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

Bruno Dias  
INESC-ID, IST - Universidade de Lisboa, Portugal  
bruno.s.dias@ist.utl.pt

Matthew E. P. Davies  
INESC TEC, Porto, Portugal  
mdavies@inesctec.pt

David M. Matos  
INESC-ID, IST - Universidade de Lisboa, Portugal  
david.matos@inesc-id.pt

H. Sofia Pinto  
INESC-ID, IST - Universidade de Lisboa, Portugal  
sofia@inesc-id.pt

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 enable DJs to speed up or slow down songs in order to mix them by aligning the beats. Unfortunately, there are few (and experimental) client-side JavaScript implementations of these two operations. In this paper, we review the current state of the art for client-side implementations of time stretching and pitch shifting, their limitations, and describe 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\(^4\) can only change the duration of a signal by 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\(^5\), time stretching and pitch shifting are performed server-side, with SoX\(^6\) making real-time interaction impossible.

Our aim in this work is to enable 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 the subsequent impact on performance.

For both implementations, we have 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 as follows. First, we review the basic theory for time and frequency domain time stretching and pitch shifting algorithms. Then, we describe the existing Javascript implementations for each. After that, we present our two new implementations for two well know algorithms: (1) Phase Vocoder with Identity Phase Locking (OPL) 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)\(^12\) and delay line modulation\(^4\), and others that work in the frequency domain, like the Phase Vocoder\(^5\) and Spectral Modelling\(^15\). 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 ver-
sions of OLA. For this reason, we start with the basic OLA algorithm.

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.
\[
x_i(n) = x(n + i \cdot H_a), \quad x_i \in \mathbb{R}^N
\] (1)

2. Apply a window \( w \) (e.g.: Hamming window) to each analysis frame
\[
x_{w_i}(n) = w(n) \cdot x_i(n), \quad w \in \mathbb{R}^N
\] (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 \( y \)
\[
y(n + i \cdot H_s) = \begin{cases} y_i(n), & \text{if } i = 0 \\ y(n + i \cdot H_s) + \frac{w(n) \cdot y_i(n)}{w(n)}, & \text{if } i > 0 \end{cases}
\] (3)

with \( y_i = x_{w_i} \) for the basic OLA, \( n \in \mathbb{N} \land 0 \leq n \leq N \) and \( i \in \mathbb{N} \land i \cdot H_s \leq M \land i \cdot H_a \leq L \). The relation between both overlap/hop sizes and the stretching factor is defined as:
\[
\alpha = \frac{H_a}{H_s}.
\] (4)

Usually, one of the hop sizes is fixed while the other is a function of \( \alpha \). We should note that \( \alpha \) can change at each new frame, which is particularly relevant for real-time audio processing. OLA does not preserve phase relations between consecutive frames and, as such, there are noticeable artifacts in the output signal: modulation of harmonic structures (e.g.: human voice) and reverberation. The temporal complexity of OLA is \( O(N) \). In figure [3](a), we can see what happens to both \( H_a \) and \( H_s \) when changing \( \alpha \).

To perform pitch shifting with OLA, we could couple a re-sampler to an OLA time stretcher. In order to allow simultaneous time stretching and pitch shifting, let \( t \) and \( t + 1 \) be the current and next frames, \( \beta_t \) be the desired pitch and \( R_t \) the sampling rate of the input signal. Then, the new stretching factor \( \alpha_{t+1} \) and the new sample rate \( R_{t+1} \) are defined as
\[
R_{t+1} = R_t \cdot \beta_{t+1} \quad \text{(5)}
\]
\[
\alpha_{t+1} = \frac{\alpha_t}{R_t} \quad \text{(6)}
\]

2.2 WSOLA

First introduced in [12], WSOLA applies a delay \( \delta \in [-\delta_{\text{max}} : \delta_{\text{max}}] \) to each analysis frame, such that the waveforms of two overlapping synthesis frames are as similar as possible in the overlapping regions, where WSOLA is equivalent to OLA when \( \delta_{\text{max}} = 0 \). 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
\[
x_i(n) = x(n + H_a \cdot \delta_i)
\] (7)

where \( H_a \) is the analysis hop size for frame \( i \) and is defined as
\[
H_a = i \cdot H_a + \delta_i
\] (8)

and
\[
\delta_i = x_t \cdot x_{t-1} - [\delta_{\text{max}}]
\] (9)

where \( \delta_{\text{max}} < H_a \). This algorithm removes reverberation and modulation but, for transient rich sounds (percussion, guitar riffs), there might be some missed transients and stuttering (i.e.: repeated transients). Regarding the temporal complexity of WSOLA, if the cross-correlation is implemented with FFT Convolution [14], we get \( O(N \log N) \). Else, with 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, 8, 18] algorithm used for time stretching [5], pitch shifting [8] and other audio effects like robotization [18]. 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 \( x_{w_i} \)
\[
X_i(n, \Omega_k) = \sum_{n=0}^{N} x_{w_i}(n) \cdot e^{-j\Omega_k n}, X_i \in \mathbb{C}^N
\] (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 \( \Omega_k \).

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 to polar coordinates.

3. Calculate the difference between current and previous phase spectra and, then, the sample-wise difference with the frequency centres \( \Omega_k \)
\[
\Delta_{\angle} X_i = \angle X_i - \angle X_{i-1} - H_a \cdot \Omega_k
\] (11)

4. Because phase values are given in modulo \( 2\pi \), and, as such, phase ‘jumps’ can occur, we unwrap the phase in order to obtain a continuous phase function
\[
\Delta\angle X_i = \Delta_{\angle} X_i - 2\pi \cdot \left\lfloor \frac{\Delta_{\angle} X_i}{2\pi} \right\rfloor
\] (12)

5. Compute the instantaneous frequency \( \omega \) for each bin \( k \)
\[
\omega_k = \Omega_k + \frac{\Delta\angle X_i}{H_a}
\] (13)

6. Use \( \omega_k \) to compute the output phase spectra \( \angle Y_i \) by advancing the previous output \( \angle Y_{i-1} \) according to the synthesis hop size \( H_s \)
\[
\angle Y_i = \angle Y_{i-1} + H_s \cdot \omega_k
\] (14)

7. Compute the synthesis frame \( Y_i \in \mathbb{C}^N \) by reusing the input magnitude spectra and the new phase spectra
\[
Y_i = |X_i| \cdot e^{j\angle Y_i}
\] (15)
The main difference is within the “reading head” position. For each analysis frame $i$, the analysis hop size $H_{ai}$ is calculated according to the optimal displacement $\delta_i$, obtained through cross-correlation between analysis frame $i$ and $i-1$, for a maximum delay of $\delta_{\text{max}}$: $H_{ai} = i \cdot H_a + \delta_i$.

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

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

This algorithm ensures horizontal phase coherence, i.e.: phase continuity for the same frequency bin for 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 Identity Phase-Locking [7]. 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 [7]). 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 is re-sampling, in the same manner as OLA and WSOLA. The second is adding an additional step to the phase correction, as described in [8], which proceeds with the following steps. After identifying the spectral peaks, for a pitch shift factor $\beta$, 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 [11], via its playbackRate attribute [16].

For pitch shift only implementations in JavaScript, there is pitchshift.js [1] and jungle.js [7]. The first is a port of a C++ implementation [4] of the Phase Vocoder for pitch shifting [8] while the second is an implementation of [4].

In Vexwarp [4] time stretching is performed with the basic phase vocoder algorithm [5]. With this application, it is not possible to perform real-time processing of the input signal.

Soundtouch.js [3] is a port of the C++ library SoundTouch [4], a WSOLA implementation. This library performs both time stretching and pitch shifting (through re-sampling) 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 effect, using Emscripten [17]. Currently, it has no documentation and the only demo uses deprecated APIs that are no longer available.

[1] https://github.com/cwilso/Audio-Input-Effects
In the WAVES project, the Audio library supports time stretching and pitch shifting through granular synthesis and resampling, offering two classes to perform both tasks: GranularEngine and SegmentEngine. The first class 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 stretch 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, meaning that web browsers perform an additional step to ensure $\alpha$ complies 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. PROPOSED IMPLEMENTATIONS

We now 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 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 processing, 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 can be easily adapted by coupling a re-sampler to the helper classes. To maintain a static memory footprint, we used an existing circular buffer implementation, CBuffer. Even though our time stretchers are implementations of (totally) different algorithms, there is a common API to both:

- **process(Array inputFrame, CBuffer outputFrame)**: given a (mono) frame, performs a time stretching iteration and pushes $H_s$ samples in the output CBuffer.
- **get_hr**: 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 and can 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, 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 $\alpha$, 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 $\alpha$ and the real $\alpha$. 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 $\alpha$ and the real $\alpha$, 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

OLAT-S.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 $\alpha$, $Ovl(\alpha)$. 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:

$$H_a = \frac{N}{Ovl(\alpha)}$$
$$H_s = \alpha \ast H_a$$

where $Ovl(\alpha)$ is the function defined in `overlap_fn`. In order to properly stretch an 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.

#### 4.2 PhaseVocoder.js

PhaseVocoder.js uses, by default, fourier.js, an FFT library offering methods implemented in both asm.js and “raw” JavaScript. In order to use a different FFT library, the developer can use the following methods:

12. [https://github.com/trevnorris/cbuffer](https://github.com/trevnorris/cbuffer)
13. [http://asmjs.org](http://asmjs.org)
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 [10].

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 Vocoder, 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 within a `ScriptProcessor`. 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.

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 to integrate a re-sampler in the helper classes to provide pitch shifting. For PhaseVocoder.js, we plan to include new audio effects like robotization and whisperization.

Both implementations and their helper classes are public and open source, each one with its own Git repository. Additionally, we implemented a small demo page where the user can drag & drop several songs and play them simultaneously, controlling the volume and the stretching factor. Finally, we will conduct a formal evaluation and comparison of time stretching and pitch shifting implementations.

8. ACKNOWLEDGMENTS


[3] https://github.com/echo66/PhaseVocoderJS

---

---

---
This work was partly supported by national funds through FCT - Fundação para a Ciência e Tecnologia, under projects EXCL/EEI-ESS/0257/2012 and UID/CEC/50021/2013 and by post-doctoral grant SFRH/BPD/88722/2012. This work was partly supported by Luso-American Development Foundation.

9. REFERENCES

[1] KievII: GUI Javascript library, for web audio applications