The NessStretch: This PaulStretch Algorithm Goes to 9

Alex Ness
Indiana University East
School of Natural Science & Mathematics
alexness@iu.edu

Sam Pluta
The Johns Hopkins University
Peabody Institute
spluta@gmail.com

ABSTRACT

Paul Nasca's PaulStretch is a popular algorithm for extreme sound stretching. We examine some of the details that make the algorithm unique, and then introduce the NessStretch, a new extreme sound stretching algorithm built on the foundation of PaulStretch. The NessStretch refines PaulStretch in three ways: first, it uses a multiresolution FFT to improve low-frequency pitch resolution and high-frequency time resolution; second, it uses correlation calculations between adjacent frames to reduce phase cancellation artifacts and select optimal crossfades; third, it separates transients and/or percussive material from harmonic material, processing each separately to reduce transient smearing across harmonic time frames. In the spirit of PaulStretch, we present open-source implementations of the NessStretch in Python and in SuperCollider.

1. INTRODUCTION

Paul Nasca's PaulStretch is a popular algorithm for stretching audio up to "one quintillion times" its original length. The software was introduced in 2006 and became widely adopted in 2010 [1]. It is designed to do just one thing, and to do it well: produce smoother output for long time-stretches of audio than alternative phase vocoder approaches. The underlying FFT algorithm is simple and effective, and thanks to its licensing as free software, there are several user-friendly implementations available (a VST plug-in [2], Python scripts [3], an Audacity plugin [4], etc.). Due to its constrained functionality, unique sound, and widespread availability, PaulStretch has the unusual distinction among computer-music algorithms of becoming an internet meme [5], sparked by an "800% slower" (i.e. 12.5% playback speed) remix of Justin Bieber's "U Smile."

In this paper, we analyze the PaulStretch algorithm, focusing on the FFT processing decisions that give the algorithm its characteristic sound. We then introduce the NessStretch: a new version of PaulStretch that allows for finer control in different registers, smooths out amplitude distortion, and reduces transient smearing. We have made open-source implementations of the NessStretch available in SuperCollider and Python. Finally, we present some ideas for future experimentation and development.

2. THE PAULSTRETCH EXTREME SOUND PROCESSING ALGORITHM

2.1 Background

Computer time-stretching of audio using granular synthesis and phase vocoder techniques is a relatively recent development, dating back to the 1980s. Although granular synthesis was theorized in the 1940s by Dennis Gabor and implemented in the 1970s by Xenakis, Roads and others [6], the first references to time-shortening and time-stretching using granular synthesis come in Jones and Parks in 1988 [7] and Truax in 1990 [8]. Time-stretching with a phase vocoder was described as early as 1986 by Dolson [9] and 1987 by Gordon et al [10]. These techniques made their way into Tom Erb's popular SoundHack software, written for the Macintosh personal computer starting in 1992. Erb stated that he "wanted to have programs like Mark Dolson's `pvoc` and `convolvesf` on a computer [he] could afford" [11]. By the late 1990s, these techniques had been implemented in widely available computer music programming languages [12][13][14][15], and have since become ubiquitous.

Direct inspiration for the PaulStretch came from Leif Inge's "9 Beet Stretch" [16], with software written in Common Lisp Music by Kjetil S. Matheussen. Upon hearing about this work, Nasca "searched for a program to do it or at least for an algorithm. I couldn't find anything useful, so I decided to think how I could make it by myself" [1].

2.2 Algorithm overview

Nasca maintains two sites [17][18] with information about the PaulStretch software and phenomenon, and one of those sites includes a diagram sketching out the PaulStretch algorithm [19]. We summarize it here:

1. window the current segment of the input audio;
2. take an FFT;
3. randomize the FFT phases;
4. take an IFFT;
5. window the processed segment;
6. overlap-add the processed segment to the current output;
7. advance the input time by O/T and the output time by O, where O is the overlap duration and T is the time-stretch factor;
8. repeat.

This is perhaps the simplest frequency-domain time-stretch algorithm in wide use. It is roughly a phase vocoder [20] that overwrites the FFT's phase information with random

values. Step 7 advances the input frames more slowly than the output frames, resulting in de facto spectral interpolation and time-stretching: for example, with a frame duration of 200 ms, an overlap of 100 ms (i.e. a 2-overlap), and a stretch factor of 4, the input frames advance by 25 ms, while the output frames advance by 100 ms. In this example the output ends up roughly 4 times longer than the input.

2.3 Phase randomization

Phase-vocoder time-stretch algorithms generally suffer from transient smearing [21]. Furthermore, without sophisticated phase processing, the amplitudes of sustained frequencies can become distorted, and harmonic timbres can lose phase coherence.

Nasca realized that although careful phase processing makes sense for subtle time-stretching, it's generally beside the point for time-stretching by a large factor. Of course, by randomizing phases, PaulStretch loses the ability to reconstruct transients and sustained tones without artifacts, but these artifacts may be only a minor distraction, if not a positive side-effect, whenever the musical goal is spacious ambient music. To address transient smearing, Nasca implemented a version of the PaulStretch that analyzes frames for percussive onsets, giving them less prominence in time-stretched output [22].

As we'll see with the NessStretch, multiresolution FFT processing and transient preprocessing can mitigate the smearing, and some simple statistical analysis can mitigate amplitude distortion.

2.4 Windowing

While creators of FFT-based audio effects take various approaches to windowing, we may observe a few conventions. In general, the window shape is fixed (non-variable), and the product of the analysis and synthesis windows satisfies the COLA (constant overlap-add) condition, which is equivalent to an "equal amplitude" crossfade [23]. Quite often, the analysis window is equivalent to the synthesis window, though this is not always the case. [24]. In order to reconstruct the original signal faithfully, Puckette [25] suggests analysis and synthesis with a Hann window and an overlap of four, whereas audio processing software such as SuperCollider [26] and Python's SciPy library [27] use sine analysis and synthesis windows with an overlap of two by default, resulting in a Hann window product.

In many FFT processing contexts (filtering, for example), one can assume that consecutive processed frames are completely correlated. Whenever this is true, a matched analysis/synthesis window pair designed for perfect reconstruction works well. This assumption of perfect correlation, however, is incorrect for any FFT algorithm involving phase randomization, such as PaulStretch. Although the PaulStretch analysis window affects the frequency resolution and dynamic range of the processed signal, the phase randomization nullifies the analysis window's effect on the output envelope. Figure 1 makes this clear by showing the effect of phase randomization on a constant-amplitude sine wave, using sine analysis and synthesis windows.

Normally we would expect a constant output envelope, but that's not the case here. The input envelope has effectively been destroyed. Thus, after phase randomization, the synthesis window is the only window that can have an impact on the output envelope.

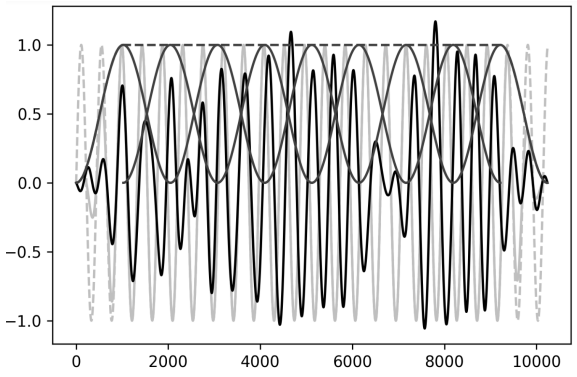

Figure 1. Amplitude distortion of a sine wave input signal due to phase randomization (sine windows superimposed on the output signal)

The current Python version of PaulStretch, `paulstretch_stereo.py`, uses an empirically derived formula [28] for the analysis and synthesis windows:

$$\left(1 - n^2\right)^{5/4} \tag{1}$$

with $-1 \leq n \leq 1$, scaled appropriately over the length of the frame. This unusual window is close to a sine window, providing a cross point of 0.698 (just shy of the sine's $\sqrt{1/2} \approx 0.707$), and giving essentially an equal-power crossfade between adjacent frames:

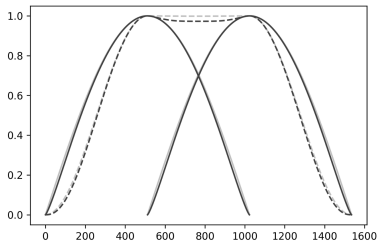

Figure 2. Sine (gray) and PaulStretch (black) windows, with crossfade power

Assuming a decorrelated output akin to noise, there is certainly a rationale for choosing an equal-power synthesis window. The average correlation between phase-randomized frames over time will be 0, so the PaulStretch window roughly preserves consistent output power over the long term, as is desirable. It works particularly well for large frames (8192 samples or more) and noisy input. However, the window fails to address undesirable localized artifacts introduced by the phase randomization, which changes the correlation between adjacent frames, making consistent power output impossible from one frame to the next. Furthermore, half of these correlations are negative, potentially causing extreme dips in both power and amplitude at the crossfade point. As we will discuss below, closely analyzing the correlation between frames and creating custom crossfades per overlap is one way to achieve a clearer sound with minimal artifacts.

3. THE NESSSTRETCH ALGORITHM

The NessStretch algorithm preserves the heart of the PaulStretch algorithm. Like the PaulStretch, the NessStretch performs an FFT analysis of the audio, randomizing the phases of each frame.

3.1 Multiresolution FFT

But whereas PaulStretch uses a single FFT window size throughout the entire frequency range (like a conventional STFT), the NessStretch uses a multiresolution FFT, with shorter windows for higher frequencies.

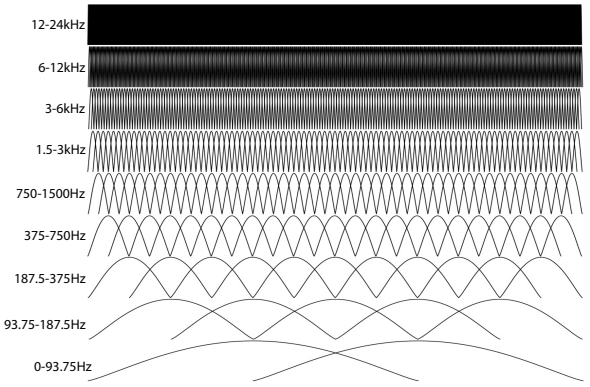

Figure 3. NessStretch windowing illustration

The default settings for a 48 kHz input are shown in the table below:

window size	number of bins	low frequency (Hz)	high frequency (Hz)	window duration (ms)
256	64	12000	24000	5.33
512	64	6000	12000	10.67
1024	64	3000	6000	21.33
2048	64	1500	3000	42.67
4096	64	750	1500	85.33
8192	64	375	750	170.67
16384	64	187.5	375	341.33
32768	64	93.75	187.5	682.67
65536	128	0.00	93.75	1365.33

Figure 4. Window sizes and frequency ranges for 48 kHz input

For each frequency band, the new algorithm calculates an FFT frame, applies a brick-wall bandpass filter [1] (or, optionally, a steep Linkwitz-Riley bandpass filter [30]) to (effectively) zero the magnitudes of the bins outside the desired frequency range, and randomize the phases of the desired bins. The crossfades between frames are then calculated based on correlation (see below) and the output frames are lined up so that their center points in the input buffer align across all frequency bands in the output buffer.

[1] A brick-wall bandpass filter introduces a ringing artifact in the time domain (the "Gibbs phenomenon"). In many contexts this artifact is undesirable, and methods have been proposed to remove it with additional frequency-domain processing [29]. However, we have found that phase randomization renders any brick-wall filter artifacts effectively inaudible.

The NessStretch's multiresolution windowing is a better match for frequency perception than PaulStretch's uniform windowing. Bin spacing varies between about 0.268 semitones at the bottom of each octave band and 0.136 semitones at the top of each octave band, and the bins are spaced approximately logarithmically through the audible frequency range. The windowing and phase randomization diffuses the pitch somewhat, but the bin spacing ensures that this diffusion will never be too extreme. This layered approach does a better job of resolving shorter, noisier high-frequency sounds (sibilance, snares, etc.), and improves clarity for the midrange as well.

(If the audio being stretched is mostly tonal—choral music, for example—it may help to double the frequency resolution, whereas for noisy, active textures—ambient crowd noise, for example—it may help to double or quadruple the time resolution instead.)

3.2 Correlation-based phase correction and crossfading

As noted by Fink et al. [31], a crossfade between two sounds should result in an output signal with consistent loudness and power. The correct crossfade is determined by the two sounds' correlation coefficient r_{xy}:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}, \qquad (2)$$

where n is the length of the crossfade (in samples), x_i and y_i are the amplitudes of the two sounds at sample number i, and $\bar{x}$ and $\bar{y}$ are the sounds' average amplitudes. Two noisy sounds, phase-randomized or not, are approximately uncorrelated: $r_{xy} \approx 0$. For two phase-randomized pure tones with the same frequency, the situation is more complicated: their correlation is the cosine of the difference of their phases [32], which will vary frame by frame between $r_{xy} = -1$ (completely inverted phase) and $r_{xy} = 1$ (completely synchronized phase), with a long-term average of $r_{xy} = 0$.

This figure shows overlapping signals crossfaded with a Hann window:

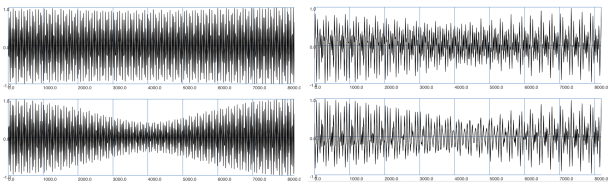

Figure 5. Sine waves and sine wave triads: positive correlation on top, negative correlation below

The plots on the left show the crossfade of a sine tone. The crossfade on top has a correlation of $r_{xy} = +0.67$, and the one on the bottom has $r_{xy} = -0.67$. The plots on the right show a triad of three sine waves with a correlation of $r_{xy} = +0.275$ on top and $r_{xy} = -0.275$ on the bottom. For both signals, the dip in amplitude, especially between the negatively correlated windows, is an audible artifact we would like to discard.

The NessStretch addresses these extreme phase-cancellation artifacts by inverting the phase of any frame correlated negatively with its predecessor. Inverting the

phase inverts the correlation coefficient, guaranteeing a non-negative value: $0 \leq r_{xy} \leq 1$. While this doesn't fix amplitude variation for each sinusoid within each overlap, it significantly reduces amplitude distortion for 50% of the crossfades created in the process.

To smooth out the remaining amplitude distortion, we look again to Fink et al. [31], who have derived correlation-variable crossfade functions that vary between equal-power (when $r_{xy} = 0$) and COLA (when $r_{xy} = 1$). Let N be the length of the crossfade in samples and $n \in [0, \ldots, N-1]$ be a crossfade sample index. Then let $\alpha = \frac{n}{N-1}$ be the normalized sample time, so that $0 \leq \alpha \leq 1$. For the NessStretch default settings, we use the following fade-out envelope:

$$w_{\text{out}}(\alpha, r_{xy}) = \frac{1}{\sqrt{1 + 2\tan^2(\frac{\pi\alpha}{2})r_{xy} + \tan^4(\frac{\pi\alpha}{2})}}. \quad (3)$$

This results in a "Hann" crossfade

$$w_{\text{out}}(\alpha, 1) = \cos^2(\pi\alpha/2) \quad (4)$$

whenever $r_{xy} = 1$, and the equal-power crossfade

$$w_{\text{out}}(\alpha, 0) = \frac{1}{\sqrt{\tan^4\left(\frac{\pi\alpha}{2}\right) + 1}} \quad (5)$$

whenever $r_{xy} = 0$. Applying this window and its inverse fade-in curve reduces much of the remaining amplitude distortion, resulting in smoother overlaps between frames:

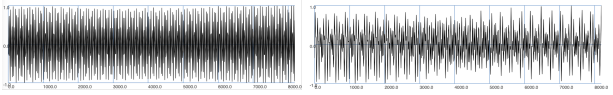

Figure 6. Positively correlated overlaps with corrective envelopes

3.3 Transients

Transients provide a challenge for any time-stretching algorithm. The NessStretch algorithm, as described above, smears out transients' low frequencies while keeping high frequencies relatively concentrated. For a sound file stretched to 100 times its original length, a transient's low frequencies fade in almost 75 seconds before its high frequencies, and they fade out almost 75 seconds after its high frequencies fade out, creating a characteristic "filter-sweep" effect that may or may not be desirable. Solutions to this problem include Nasca's modified algorithm that identifies frames with percussive onsets and uses those frames fewer times in the time-stretched output [22]. Alternatively, Nagel and Walther [33] and Grofit and Lavner [34] have each proposed separating the transient and reinserting it at its original speed after time-stretching the remaining part of the signal. Finally, Driedger and Müller have proposed using Harmonic-Percussive Source Separation (HPSS), then time-stretching using the phase-locked method on the harmonic component and an overlap-add method on the percussive component [35].

Our own two solutions incorporate some of these ideas to make the NessStretch's transient spread less extreme. Prior to time-stretching, we extract sharp attacks from the source audio file using a transient separation algorithm [36] or HPSS [37]. The transients or percussive elements are then stretched with the NessStretch algorithm, keeping the default frame sizes for the top four octaves, but using a maximum window size of 2048 samples for everything below. This reduces the duration of transients' low frequencies by a factor of 32 (in the example above, down from nearly 150 seconds to just under 5 seconds). While still maintaining the algorithm's signature sound around transients, these attacks are sharper and do not muddy otherwise harmonic regions.

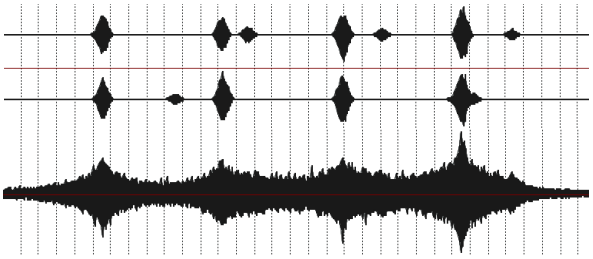

Figure 7. Comparing NessStretch transient smearing: adjusted settings on the top (max window size of 2048, default settings on bottom (max window size of 65536)

4. LISTENING COMPARISONS

The difference between the PaulStretch and the NessStretch is apparent for most input files. Compare, for example, "U Smile" PaulStretched and NessStretched to 1% of the original speed (100 times duration), available at `http://www.sampluta.com/NessStretch.html`.

The PaulStretched version is a beautiful wash, blending together the voice, instruments, and drums into an ambient haze. The vocal consonants are completely lost, and the snare drums fade in and out in waves.

In the NessStretch version, on the other hand, not only do the voice and instruments all maintain their identity and presence, but it's possible to tune into details that are too subtle or short to hear in the original recording: vowel formants; pitch bends between sung notes; piano and bass harmonics; early reflections from the snare hits. Even at a fraction of the original speed, the texture is varied and active.

We have also provided a comparison between stretches with and without transient separation and HPSS preprocessing. The sonic difference here is likely more of matter of taste. Without separation, noise and pure tones blend together in the texture, and percussive transients gradually emerge and disappear. With separation, transients are more distinct, adding a more "aggressive" layer to a harmonic background.

5. PYTHON AND SUPERCOLLIDER IMPLEMENTATIONS

We have made the NessStretch code available on GitHub at:

`https://github.com/spluta/TimeStretch/`

It is released under a GPLv3 license.

6. FUTURE WORK

We have implemented the NessStretch in two scripting languages, Python and SuperCollider sclang. The current versions run at approximately real-time: with either version, a six-hour time-stretch of a stereo signal split into resonance and transients (thus four channels of audio) takes around six hours total on a modern eight-core laptop. A server-side version is implemented in SuperCollider scsynth, but it is not able to achieve the sample-accurate sonic results of the scripting versions. We would like to improve the efficiency and usefulness of the algorithm by creating a C++ version (following PaulStretch), for easy integration into audio editors like Audacity.

Acknowledgments

Special thanks to Paul Nasca for his algorithm, J.-P. Drécourt for his implementation of the PaulStretch in SuperCollider, Jem Altieri for helping with Python refactoring and command line documentation, Caroline Mallonée for listening to output samples, and Miller Puckette and Alex Harker for invaluable feedback on the project.

7. REFERENCES

[1] "PaulStretch: An Interview with Paul Nasca Its Creator — Microscopics," Available at http://www.microscopics.co.uk/blog/2010/paulstretch-an-interview-with-paul-nasca/ (2021-01-22).

[2] Xenakios, "PaulXStretch plugin — Xenakios's Blog," Available at https://xenakios.wordpress.com/paulxstretch-plugin/ (2021-01-13).

[3] N. Paul, "Paulstretch Python version," Available at https://github.com/paulnasca/paulstretch_python (2021-01-13).

[4] "Paulstretch - Audacity manual," Available at https://manual.audacityteam.org/man/paulstretch.html (2021-01-13).

[5] "Time Stretch / 800% Slower — Know Your Meme," Available at https://knowyourmeme.com/memes/time-stretch-800-slower (2021-01-13).

[6] C. Roads, "Introduction to Granular Synthesis," *Computer Music Journal*, vol. 12, no. 2, pp. 11–13, 1988. [Online]. Available: http://www.jstor.org/stable/3679937

[7] D. L. Jones and T. W. Parks, "Generation and Combination of Grains for Music Synthesis," *Computer Music Journal*, vol. 12, no. 2, pp. 27–34, 1988. [Online]. Available: http://www.jstor.org/stable/3679939

[8] B. Truax, "Time shifting of sampled sound with a real-time granulation technique," *in Proceedings of the 1990 International Computer Music Conference*, 1990.

[9] M. Dolson, "The Phase Vocoder: A Tutorial," *Computer Music Journal*, vol. 10, no. 4, pp. 14–27, 1986. [Online]. Available: http://www.jstor.org/stable/3680093

[10] J. W. Gordon and J. Strawn, *An introduction to the Phase Vocoder*. Los Altos, CA: William Kaufmann, 02/1987 1987. [Online]. Available: https://ccrma.stanford.edu/files/papers/stanm55.pdf

[11] T. Erbe, "SoundHack: A Brief Overview," *Computer Music Journal*, vol. 21, no. 1, pp. 35–38, 1997. [Online]. Available: http://www.jstor.org/stable/3681214

[12] M. Puckette, "Phase-locked vocoder," *in Proceedings of the 1990 International Computer Music Conference*, pp. 222 – 225, 1995.

[13] R. Karpen, "sndwarp," Available at http://www.csounds.com/manual/html/sndwarp.html (2021-01-27), 1997.

[14] J.-F. Charles, "A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter," *Computer Music Journal*, vol. 32, no. 3, pp. 87–102, 2008. [Online]. Available: http://www.jstor.org/stable/40072649

[15] R. Karpen, "Csound's phase vocoder and extensions," in *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*, R. Boulanger, Ed. Cambridge, MA, USA: MIT Press, 2000, p. 541–560.

[16] L. Inge, "9 Beet Stretch," Available at http://www.xn--lyf-yla.com/about9.htm (2021-01-24).

[17] P. Nasca, "Paul's Extreme Sound Stretch," Available at http://hypermammut.sourceforge.net/paulstretch/ (2021-01-16).

[18] "Open Source projects - Paul Nasca," Available at http://www.paulnasca.com/open-source-projects#TOC-Paul-s-Extreme-Sound-Stretch (2021-01-16).

[19] P. Nasca, "Algorithms created by me — Paul Nasca," Available at http://www.paulnasca.com/algorithms-created-by-me#TOC-PaulStretch-extreme-sound-stretching-algorithm (2021-01-16).

[20] J. O. Smith, "Phase Vocoder," Available at https://ccrma.stanford.edu/~jos/sasp/Phase_Vocoder.html (2020-07-26).

[21] A. Altoe, "A transient-preserving audio time-stretching algorithm and a real-time realization for a commercial music product," Ph.D. dissertation, Faculty of Engineering, University of Padova, Padua, Italy, 2012.

[22] P. Nasca, "Paulstretch newmethod," Available at https://github.com/paulnasca/paulstretch_python/blob/master/paulstretch_newmethod.py (2021-01-16).

[23] J. O. Smith, "COLA Examples," Available at https://ccrma.stanford.edu/~jos/sasp/COLA_Examples.html (2021-01-16).

[24] M. Puckette, "On timbre stamps and other frequency-domain filters," *in Proceedings of the International Computer Music Conference*, pp. 287–290, 2007.

[25] ——, *The Theory And Techniques Of Electronic Music*. World Scientific Publishing Company, 2007, pp. 275–276.

[26] "FFT — SuperCollider 3.11.1 Help," Available at https://doc.sccode.org/Classes/FFT.html (2021-01-21).

[27] Fourier Transforms (scipy.fft) — SciPy v1.6.0 Reference Guide. Available at https://docs.scipy.org/doc/scipy/reference/tutorial/fft.html (2021-01-21).

[28] J.-P. Drécourt, Personal communication, 2021.

[29] C. Pan, "Gibbs phenomenon removal and digital filtering directly through the fast Fourier transform," *Signal Processing, IEEE Transactions on*, vol. 49, pp. 444 – 448, 03 2001.

[30] "Linkwitz-Riley Crossovers: A Primer," Available at https://www.ranecommercial.com/legacy/note160.html (2021-01-22).

[31] M. Fink, M. Holters, and U. Zölzer, "Signal-matched power-complementary cross-fading and dry-wet mixing," *in DAFx-2016 - Brno*, 09 2016.

[32] J. D. Cook, "Correlation of two out-of-phase sine waves," Available at https://www.johndcook.com/blog/2016/03/06/correlating-two-sine-waves/(2021-01-16).

[33] F. Nagel and A. Walther, "A novel transient handling scheme for time stretching algorithms," *Journal of the Audio Engineering Society*, October 2009.

[34] S. Grofit and Y. Lavner, "Time-Scale Modification of Audio Signals Using Enhanced WSOLA With Management of Transients," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 16, pp. 106 – 115, 02 2008.

[35] J. Driedger and M. Müller, "TSM Toolbox: MATLAB Implementations of Time-Scale Modification Algorithms," 01 2014.

[36] P. A. Tremblay, O. Green, G. Roma, and A. Harker, "From Collections to Corpora: Exploring Sounds through Fluid Decomposition," *in Proceedings of the International Computer Music Conference*, 2019.

[37] B. McFee, C. Raffel, D. Liang, D. Ellis, M. Mcvicar, E. Battenberg, and O. Nieto, "librosa: Audio and Music Signal Analysis in Python," *Python in Science Conference*, pp. 18–24, 01 2015.