Time Stretching & Pitch Shifting with the Web Audio API: Where are we at?

ABSTRACT

Audio time stretching and pitch shifting are operations that all major commercial and/or open source Digital Audio Workstations, DJ Mixing Software and Live Coding Suites offer. These operations allow users to change the duration of audio files while maintaining the pitch and vice-versa. Such operations allow, for example, a DJ to speed up, or slow down, a song that is being mixed into another song, in order to align the tempo, and beats, of both songs. Unfortunately, there are few (and experimental) client-side JavaScript implementations of these two operations.

In this paper, we review the current state of art for client-side implementations of time stretching and pitch shifting, their limitations, and describe two new implementations for two well-known algorithms: (1) Phase Vocoder with Identity Phase Lock and (2) a modified version of Overlap & Add.

Additionally, we discuss some issues related to the Web Audio API (WAA) and frequency-based audio processing regarding latency and audio quality in pitch shifting and time stretching towards raising awareness about possible changes in the WAA.

1. INTRODUCTION

Time stretching and pitch shifting are two operations widely available in commercial and open source musical applications like Ableton Live,[1] Traktor Pro[2] and Ardour.[3] Currently, creative frameworks implemented in JavaScript and the Web Audio API (WAA) such as Flocking [2] can only change the duration of a signal through re-sampling of audio buffers, thus changing both the duration (i.e.: tempo) and pitch at the same time. In digital audio workstations (DAW) like EarSketch [11], time stretching and pitch shifting is performed server-side, with SoX,[4] making real-time interaction impossible. Our aim in this work is towards enabling real-time pitch shifting and time-stretching in browsers, which is currently an under-explored aspect of the WAA.

This paper presents an overview of the current implementations for both algorithms using JavaScript in the browser environment and two new implementations. Additionally, we discuss the impact of certain decisions in the WAA design and philosophy regarding the absence of frequency-based operators like the Fast Fourier Transform (FFT), and their impact on performance.

For both implementations, we had the following non-functional requirements:

- **Constant memory usage**, with fixed size circular arrays, in order to minimize the impact of the garbage collector, by maintaining a constant memory profile.
- **Intuitive API and documentation**, stating the trade-offs between audio quality and performance.
- **Minimize assumptions** regarding sample rate, number of channels, the type of audio content (percussive or harmonic) being processed, third party software components and execution environments.

The outline of the paper is the following. First, we review the basic theory for time domain and frequency domain time stretch and pitch shifting algorithms. Then, we describe the existing Javascript implementations of such algorithms. After that, we present our two new implementations for two well know algorithms: (1) Phase Vocoder with Identity Phase Locking [8] and (2) a modified Overlap & Add (OLA) algorithm. Finally, we discuss the trade-offs between existing implementations and problems with the WAA regarding these two operations.

2. THEORETICAL BACKGROUND

For both tasks, there are algorithms that work in time domain, like Overlap and Add (OLA), Waveform Similarity based OLA (WSOLA) [14] and delay line modulation [4], and others that work in the frequency domain, like the Phase Vocoder [5] and Spectral Modelling [20]. In this section, we give an overview of three popular methods: OLA, WSOLA and the Phase Vocoder.

Algorithms in both time and frequency domains, for time stretching and pitch shifting, are more sophisticated versions of OLA. For that reason, we start with an overview of the basic OLA algorithm.

[1] https://www.ableton.com/
[2] http://bit.ly/1izsxHy
[3] http://ardour.org/
[4] http://sox.sourceforge.net/

Web Audio Conference WAC-2016, April 4–6, 2016, Atlanta, USA

2.1 OLA

Let $x \in \mathbb{R}^M$ be the input signal of size M, $\alpha \in \mathbb{R}$ be the stretching factor and $y \in \mathbb{R}^L$ be the output signal of size $L = \alpha \cdot M$. There are three main steps for OLA:

1. Partition the input signal x into a set of analysis frames of size N, each overlapping with the previous one in $H_a \in \mathbb{N}^+$ (i.e.: the analysis hop size) samples.

$$x_i(n) = x(n + i \cdot H_a),\ x_i \in \mathbb{R}^N \tag{1}$$

2. Apply a window w to each analysis frame.

$$x_{w_i}(n) = w(n) \cdot x_i(n),\ w \in \mathbb{R}^N \tag{2}$$

3. Add the (synthesis) frame to the output, by overlapping it with $H_s \in \mathbb{N}^+$, the synthesis hop size, samples of the "tail" of the output signal y

$$y(n + i \cdot H_s) = \begin{cases} y_i(n), & \text{if} i = 0 \\ y(n + i \cdot H_s) + \frac{y_i(n)}{w(n)}, \text{ if } i > 0 \end{cases} \tag{3}$$

with $y_i = x_{w_i}$ for the basic OLA, $n \in \mathbb{I} \wedge 0 \leq n \leq N$ and $i \in \mathbb{N} \wedge i \cdot H_a \leq M \wedge i \cdot H_s \leq L$. The relation between both overlap/hop sizes and the the stretching factor is defined as:

$$\alpha = \frac{H_s}{H_a}. \tag{4}$$

Usually, one of the hop sizes is fixed while the other is a function of α. We should note that α can change at each new frame, which is particularly relevant in for real-time audio processing. OLA does not preserve phase relations between consecutive frames and, as such, there are noticeable artefacts in the output signal: modulation of harmonic structures (e.g.: human voice) and reverberation. The temporal complexity of OLA is $O(N)$. In figure 1(a), we can see what happens to both H_a and H_s when changing α.

To perform pitch shifting with OLA, we could couple a re-sampler to the OLA time stretcher. In order to allow simultaneous time stretching and pitch shifting, let t and $t+1$ be the current and next moments, β_t be the new desired pitch and R_b the sampling rate of the input signal. Then, the new stretching factor α_{t+1} and the new sample rate R_{t+1} are defined as

$$R_{t+1} = R_b * \beta_{t+1} \tag{5}$$

$$\alpha_{t+1} = \frac{\alpha_t}{R_t}. \tag{6}$$

2.2 WSOLA

First introduced in [14], WSOLA applies a delay $\delta \in [-\delta_{max} : \delta_{max}]$ to each analysis frame, such that the waveforms of two overlapping synthesis frames are as similar as possible in the overlapping regions.[5] The delay can be obtained by calculating the cross-correlation between the overlapping regions of each synthesis frame. Therefore, the analysis step for x_i is redefined as

$$x_i(n) = x(n + H_{a_i}) \tag{7}$$

where H_{a_i} is the analysis hop size for frame i and is defined as

$$H_{a_i} = i \cdot H_a + \delta_i \tag{8}$$

[5]WSOLA is equivalent to OLA when $\delta_{max} = 0$.

and

$$\delta_i = x_i \star x_{i-1}[\delta_{max}] \tag{9}$$

where $\delta_{max} < H_a$. This algorithm removes the reverberation and modulation but, for transient rich sounds (percussion, guitar riffs), there might be some missing transients and stuttering (i.e.: repeated transients). Regarding the temporal complexity of WSOLA, if the cross-correlation is implemented with FFT Convolution [16], we get $O(Nlog_2N)$. Otherwise, if it is a "naive" implementation of the convolution, we get $O(N^2)$. To perform pitch shifting, we can use the same method described for OLA.

2.3 Phase Vocoder

The Phase Vocoder is a well documented [3, 5, 6, 7, 8, 10, 20] algorithm used for time stretching [5, 6], pitch shifting [10] and other audio effects like robotization [20]. In general, it has a higher computational cost than time domain methods like WSOLA but offers higher quality audio, without missing transients. Each iteration of the Phase Vocoder has eight steps which occur in-between steps 2 and 3 of OLA:

1. Calculate the forward Fourier transform of x_{w_i}

$$X_i(n, \Omega_k) = \sum_{n=0}^{N} x_{w_i}(n) \cdot e^{-j\Omega_k n}, X_i \in \mathbb{C}^N \tag{10}$$

where $\Omega_k = \frac{2\pi k}{N}$ is the frequency center for frequency bin k and $e^{-j\Omega_k n}$ is a complex sinusoid of frequency Ω_k.

2. Calculate the magnitude $|X_i| \in \mathbb{R}^N$ and phase $\angle X_i \in \mathbb{R}^N$ spectra for X_i by converting X_i to polar coordinates.

3. Calculate the difference between current and previous phase spectras and, then, the sample-wise difference with the frequency centres Ω_k

$$\Delta_{\angle X_i} = \angle X_i - \angle X_{i-1} - H_a \cdot \Omega_k \tag{11}$$

4. Due to the fact that the phase values are given in modulo 2π and, as such, phase 'jumps' will occur, we need to unwrap the phase in order to obtain a continuous phase function

$$\Delta^p_{\angle X_i} = \Delta_{\angle X_i} - 2\pi \cdot \lfloor \frac{\Delta_{\angle X_i}}{2} \rceil \tag{12}$$

5. Compute the instantaneous frequency ω for each freq. bin k

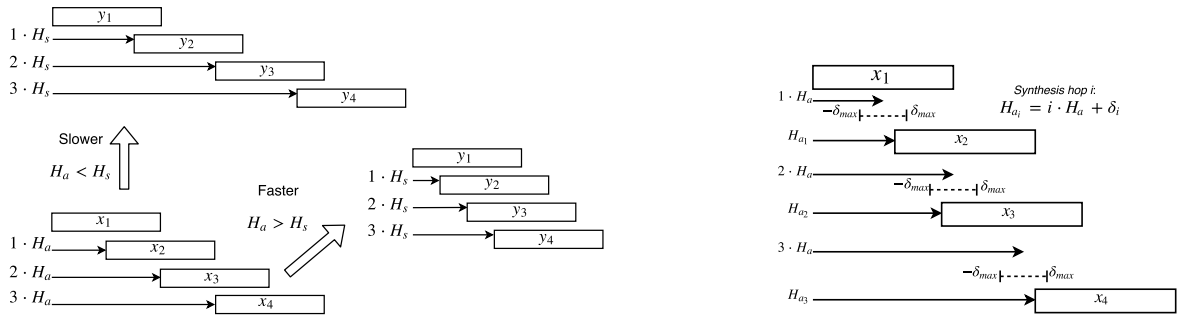
$$\omega_k = \Omega_k + \frac{\Delta^p_{\angle X_i}}{H_a} \tag{13}$$

6. Now, we can use ω_k to compute the output phase spectra $\angle Y_i$ by advancing the previous output $\angle Y_{i-1}$ according to the synthesis hop size H_s

$$\angle Y_i = \angle Y_{i-1} + H_s \cdot \omega_k \tag{14}$$

7. Compute the synthesis frame $Y_i \in \mathbb{C}^N$ by reusing the input magnitude spectra and the new phase spectra

$$Y_i = |X_i| \cdot e^{j \cdot \angle Y_i} \tag{15}$$



(a) (b)

Figure 1: Illustrations for (a) the basic OLA algorithm, with both analysis and synthesis steps and (b) the WSOLA analysis step. For time compression, we have $H_a < H_s$ and for time expansion, $H_a > H_s$. In WSOLA, the main difference is within the "reading head" position. For each analysis frame i, the analysis hop size H_{a_i} is calculated according to the optimal displacement δ_i, obtained through cross-correlation between analysis frame i and $i-1$, for a maximum delay of δ_{max}: $H_{a_i} = i \cdot H_a + \delta_i$

8. Finally, we calculate the inverse Fourier transform y_i of the frequency Y_i

$$y_i(n) = w(n) \cdot \sum_{k=0}^{N} Y_i(n, \Omega_k) \cdot e^{j\Omega_k n} \qquad (16)$$

This algorithm ensures *horizontal phase coherence* (i.e.: phase continuity for the same frequency bin, in consecutive frequency frames) is guaranteed. However, *vertical phase coherence* (i.e.: phase relations between different frequency bins, in the same frame) is usually destroyed in the phase correction process. The loss of *vertical phase coherence* results in a distinct artefact: *phasiness* (i.e.: a metallic tunnel sound). The temporal complexity of the Phase Vocoder is $O(N\,log_2 N)$.

To maintain both vertical and horizontal coherence, we can apply a technique known as *Identity Phase-Locking* [9]. The main idea is that frequency bins that are not spectral peaks contribute to the partials of the nearest spectral peak. A spectral peak is a local maximum in the magnitude spectra. In order to find the spectral peaks, we can use a simple heuristic: if a frequency bin has the maximum magnitude when compared with four neighbour bins, then it is a spectral peak. After identifying all spectral peaks, we need to infer the "region of influence" of each peak (i.e.: for a given peak, which are the non-peak bins that contribute to the peak partial) (see section IIIC of [9]). After identifying both the spectral peaks and the "regions of influence", the usual phase correction method is applied to the peak bins. The phase of a non-peak bin will be equal to the phase of its corresponding peak bin.

To pitch shift with a Phase Vocoder, there are two methods available. The first one is re-sampling, in the same manner as OLA and WSOLA. The second one is adding an additional step to the phase correction, as described in [10], which proceeds with the following steps. After identifying the spectral peaks, for a pitch shift factor β, each peak will be shifted to a new (angular) frequency $\beta\omega$, corresponding to a freq. shift $\Delta\omega = \omega(\beta - 1)$. When $\Delta\omega$ is an integer, we just need to copy the Fourier transform values from the original "region of influence" to the new one (around the new peak bin). If $\Delta\omega$ is a fractional number, a naive solution is to round $\Delta\omega$ to the nearest integer. This solution presents acceptable results for low sample rates and large FFT frame sizes.

3. EXISTING IMPLEMENTATIONS

In this section we give an overview regarding time stretching and pitch shifting implementations with web technologies like JavaScript and the WAA, as well as native web browser implementations, available through the Audio Element [13], via its *playbackRate* attribute [18].

For pitch shift only implementations in JavaScript, there is pitchshift.js [1] and jungle.js.[6] The first is a port of a C++ implementation[7] of the Phase Vocoder for pitch shifting [10] while the second is an implementation of the algorithm presented in [4].

In Vexwarp,[8] time stretching is performed with the basic phase vocoder algorithm [5, 6]. With this application, it is not possible to perform real-time processing of the input signal.

Soundtouch.js[9] is a port of the C++ library SoundTouch,[10] a WSOLA implementation. This library performs both time stretching and pitch shifting (through resampling) with some additional features:

- Cross-correlation is computed with an interleaved array with all audio channels.
- Instead of implementing the "naive" approach to cross-correlation, the developers used a hierarchical algorithm.

This port is tightly coupled regarding buffers, buffer management, stretcher, re-sampler and parameter adjustments, as well as some hard coded parameters, making integration into new applications a difficult task.

There is another WSOLA implementation: tempo.js [1], the result of the compilation of a port of the SoX tempo

[6]https://github.com/cwilso/Audio-Input-Effects
[7]http://downloads.dspdimension.com/smbPitchShift.cpp
[8]http://www.vexflow.com/vexwarp/
[9]https://github.com/also/soundtouch-js
[10]http://www.surina.net/soundtouch/

effect, using Emscripten [19]. Currently, it has no documentation and the only demo available uses deprecated APIs that are no longer available.

In the WAVES project,[11] the Audio library [15] supports time stretching and pitch shifting through granular synthesis and re-sampling, offering two classes to perform both tasks: GranularEngine and SegmentEngine. The first one causes significant transient smearing. The second class requires the developer to pass a JSON object detailing the segmentation of the input audio buffer.

Regarding the native implementations in web browsers, all major browsers, like Opera, Safari, Chrome and Firefox, implement time stretching to be used with the Audio tag, controlling the stretching factor with the *playbackRate* attribute of the Audio tag/object. The stretching factor in the current implementations seems to be limited to the range $\alpha \in [0.5 : 4]$, where 0.5 is the slowest speed and 4 is the fastest one, meaning that web browsers perform an additional step in order to make the provided α comply with equation (4). Currently, we can only detail the implementation of Firefox and Chromium due to the closed-source nature of Opera and Safari. In Firefox, time stretching is performed by the SoundTouch library. Chromium uses a custom WSOLA implementation. For both browsers, when slowing down, there is some stuttering (more noticeable in Firefox). When speeding up, namely for *playbackRate* values greater than 1.2, there are some missing transients for percussive sounds.

4. NEW IMPLEMENTATIONS

In this section, we detail our two implementations for time stretching, OLA-TS.js (modified OLA) and PhaseVocoder.js (Phase Vocoder with Identity Phase-Locking), as well as some helper classes we created to ease the integration of the time stretchers.

Both implementations operate on a single audio channel and can be included in a *ScriptProcessor* or *AudioWorker* for real-time interaction/processing, or they can be used in batch processsing, in a similar way to Vexwarp, in order to integrate in frameworks and applications like Flocking and EarSketch. They do not include pitch shifting capabilities but that be easily changed by coupling a re-sampler to the helper classes. In order to maintain a static memory footprint, we used an existing circular buffer implementation, CBuffer.[12] Even though our time stretchers are implementations of (totally) different algorithms, there is a common API to both stretchers:

- *process(Array inputFrame, CBuffer outputFrame)*: given a (mono) frame, performs a time stretching iteration and pushes H_s samples in the output CBuffer.
- *get_ha*: returns the current analysis hop size. This function calculates the increment to the "read head" of the input signal, when playing an audio file.
- *get_hs*: returns the current synthesis hop size. This function calculates the increment to the output signal position which an be used to guide the cursor in the UI of an audio player using OLA-TS.js or PhaseVocoder.js as time stretchers.
- *clear_buffers*: clears all internal buffers, like the overlapping buffer. This can be useful for audio players that need to create a noticeable stop in the transition to the next file in a playlist, in order to avoid using the phase of the previous song to adjust the phase of the next song.
- *set_alpha(Number newAlpha)*: given the new stretching factor, it computes the new values for H_s, H_a (both integers) and invokes the function pointed by *overlap_fn*.
- *get_alpha*: returns the last specified stretching factor.
- *overlap_fn*: a public field pointing to a function that, given a stretching factor α, will return a new overlapping factor.
- *get_real_alpha*: there are stretching factors that do not allow H_s and H_a to be integers and this might present a problem because the input signal "read head" is an integer number (H_a is used to increment the "read head"). When the developer uses *set_alpha* to specify a new stretching factor, both H_s and H_a are rounded to the closest integer. As a result, there will be a difference between the specified α and the real α. This difference can cause problems in use cases like a DJ application that automatically synchronizes two songs to a master tempo. If there is a divergence between the specified α and the real α, after a certain amount of time, this divergence can cause the beats of both songs to drift out of sync. Therefore, we created this function in order to allow the developer to create adequate controllers to adjust the speed of the audio players to circumvent this issue.

4.1 OLA-TS.js

OLA-TS.js diverges from the basic OLA algorithm as follows: the window has an exponent that is a function of the stretching factor, $W'(n) = W(n)^{\beta(\alpha)}$. The overlapping factor is also a function of α, $Ovl(\alpha)$. We include two default functions for both the overlapping factor and the exponent. The exponent function can be redefined by changing the public field *beta_fn*. The default functions for the exponent and overlapping factor were designed through experimentation. Both of them are a series of step functions design to minimize the modulation described in section 2.1.

Both OLA-TS.js and PhaseVocoder.js use the following formulas to define the analysis and synthesis hop sizes:

$$H_a = \frac{N}{Ovl(\alpha)} \tag{17}$$

$$H_s = \alpha * H_a \tag{18}$$

where $Ovl(\alpha)$ is the function defined in *overlap_fn*. In order to properly stretch a input signal, the developer should use a predetermined sequence of instructions. To make integration in other applications easier, we implemented helper classes to manage the buffering and the "read heads" for the input buffers. Both implementations and their helper classes are public and open source, each one with it's own Git repository.[13][14] Additionally, we implemented a small

[11] http://wavesjs.github.io/
[12] https://github.com/trevnorris/cbuffer
[13] https://github.com/echo66/OLA-TS.js
[14] https://github.com/echo66/PhaseVocoderJS

demo page where the user can drag & drop several songs and play them simultaneously, controlling the volume and the stretching factor.

4.2 PhaseVocoder.js

PhaseVocoder.js uses, by default, fourier.js, an FFT library offering methods implemented in both asm.js[15] and "raw" JavaScript. In order to use a different FFT library, the developer can use the following methods: *set_stft_fn(Function stftCallback)*, *set_istft_fn(Function istftCallback)* and *init_fft_fn(Function initCallback)*:

- *stftCallback(Array inputFrame, Array windowFrame, Number wantedSize, Object out)*: *inputFrame* is a sequence of samples; *windowFrame* is the discretization of the window function; *wantedSize* is the desired size for the real and imaginary arrays of the output[16]; *out* is a JSON object with four arrays: real, imaginary, magnitude and phase, all of them describing the result of the forward Short Time Fourier Transform (STFT). This function will be invoked for each time frame being processed by PhaseVocoder.js.

- *istftCallback(Array real, Array imag, Array windowFrame, Array timeFrame)*: *real* and *imag* are the real and imaginary arrays describing a frequency frame; *timeFrame* is the result of the inverse STFT. This function will be invoked when synthesizing a frequency frame, after the phase adaptation.

- *initCallback(Number frameSize)*: this function is invoked after being added to a PhaseVocoder.js instance.

4.3 Helper Classes

For both implementations, we created a set of helpers with a common API: *BufferedTS* and *WAAPlayer*.

In BufferedTS, we manage the output buffering of two time stretchers (i.e.: 2 channels), as well as the the "read head" position. For this class, we provide two (public) methods, (1) *process(AudioBuffer oBuf)* which writes the next output frame into *oBuf* and (2) *set_audio_buffer(AudioBuffer iBuf)* which defines the input audio buffer to be processed with the time stretchers, and two (public) read/write fields, *position* and *alpha*, allowing access to both the "read head" and the stretch factor α.

To ease integration with the WAA, we implemented *WAAPlayer*, which integrates a *BufferedTS* instance into a *ScriptProcessor*.

5. EVALUATION AND DISCUSSION

In this section, we compare our implementations with the existing ones, reviewed in Section 3.

Unlike Vexwarp, PhaseVocoder.js allows real-time and block-wise processing of the input signal, with a minimized impact in transient smearing and no metallic sound, due to the inclusion of *vertical phase coherence*. Unfortunately, due to the number of FFTs used (for $\alpha > 1$, there are 4 forward and 4 inverse FFTs per output buffer), the computational costs are greater than soundtouch.js. But, in order to avoid missing transients in soundtouch.js, we must decrease the frame size and/or increase the cross-correlation range, which will increase the computational costs of soundtouch.js and, as a consequence, can make PhaseVocoder.js competitive.

OLA-TS.js allows $O(N)$ time stretching, unlike PhaseVocoder.js, $O(N\,log_2 N)$, and soundtouch.js, $O(N^2)$, but can introduce noticeable modulation in harmonic structures like the voice of a human singer, for frame sizes smaller than 4096 samples, with a sample rate of 44100 Hz. Additionally, it may require some manual experimentation with the overlap and beta parameters to reduce the modulation.[17] Due to the recommended frame size (4096 samples), OLA-TS.js is adequate for applications where there are smooth adjustments in the stretching factor.

To make it simple for the reader to understand some of the main features and problems with each reviewed and/or discussed implementation, Table 1 includes, for each implementation, its algorithm, audio artefacts and the effect(s) implemented.

6. TIME STRETCHING AND PITCH SHIFTING ISSUES WITH THE WAA

Currently, the WAA offers two classes to work in frequency domain: (1) *AnalyserNode*, allowing the developer to obtain the magnitude spectra (but no phase spectra) of a time sequence, (2) *PeriodicWave*, an interface to define a periodic waveform for an *OscillatorNode* in order to perform synthesis using a similar method to the one described in [12]. Currently, there is no way to retrieve the full frequency description in a *ScriptProcessor* or *AudioWorker*. This situation requires developers to use JavaScript implements of the FFT in order to implement high quality time stretching with Phase Vocoders, instead of relying on native implementations exposed in a JavaScript API like the WAA. This leads to an increased overhead in computational costs because JavaScript is an interpreted language. This overhead can cause sudden audio dropouts when using algorithms like the Phase Vocoder, Spectral Modelling or Percussive-Harmonic Separation [17] within a *ScriptProcessor*.[18] And, in a Phase Vocoder, the bulk of the computational cost is due to the FFT. This absence caused significant discussion within the WAA community.[19]

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the existing time stretching and pitch shifting implementations using web technologies, as well as two new implementations. Then, we compared the implementations regarding computational costs and the existence of audio artefacts. Additionally, we commented on the current state of the WAA specification regarding frequency operations and how that affects time stretching and pitch shifting.

Our next step regarding OLA-TS.js and PhaseVocoder.js will be to integrate the implementations in creative frameworks and applications like Flocking and EarSketch. Additionally, we plan on integrating a re-sampler in the helper classes in order to provide pitch shifting. For PhaseVocoder.js, we plan to include new audio effects like robotization and whiperization.

[15] http://asmjs.org

[16] *wantedSize* can not be bigger than *windowFrame.length*.

[17] We have two default functions to adjust both parameters.

[18] In Google Chrome/Chromium, if you play three or more songs, audio drop-outs occur. In Firefox, these drop-outs happen for six or more songs.

[19] https://github.com/WebAudio/web-audio-api/issues/468

Table 1: Comparison of Time Stretching/Pitch Shift JavaScript and Native implementations

Name	T. Stretch/P. Shift	Algorithm	Audio Artifacts
SoundTouch.js	Both	WSOLA + Re-Samplig	Missing Transients
WAVES Audio library	Both	G. Synthesis + Re-Sampling	Smeared Transients
pitchshift.js	Pitch Shift	Phase Vocoder	None
jungle.js	Pitch Shift	Delay-Line Modulation	Reverberation
Vexwarp	Time Stretch	Phase Vocoder	Metal Tunnel
tempo-sox.js	Time Stretch	WSOLA	Unknown
PhaseVocoder.JS	Time Stretch	Phase Vocoder	None
OLA-TS.JS	Time Stretch	Modified OLA	Some modulation in harm. structures
Firefox Audio	Time Stretch	WSOLA	Missing Transients
Chromium Audio	Time Stretch	WSOLA	None

8. REFERENCES

[1] KievII: GUI Javascript library, for web audio applications https://github.com/janesconference/KievII/ (developers), 2013.

[2] C. Clark and A. Tindale. Flocking: A Framework for Declarative Music-Making on the Web. In *Proceedings of the International Computer Music Conference*, pages 1550–1557, Athens, 2014.

[3] A. de Gotzen, N. Bernardini, and D. 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, December 2000.

[4] S. Disch and U. Zolzer. Modulation and Delay Line Based Digital Audio Effects. In *Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects (DAFx99)*, NTNU, Trondheim, December 1999.

[5] M. Dolson. The phase vocoder: A tutorial. *Computer Music Journal*, 10(4):14–27, 1986.

[6] T. Karrer, E. Lee, and J. Borchers. Phavorit: A phase vocoder for real-time interactive time-stretching. In *Proceedings of the International Computer Music Conference*, pages 708–715, New Orleans, USA, November 2006.

[7] T. Karrer, E. Lee, and J. Borchers. An Analysis of Startup and Dynamic Latency in Phase Vocoder-Based Time-Stretching Algorithms. In *Proceedings of the International Computer Music Conference*, pages 73–80, Copenhagen, Denmark, August 2007.

[8] J. Laroche and M. Dolson. Phase-vocoder: about this phasiness business. In *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 4–8, 1997.

[9] J. Laroche and M. Dolson. Improved phase vocoder time-scale modification of audio. *IEEE Transactions on Speech and Audio Processing*, 7(3):323–332, 1999.

[10] J. Laroche and M. 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*, pages 91–94, 1999.

[11] A. Mahadevan, J. Freeman, and B. Magerko. EarSketch: Teaching computational music remixing in an online Web Audio based learning environment. In *1st Web Audio Conference*, Paris, 2015.

[12] J. A. Moorer. The Synthesis of Complex Audio Spectra by Means of Discrete Summation Formulas. *Journal of Audio Engineering Society*, 24(9):717–727, 1976.

[13] S. Pieters, A. van Kesteren, P. Jagenstedt, D. Denicola, I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, T. Atkins, S. Pieters, Y. Weiss, and M. Marquis. HTML 5.1, W3C Working Draft (http://www.w3.org/TR/html51/), 2015.

[14] M. Roelands and W. Verhelst. Waveform similarity based overlap-add (WSOLA) for time-scale modification of speech: structures and evaluation. In *EUROSPEECH*, 1993.

[15] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmith. Of Time Engines and Masters - An API for Scheduling and Synchronizing the Generation and Playback of Event Sequences and Media Streams for the Web Audio API. In *1st Web Audio Conference*, Paris, 2015.

[16] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. 2011.

[17] H. Tachibana, N. Ono, H. Kameoka, and S. Sagayama. Harmonic/percussive sound separation based on anisotropic smoothness of spectrograms. *IEEE/ACM Transactions on Audio, Speech and Language Processing*, 22(12):2059–2073, December 2014.

[18] Teoli and C. Mills. Web Audio playbackRate explained (https://developer.mozilla.org/en-US/Apps/Build/Audio_and_video_delivery/WebAudio_playbackRate_explained), 2014.

[19] A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, Portland, OR, USA, October 2011.

[20] U. Zolzer. *DAFX: Digital Audio Effects, second edition*. Wiley, New Jersey, 2011.