SAMPLE: a Python Package for the Spectral Analysis of Modal Sounds

Marco Tiraboschi **Federico Avanzini**
Laboratory of Music Informatics (LIM)
Department of Computer Science
University of Milan, Italy
marco.tiraboschi,federico.avanzini@unimi.it

ABSTRACT

We present SAMPLE, a Python package of tools for spectral analysis and modal parameters estimate. The core of the package is an implementation of the “Spectral Analysis for Modal Parameters Linear Estimate” (SAMPLE) algorithm. This includes a custom implementation of a Sinusoidal Analysis algorithm based on Spectral Modelling Synthesis. Our custom implementation is specifically designed for modal tracking. We also included utilities for automatically tuning the algorithm parameters, using a Bayesian optimization method based on Gaussian Processes. For this purpose, we implemented efficient routines for computing perceptual audio representations for loss functions, such as the multiscale-spectrogram, the mel-spectrogram and the cochleagram. The package also comes with a Graphical User Interface, which allows to load and trim audio inputs, set the algorithm parameters, run the algorithm, listen to a resynthesis of the input, and export the results. The GUI is distributed both as an extra for the Python package and as a standalone executable.

1. INTRODUCTION

Modal synthesis [1] is an approach to sound synthesis based on physical or pseudo-physical models. It is used to generate sounds as results of interactions between objects, such as impacts and frictions [2], and it is becoming more and more relevant with the recent advancements in audio for videogames and extended-reality.

In many cases, technical issues stand between the sound designer and physical models. In modal synthesis, many are the parameters that the sound designer must finely tune: they have to decide dozens of modal frequencies, amplitudes, and decay times.

We introduce the SAMPLE Python package. It includes the implementation of several algorithms aimed at the automatic estimate of modal parameters from audio examples, including SAMPLE [3] and BeatsDROP.

It is available via the Python Package Index, on GitHub and on Zenodo [4]. In can be installed as a Python package, or used via the GUI as a standalone executable.

Copyright: ©2022 Marco Tiraboschi et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

The package is thoroughly tested with unit tests using the Python builtin module *unittest* [5]. Code coverage is 100% on all the code base, excluding the GUI module.

2. SAMPLE

Our main goal for this package was to implement the “Spectral Analysis for Modal Parameters Linear Estimate” (SAMPLE) algorithm [3] in such a way that it would be easily available, distributable, and usable.

Most components of the package gravitate around the *SAMPLE* class. We designed this class following the patterns of the *scikit-learn* API [6, 7]. Every SAMPLE object includes a sinusoidal model and a regressor, that can be either a linear regressor or a semi-linear regressor.

Instances of the SAMPLE class can be fitted to audio examples to estimate the modal amplitudes, frequencies and decay times of the recorded objects. The input audio should be a sound generated hitting the object.

2.1 Sinusoidal Model

The *SinusoidalModel* class implements the analysis algorithm of *Sinusoidal Modelling Synthesis* (SMS) [8, 9, 10]. This includes both the spectral peak detection and the spectral peak continuation algorithms. Processing audio backwards is also supported. This is common in analysis techniques for additive synthesis [11].

The ModalModel class inherits from SinusoidalModel and adds custom functionalities for modal sounds. During the analysis step a ModalModel can check additional constraints. A ModalModel can discard trajectories if the modal frequencies are out of the desired boundaries, e.g. infrasounds or ultrasounds. It will, also, discard ill-behaved partials whose magnitude increases with time, instead of decreasing. When a trajectory stops, its starting amplitude is estimated and it can be discarded if it is too quiet. Finally, the stopped trajectory can be merged with a previous trajectory if their frequencies are similar.

2.2 Hinge Regression

The *HingeRegression* class implements the regression algorithm for the modal amplitudes and decay times. It is based on the Rectangular Trust Region Dogleg Approach (DogBox) [12] for non-linear least-squares optimization, implemented in SciPy [13].

HingeRegression fits an amplitude trajectory output by the sinusoidal model to a hinge function $h_{k,q,\alpha}(t)$, which

is a function that is linear for $t < \alpha$ and then continues as a constant.

$$h_{k,q,\alpha}(t) = k \cdot \min(t, \alpha) + q \tag{1}$$

HingeRegression improves the parameters estimate with respect to linear least-squares when the noise floor is high. In this case, linear regression is biased towards lower values for both the amplitude and the decay time [3].

2.3 BeatsDROP

The package also includes the implementation of the "Beats Duality for the Resolution Of Partials" (BeatsDROP) algorithm. The *DualBeatRegression* class implements a regression model for two modal partials with very similar frequencies. In this case, the sinusoidal model would not find two different trajectories, but it would output a single trajectory, in which the two partials interfere.

DualBeatRegression fits the modal frequencies, amplitudes and decay times of both partials to the sinusoidal model trajectory. The loss function includes the differences for both the amplitude and the frequency of the trajectory. This algorithm exploits the fact that beats of uneven amplitudes produce modulations of both amplitude and frequency. This regessor can be used instead of HingeRegression, to find two sets of modal parameters from one trajectory.

3. GRAPHICAL USER INTERFACE

We designed a Graphical user Interface to enable non-developers to access to our methods. It allows to load an audio file, apply the SAMPLE model, and export the results as a JSON file. Figure 1 displays a series of screenshots of the GUI in action.

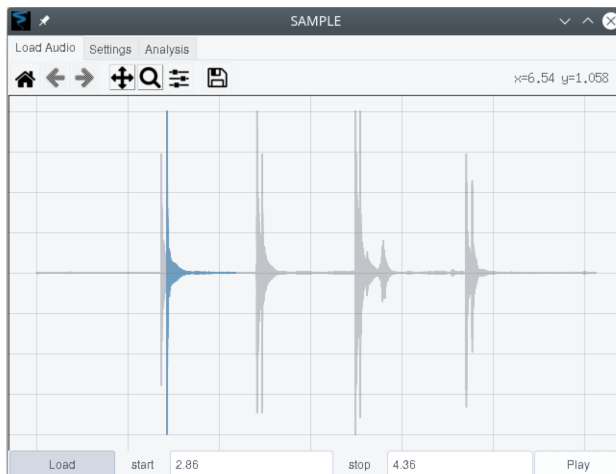
3.1 Load Audio

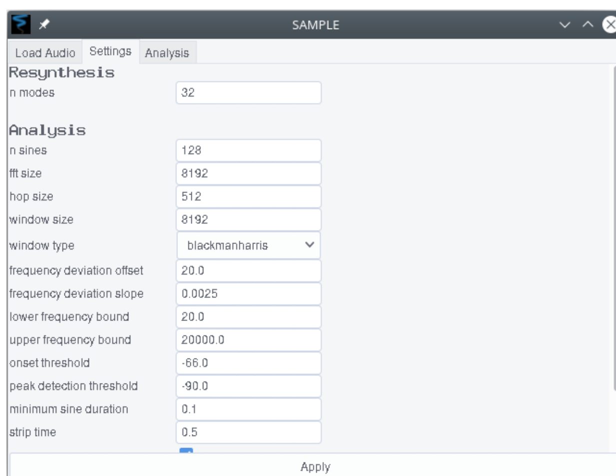
The first pane of the GUI is the *"Load Audio"* pane, which allows the user to load and trim a target audio file.

The *"Load"* button opens a file browser for selecting the audio file to load. After loading a file, the GUI displays the audio waveform. We initialize the region-of-interest (ROI) using an algorithm for onset detection [14] and highlight it in a different color.

The SAMPLE algorithm will only process audio in the selected ROI. The users can adjust the ROI by specifying the desired *"start"* and *"stop"* timestamps in the text input boxes. Alternatively, they can modify the start and the stop timestamps by clicking on the waveform display: when the user clicks down on the display the GUI decides whether to affect the start or the stop timestamp (depending on which one is closer), while the specific timestamp value is determined on mouse release. The user can zoom in and out the waveform display for a finer selection of the ROI.

The user can listen to the currently selected ROI of the audio by clicking on the *"Play"* button. We handle audio playback for multiple platforms using PyGame [15].





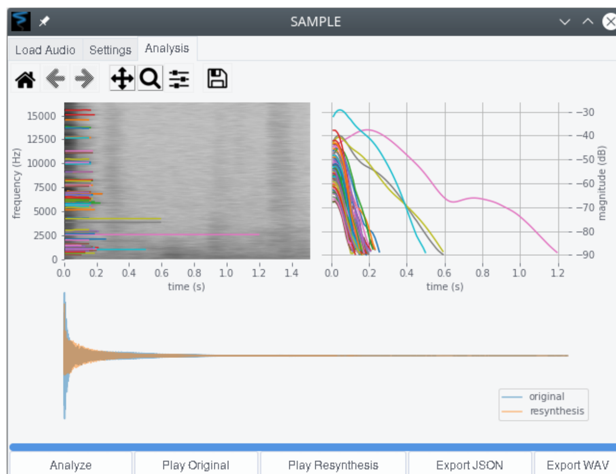


Figure 1. The three panes of the GUI, as they render on Ubuntu 20.04 with the *arc* theme. In the *"Load Audio"* pane, the user can load, trim and play the audio input. In the *"Settings"* pane, the user can tweak the algorithm parameters. In the *"Analysis"* pane, the user can run the algorithm, play an audio resynthesis and export the estimated modal parameters.

3.2 Settings

The second pane of the GUI is the *"Settings"* pane, which allows the user to specify most of the parameters for the SAMPLE algorithm.

- `n modes` controls the maximum number of modes that will be resynthesized by the model. Only the modes with the highest energy will be synthesized. The user can use this parameter to explicitly set the *level of audio detail* (LOAD). This parameter only affects the resynthesis, and not the analysis.
- `n sines` controls the maximum number of sinusoidal peaks that SAMPLE will track in each STFT frame.
- `fft size` is the dimension of the FFT for each frame. The user input is automatically rounded up to the next power of 2.
- `hop size` is the distance between STFT frames on the time axis, in samples.
- `window size` is the dimension of the STFT analysis window, in samples. It doesn't have to be a power of 2 and it is capped at the current `fft size`.
- `window type` is the name of the window function. It can be any of the window functions supported by SciPy [13] which doesn't need any more arguments than the window size.
- `frequency deviation offset` is the threshold for the peak continuation at 0 Hz, in hertz. Peaks at a frequency difference below the threshold will be considered the continuation of one another.
- `frequency deviation slope` determines the increment of the threshold for the peak continuation with frequency. The threshold at frequency f is

$$\tau(f) = \tau_{\text{offset}} + \tau_{\text{slope}} \cdot f \tag{2}$$

- `lower frequency bound` is the minimum frequency for accepting a trajectory, in hertz.
- `upper frequency bound` is the maximum frequency for accepting a trajectory, in hertz.
- `onset threshold` is the initial amplitude threshold for accepting a trajectory, in dBFS. If the linear fit of the amplitude trajectory returns an intercept below the threshold, the trajectory is discarded.
- `peak detection threshold` is the threshold for detecting a peak in the STFT, in dBFS.
- `minimum sine duration` is the minimum trajectory length for accepting a trajectory, in seconds.
- `strip time` is the maximum onset time for accepting a trajectory, in seconds. Trajectories starting later than this time will be discarded.
- `reverse` controls whether to process the audio backwards or not.
- `gui theme` is the name of the GUI theme, as defined in `ttkthemes` [16]. The new theme is applied the next time the GUI starts. When the theme is changed, the user is prompted whether they want to reload the GUI or not.

3.3 Analysis

The last pane of the GUI is the *"Analysis"* pane. The *"Analyze"* button runs the SAMPLE algorithm on the target audio. A progress bar displays how much of the target audio has been analyzed by the algorithm.

Once the analysis process has finished, the GUI populates the visual display. The top left subplot shows the spectrogram of the target audio in grayscale. The frequency trajectories are overlayed in different colors. The top right subplot displays the amplitude trajectories. Finally, the bottom subplot shows the waveform of the target and of the resynthesized audio. The resynthesis is a simple additive synthesis, where all partials have exponential amplitude envelopes.

$$\hat{x}(t) = \sum_{i=1}^{\text{n_modes}} a_i \exp\left(-\frac{2}{d_i}t\right) \cos\left(2\pi\nu_i t + \phi_i\right) \tag{3}$$

Where a_i is the modal amplitude, d_i the decay time, and ν_i the modal frequency of the i-th mode. Phase values ϕ_i can be chosen arbitrarily, at random, or set to zero.

The user can listen and compare to the target audio and the resynthesis, by clicking on the *"Play Original"* button and *"Play Resynthesis"* buttons, respectively.

The *"Export JSON"* button saves the inferred modal parameters as a JSON file. The *"Export WAV"* button saves the resynthesised audio as a WAV file. A file browser opens to make the user choose where to save the file.

4. AUTOMATIC OPTIMIZATION

The SAMPLE algorithm has many hyperparameters and it is not always obvious a-priori what the best values might be for the specific target sound. For this reason, we implemented a module to apply automatic hyperparameter optimization to SAMPLE. Figure 2 shows the loss function values estimated by the minimization process while optimizing the analysis window size, the number of peaks, the peak threshold and the minimum trajectory duration.

The class *SAMPLEOptimizer* allows to define the optimization problem with great flexibility. The developer can specify the value for any number of hyperparameters, which will not be optimized. A good practise is to specify a maximum number of modes to use in resynthesis, to avoid overfitting, because having a high number of modes often causes noise to be modelled as sinusoids even for a small decrease in the loss function. The specific value depends on the complexity of the input sound, but we generally obtained satisfactory results setting the number of modes between 16 and 128.

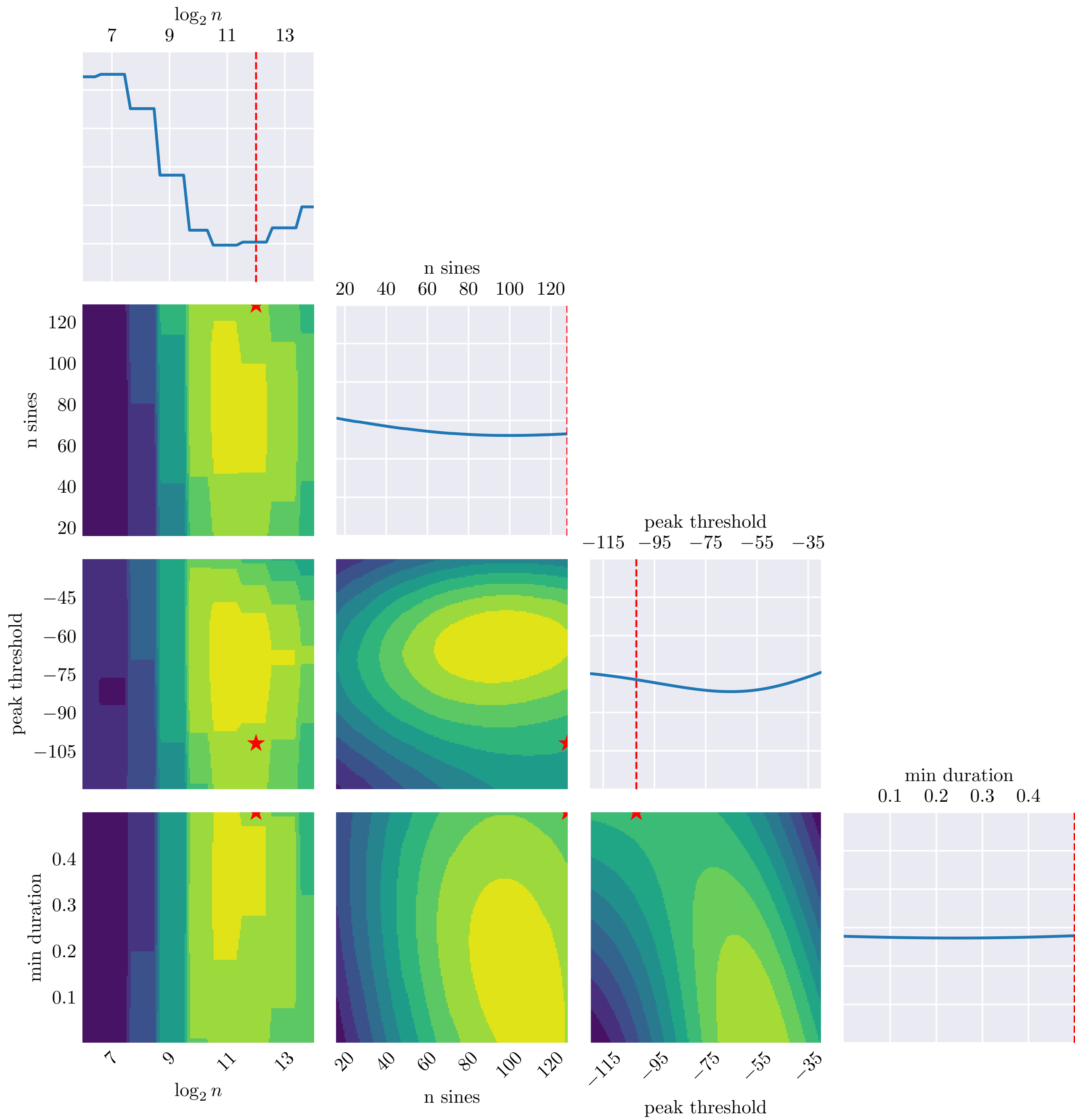


Figure 2. Partial Dependence plots for SAMPLE hyperparameter optimization on a synthetic modal sound. They show the dependence of the cochleagram loss function on each individual or pair of hyperparameters. A partial dependence plot (PDP) shows “the dependence between the target response and a set of input features of interest, marginalizing over the values of all other input features” [6]. Visualizing a function of more than two inputs would be impossible, so PDPs visualize different *views* of the same function. The line plots on the diagonal show the dependence of the loss function on one hyperparameter at a time. The contour plots in the lower triangle show the dependence of the loss function on two hyperparameters at a time.

Alternatively, the developer can specify hyperparameters as Scikit-Optimize [17] dimensions. These hyperparameters will be optimized by a Bayesian optimization algorithm based on Gaussian Processes (skopt.gp_minimize).

Some arguments of the *SAMPLE* class are not suitable for optimization. For example, the analysis window is a high-dimensional vector: optimizing all the values in the vector would be absurd. Instead, we would like to find the best window size and type. A remap function can be specified to transform hyperparameters into valid arguments for the constructor. The default remap function allows to specify the logarithm of the FFT size, to define the window with a name and a window size (as a fraction of the FFT size) and to specify the window overlap as a fraction of the size, instead of specifying the hop size in samples.

The loss function defines the optimization objective. It accepts the two arrays of original samples and resynthesized samples and returns a score, where lower is better. By default, it is a multi-scale spectral loss.

4.1 Loss Functions

We implemented several time-frequency audio representations with the aim of defining perceptually-based loss functions.

The first loss function we implemented is the multi-scale spectral loss. This is defined as the weighted sum of the L_p distances (usually with $p = 1$) between the spectrograms of the two sounds, with linear amplitude and in decibel, for different window sizes [18]. Given a set of spectrogram functions with different resolutions Σ, the loss function is defined as

$$\begin{aligned}\ell_{\Sigma,\alpha,p}(x, y) := \sum_{S_i\in\Sigma} \left\|S_i(x) - S_i(y)\right\|_p + \\ + \alpha \sum_{S_i\in\Sigma} \left\|\log S_i(x) - \log S_i(y)\right\|_p\end{aligned} \tag{4}$$

To make this function more efficient, we optionally allow developers to specify a number of jobs or a process pool to compute the different spectrograms in parallel.

We also implemented a fast cochleagram function. A cochleagram is a time-frequency representation whose time and frequency resolution change with frequency and mimic human perception [19]. Because of this property, there is no need to compute cochleagrams at different resolutions to define a perceptually-based loss function. We implemented the cochleagram as the convolutions of the audio signal with a set of impulse responses (IRs) of *gammatone filters*. Traditionally, after the convolution a simple non-linearity would be applied, such as half-wave rectification. We allow the option of convolving the signal with the analytic signals of the IRs: this is equivalent to computing the analytic signal of the output [20], but faster. In this case, the output is complex and the default non-linearity is the absolute value, so that the cochleagram is real.

The resulting cochleagram has the same sampling rate as the input. This is usually excessive, especially for the purpose of defining a loss function, and the cochleagram is downsampled to the desired sample rate. This is wasteful, because we compute a high-resolution output and then discard a great portion of it. We implemented a custom function for what is commonly known as a *strided* convolution: the output for a strided convolution with stride $s \in \mathbb{N}^+$ at sample i is the output for a convolution at sample $s \cdot i$.

$$(x *_s y)[i] := (x * y)[si] \tag{5}$$

For a sufficiently big stride, our implementation is faster than using the best of all of the convolution methods available in SciPy [13], which are: *direct* (convolution in the time-domain), *fft* (product in the frequency-domain), *auto* (automatically determines whether to use *direct* or *fft*), and *overlap-and-add* [21]. Figure 3 shows the average speed-up for different strides and input sizes.

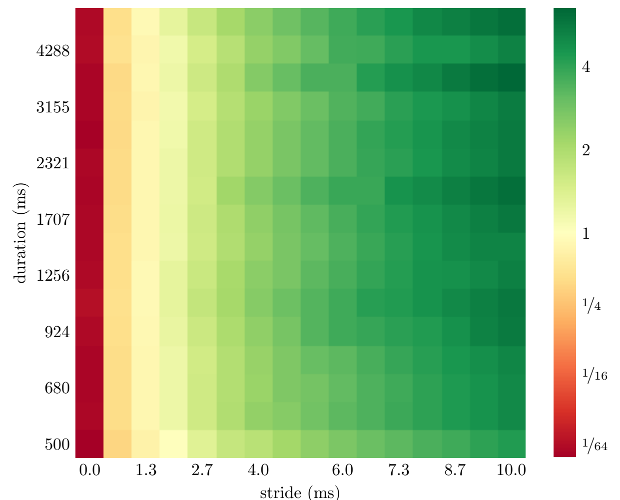


Figure 3. Speed-up of cochleagram method using the custom implementation for strided convolution. Speed-up is the ratio between the run-time of the custom implementation and the run-time of the best of four convolution methods (*direct*, *fft*, *auto*, and *overlap-and-add*). Speedup is evaluated for different input sizes and stride lengths (0 ms means 1 sample of stride) at 44.1 kHz averaging the run-times of 32 trials per case.

5. CONCLUSION

We presented the SAMPLE package for Python and its main components. The package is tested with unit tests with full code coverage.

We provided a general overview of the classes that implement the SAMPLE [3] and the BeatsDROP algorithms for the analysis of modal sounds.

We presented the Graphical User Interface for the SAMPLE method and how it constitutes a sufficient environment for preprocessing target audio files, configuring the algorithm hyperparameters, running the algorithm, inspecting and exporting the results.

Finally, we presented a module for interfacing SAMPLE with a Bayesian hyperparameter optimization function based on Gaussian Processes.

In the future, we plan to keep adding to the package any method related to SAMPLE that we may develop.

6. REFERENCES

[1] J.-M. Adrien, *The Missing Link: Modal Synthesis*, pp. 269–298. Cambridge, MA, USA: MIT Press, 1991.

[2] F. Avanzini, M. Rath, and D. Rocchesso, "Physically-based audio rendering of contact," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME2002)*, vol. 2, (Lausanne, Switzerland), pp. 445–448, IEEE, 2002.

[3] M. Tiraboschi, F. Avanzini, and S. Ntalampiras, "Spectral Analysis for Modal Parameters Linear Estimate," in *Proceedings of the 17th Sound and Music Computing Conference* (S. Spagnol and A. Valle, eds.), SMC, (Torino, Italy), pp. 276–283, Sound and Music Computing Network, Axea sas/SMC Network, 6 2020.

[4] M. Tiraboschi, "SAMPLE – Python package." https://doi.org/10.5281/zenodo.6536419, 2021.

[5] P. Hamill, *Unit test frameworks: tools for high-quality software development*. "O'Reilly Media, Inc.", 2004.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[7] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[8] X. Serra, *A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition*. PhD thesis, CCRMA Department of Music, Stanford University, 1989.

[9] X. Serra and J. Smith, "Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition," *Computer Music Journal*, vol. 14, no. 4, pp. 12–24, 1990.

[10] X. Serra, *Musical sound modeling with sinusoids plus noise*, ch. 3, pp. 91–122. Lisse, the Netherlands: Swets & Zeitlinger Publishers, 1997.

[11] J. Moorer, "Signal processing aspects of computer music: A survey," *Proceedings of the IEEE*, vol. 65, no. 8, pp. 1108–1137, 1977.

[12] C. Voglis and I. Lagaris, "A rectangular trust region dogleg approach for unconstrained and bound constrained nonlinear optimization," in *WSEAS International Conference on Applied Mathematics*, (Boca Raton, FL, USA), p. 4, CRC Press, 2004.

[13] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[14] S. Bock and G. Widmer, "Maximum filter vibrato suppression for onset detection," in *Proceedings of the 16th International Conference on Digital Audio Effects*, (Maynooth, Ireland), 2013.

[15] M. Von Appen *et al.*, "Pygame - a free and open source python programming language library for making multimedia applications." https://pypi.org/project/pygame. Version 2.1.2.

[16] RedFantom *et al.*, "Ttkthemes - a group of themes for the ttk extensions of tcl." https://pypi.org/project/ttkthemes. Version 3.2.2.

[17] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, "Scikit-optimize," Oct. 2021.

[18] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "Ddsp: Differentiable digital signal processing," in *International Conference on Learning Representations*, 2020.

[19] D. Wang and G. J. Brown, *Computational auditory scene analysis: Principles, algorithms, and applications*. Wiley - IEEE press, 2006.

[20] J. O. Smith, "Analytic signals and Hilbert transform filters," in *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications*, W3K Publishing, Second ed., 2007.

[21] R. G. Lyons, *Understanding Digital Signal Processing*, ch. 13.10. Pearson, Third ed., 2011.