific and numerical computing. With a few small exceptions, which are clearly specific to spectral data (such as the phase vocoder), these libraries contained every one of the fundamental operations we could think of.

It may seem as though embedding an interpreted language within an interpreted language could offer no performance advantage. There is certainly some performance hit when switching languages and transferring data. However, the biggest advantage comes not strictly from performance but from the diversity of operations which become available with highly optimized implementations. NumPy is used internally by ScPy for handling data arrays, so the entire NumPy library of mathematical functions comes at no cost. Any other Python library can easily be imported as well. Limited experiments have been done with the SciPy library. It would also be possible to use Theano to do processing on the GPU for extra performance. Implementing an equivalent to these popular Python libraries in pure SuperCollider is certainly possible, but it would require a very large amount of work and would see no adoption outside of the audio computing community.

In order to facilitate the use of ScPy in live coding performances, many errors are ignored or handled in a way that can be easily recovered from. In particular, floating point errors such as division by zero or other invalid operations are ignored. Special values such as infinity or NaN are replaced with zeros. This is done because SuperCollider has poor handling of these special values and in some cases will crash if they are found. Python exceptions are printed to the SuperCollider post window, where they can easily be examined and corrected by the user.

# 4 Usage

Use of ScPy is through the classes `Py` and `PyOnce`. `Py` is used for processing real-time data, and `PyOnce` is used for doing initialization or other one time processing. Both have one required argument, a block of Python code. A map of variable names may optionally be provided to bind SuperCollider variables to Python variables. Additionally, there is an optional `DoneAction` argument which specifies an action to occur in SuperCollider after the Python code has finished executing.

```
1 (PyOnce("
2     # comments in Python have different syntax!
3     print('Hello from Python!')
4     print('x is', x)
5 ", (x: 62)))
```

Listing 1: Hello world with ScPy.

When Listing 2 is run, an anonymous synth is created which will take in audio from an external source, perform an STFT, process the spectral data with ScPy, perform an inverse STFT, then finally output the resulting audio. Lines 8–9 contain Python code defining the processing function where operations are to be inserted. Further examples will replace only those lines. Line 16 contains

```
(s.waitForBoot {
    var buf = { Buffer.alloc(s, 512) }.dup;
    var hop = 1/4;

    s.freeAll; // stop previous versions of the synth

    PyOnce("
        def fn(x):
            return x
    ", (hop:hop));

    {
        var in = AudioIn.ar([1, 2]);
        var x = FFT(buf.collect(_.bufnum), in, hop);
        Py("
            out(x, fn(array(x)))
        ", (x:x, time:Sweep.kr));
        Out.ar(0, IFFT(x));
    }.play(s);
})
```

Listing 2: SuperCollider boilerplate for no-op FFT effect with ScPy.

the Python code to convert the spectrum into a NumPy array, process it, then write back the processed version.

A special function, `out`, is provided to mutate data in SuperCollider buffers. It takes a variable which is bound to a `Buffer` or `FFT` object and a new value. The buffer is modified in place to hold the new data.

Complex numbers are used extensively, even to represent non-Cartesian data such as polar coordinates or phase vocoder bins in ScPy. In polar form, the real and imaginary parts represent magnitude and phase, respectively. Similarly for the phase vocoder, real and imaginary parts represent magnitude and frequency. This is done to take advantage of the limited data types available in NumPy and relative convenience of working with complex numbers as opposed to tuples or classes. Future work could extend NumPy to give more descriptive names to these data fields.

# 5 Implementation

ScPy is written mostly in C++, but with a small SuperCollider class library to connect the C++ back-end, and a small Python library that provides some useful operations. Due to differences between these languages, connecting them was somewhat awkward in some cases.

Since UGens may only be used as part of a synth, `PyOnce` is provided as

```
pv = PhaseVocoder(hop)

def fn(x):
    x = pv.forward(x)
    idx = indices(x.shape)[1]
    x = pv.shift(x, lambda freq:
        freq * (0.8 + mod(-time + 0.1*idx, 10)*0.045))
    x = pv.backward(x)
    return x
```

Listing 3: A novel effect to transform any sound into a falling Shepard tone. This example makes use of the phase vocoder `shift` operation, which applies a function to frequency, then reorganizes the spectrum to move the new frequencies into the appropriate bins.

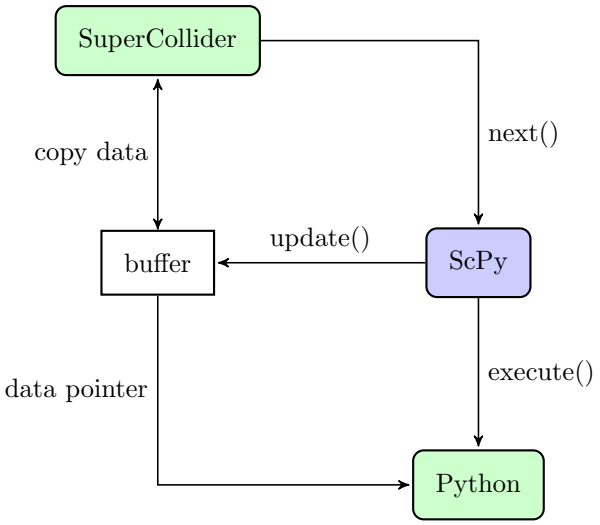
```
pv = PhaseVocoder(hop)

def fn(x):
    x = pv.forward(x)
    x = pv.to_bin_offset(x)
    x.imag = -x.imag # imaginary part holds frequency
    x = pv.from_bin_offset(x)
    x = pv.backward(x)
    return x
```

Listing 4: An effect which inverts the frequency of each analysis bin across the center of the bin, creating an unusual detuning effect.

a wrapper which hides this detail from the user and allows the execution of Python code outside of a synth. It is useful for doing initialization work. Since the current implementation executes Python code in a single global namespace, a typical usage patten is for state to be defined in a `PyOnce` block which is later accessed and modified in a `Py` block.

One issue involves the process of passing data between SuperCollider and C++. Since UGens are designed to work primarily with data streams, adding support for passing in many types of data came with a number of challenges. Essentially, all data must be serialized into an array of floats. Strings must be converted to an array of ASCII character codes, and prepended with their length. Arguments passed through to the Python code can have one of many types, and therefore type information must be encoded as well. However, just passing the type of a variable is not enough to know what we can do with it, since we also want to know if it inherits from a class we can use. The encoding and decoding implementation is unfortunately fairly complex, but creating a UGen with inputs this complicated was likely not considered during the design of SuperCollider, so some friction was to be expected in this area.

Figure 2: A high level overview of data flow and processing.




Since `FFT` objects in SuperCollider operate on a single channel only, they must be placed in `Array` objects to support multichannel audio. `Array`s can hold objects of any type, so it is natural to convert them to Python `list` objects, which have similar properties. However, this presents an interesting problem: multichannel `FFT` objects are then converted to `list`s of `ndarray` objects in Python. These are a bit more awkward to work with than basic `ndarray` objects, but can easily be converted using a call to the `array` function in Python. This can be seen in practice on line 16 of Listing 2. Although it would be possible to do this conversion automatically, it is left out for the purpose of not introducing a feature simply to work around an incomplete implementation of `FFT` objects in SuperCollider. The current conversions will continue to function as intended if `FFT` objects are given multichannel support in the future.

Control UGens only provide a single floating point value per cycle so it would seem natural to convert them to a Python `float`, but in ScPy they are instead transferred to Python in a one element `ndarray`. This is because Python provides no numerical types with interior mutability. In practice, NumPy accepts a one element `ndarray` almost anywhere a `float` is accepted, so this detail is not an annoyance.

As shown in Listing 3 and Listing 4, ScPy includes a phase vocoder implemented in Python for performing frequency transformations. Interestingly, although all of SuperCollider's FFT operations are referred to as phase vocoder operations, there is actually no implementation of a phase vocoder to be found in the SuperCollider code base at the time of writing. It is likely that a phase vocoder was once intended to be implemented, but the 32 so-called phase

Table 1: Type correspondence between SuperCollider and Python

| SuperCollider type | Python type |
|---|---|
| `Float` | `float` |
| (subclass of) `UGen` | `ndarray` of `float` |
| `Buffer` | `ndarray` of `float` |
| `FFT` | `ndarray` of `complex` |
| `Array` | `list` |

vocoder effects are simply transformations to the Cartesian or polar representations of the spectrum.

# 6 Future Work

At the time of writing, ScPy only supports handling audio rate data via SuperCollider's `Buffer` objects. In order to make some use cases more ergonomic, ScPy could certainly be extended to support raw audio rate input. For example, this would make time domain audio data even simpler to work with than frequency domain data is currently. It would also enable SuperCollider's FFT UGen to be easily re-implemented in Python for greater flexibility.

Support for more SuperCollider types, such as events or strings, could be added to open up ScPy to more use cases. Adding support for more types should be trivial in the existing code base.

Multi-core processing support would allow for many more sounds to be processed in real-time. Currently, all processing is done on a single thread, even with many UGens running at once. Using the supernova server [2] may be helpful, however it has not yet been tested and ScPy may require some modifications to be usable with supernova.

Creating a more generic extension which supports many languages would have a number of advantages. Although extending SuperCollider was the focus of this project, similar improvements could be made to the CSound [3, 16], Chuck [24] or Faust [20] environments as well. It may be desirable to completely separate the embedded language from the host, in order to allow easier interoperation and a cleaner implementation.

Throughout Section 2, articles are discussed which contain processing techniques that could be used in ScPy. Although implementations of many of these techniques are likely to become a part of ScPy in the future, there is no obstacle to the reader implementing them as Python libraries usable by ScPy or even as open-source contributions to the project.

# References


[1] David Birnbaum et al. "Towards a dimension space for musical devices". In: *Proceedings of the 2005 conference on New interfaces for musical expression*. National University of Singapore. 2005, pp. 192–195.

[2] Tim Blechmann. "supernova, a multiprocessor-aware synthesis server for SuperCollider". In: *Proceedings of the Linux Audio Conference*. 2005, pp. 141–146.

[3] Richard Charles Boulanger. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT press, 2000.

[4] Russell Bradford, Richard Dobson, et al. *Real-time sliding phase vocoder using a commodity GPU*. University of Huddersfield and ICMA, 2011.

[5] Russell Bradford, Richard Dobson, et al. "Sliding is Smoother than Jumping". In: *International Computer Music Conference 2005 (ICMC 2005)*. University of Bath. 2005, pp. 287–290.

[6] Russell Bradford, Richard Dobson, et al. "The sliding phase vocoder". In: (2007).

[7] Nick Collins et al. "Live coding in laptop performance". In: *Organised sound* 8.3 (2003), pp. 321–330.

[8] Amalia De Götzen, Nicola Bernardini, and Daniel Arfib. "Traditional implementations of a phase vocoder: the tricks of the trade". In: *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00), Verona, Italy*. 2000.

[9] Antonio De Sena and Davide Rocchesso. "A fast Mellin transform with applications in DAFx". In: *Proc. of the 7th Int. Conference on Digital Audio Effects (DAFx'04)*. 2004, pp. 65–69.

[10] John C Glover, Victor Lazzarini, and Joseph Timoney. "Python for audio signal processing". In: (2011).

[11] Thomas Hermann. *OctaveSC*. `http://www.sonification.de/projects/sc3/index.shtml`. 2006.

[12] Florian Keiler and Sylvain Marchand. "Survey on extraction of sinusoids in stationary sounds". In: *Digital Audio Effects (DAFx) Conference*. 2002, pp. 51–58.

[13] Michael Klingbeil. "Software for spectral analysis, editing, and synthesis". In: *Proceedings of the International Computer Music Conference*. 2005, pp. 107–110.

[14] Jean Laroche and Mark Dolson. "Improved phase vocoder time-scale modification of audio". In: *IEEE Transactions on Speech and Audio processing* 7.3 (1999), pp. 323–332.

[15] Jean Laroche and Mark Dolson. "New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects". In: *Applications of Signal Processing to Audio and Acoustics, 1999 IEEE Workshop on*. IEEE. 1999, pp. 91–94.

[16] Victor Lazzarini. "Extensions to the Csound language: from user-defined to plugin opcodes and beyond". In: *Proc. of the 3rd Linux Audio Conf.* Citeseer. 2005, pp. 13–20.

[17] Thor Magnusson. "ixi lang: a SuperCollider parasite for live coding". In: *Proceedings of International Computer Music Conference*. University of Huddersfield. 2011, pp. 503–506.

[18] Brian McFee et al. "librosa: Audio and music signal analysis in python". In: *Proceedings of the 14th Python in Science Conference*. 2015.

[19] Click Nilson. "Live coding practice". In: *Proceedings of the 7th international conference on New interfaces for musical expression*. ACM. 2007, pp. 112–117.

[20] Yann Orlarey, Dominique Fober, and Stéphane Letz. "Faust: an efficient functional approach to DSP programming". In: *New Computational Paradigms for Computer Music* 290 (2009).

[21] William Sethares and James Bucklew. "Kernel Techniques for Audio Morphing". In: (2012).

[22] *Systems interfacing with SC*. `http://supercollider.sourceforge.net/wiki/index.php/Systems_interfacing_with_SC`. July 2011.

[23] Arie Van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography." In: *Sigplan Notices* 35.6 (2000), pp. 26–36.

[24] Ge Wang. *The chuck audio programming language. a strongly-timed and on-the-fly environ/mentality*. Princeton University, 2008.