TSM TOOLBOX: MATLAB IMPLEMENTATIONS OF TIME-SCALE MODIFICATION ALGORITHMS

Jonathan Driedger, Meinard Müller,

International Audio Laboratories Erlangen,*
Erlangen, Germany

{jonathan.driedger,meinard.mueller}@audiolabs-erlangen.de

ABSTRACT

Time-scale modification (TSM) algorithms have the purpose of stretching or compressing the time-scale of an input audio signal without altering its pitch. Such tools are frequently used in scenarios like music production or music remixing. There exists a large variety of different algorithmic approaches to TSM, all of them having their very own advantages and drawbacks. In this paper, we present the TSM toolbox, which contains MATLAB implementations of several conceptually different TSM algorithms. In particular, our toolbox provides the code for a recently proposed TSM approach, which integrates different classical TSM algorithms in combination with harmonic-percussive source separation (HPSS). Furthermore, our toolbox contains several demo applications and additional code examples. Providing MATLAB code on a well-documented website under a GNU-GPL license and including illustrative examples, our aim is to foster research and education in the field of audio processing.

1. INTRODUCTION

Time-scale modification (TSM) is the task of manipulating an audio signal such that it sounds as if its content was performed at a different tempo. TSM finds application for example in music remixing where it is used to adjust the playback speed of existing recordings such that they can be played simultaneously at the same tempo [1, 2]. Another field of application is the adjustment of the audio streams in video clips. For example, when generating a slow motion video, TSM can be used to synchronize the audio material with the visual content [3].

There exists a large variety of different TSM algorithms which all have their respective advantages and drawbacks. Some of the TSM procedures yield results of high perceptual quality only when applied to a certain class of audio signals. For example, 'classical' well-known TSM algorithms like WSOLA [4] or the phase vocoder [5, 6] are capable of preserving the perceptual quality of harmonic signals to a high degree, but introduce noticeable artifacts when modifying percussive signals. However, it has been shown that it is possible to substantially reduce artifacts by combining different TSM procedures. For example, in [7], a given audio signal is first decomposed into a harmonic and a percussive component. Afterwards, the two components are processed with different classical TSM algorithms, and final output signal is obtained by superimposing the two TSM results.

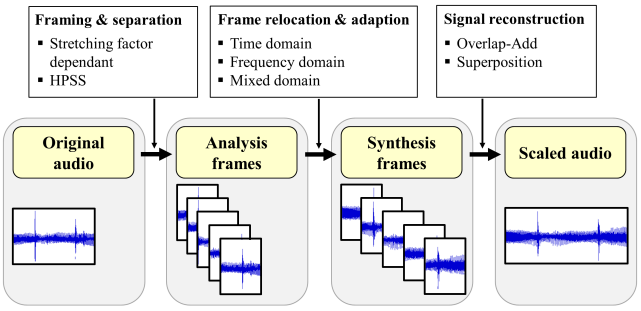


Figure 1: General processing pipeline of TSM procedures.

To foster research and to obtain a better understanding of TSM algorithms, we present in this paper the *TSM toolbox*. Published under a GNU-GPL license at [8], this self-contained toolbox serves various purposes. First, it delivers basic tools to work in the field of TSM. The toolbox includes well-documented reference implementations of the most important classical TSM algorithms within a unified framework. This not only allows users and researchers to get a better feeling for TSM results by experimenting with the algorithms, but also gives insights into implementation details and potential pitfalls. Second, to give an example of how those classical algorithms can be combined to improve TSM results, the toolbox also supplies the code of a recently proposed TSM approach based on *harmonic-percussive source separation* (HPSS), also including the code of the HPSS procedure itself. Third, the toolbox provides a MATLAB wrapper function for a commercial, proprietary, and widely used TSM algorithm. Because of its

* The International Audio Laboratories Erlangen are a joint institution of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and Fraunhofer Institut für Integrierte Schaltungen IIS.

'state-of-the-art' character, this is particularly interesting when conducting listening experiments which are the most common way of judging the perceptual quality of TSM results. Finally, the toolbox provides additional code for various example applications. Such applications include the automated generation of interfaces for comparing TSM results, the non-linear synchronization of audio recordings, and the pitch-shifting of audio signals. Although there already exist MATLAB implementations of individual TSM algorithms (for example [9, 10]), we believe that supplying an entire collection of different TSM approaches along with example applications within a unifying framework can be highly beneficial for both researchers as well as educators in the field of audio processing.

The remainder of this paper is structured as follows. In Section 2, we briefly review the basics of TSM in general as well as the TSM algorithms included in the TSM toolbox. Then, in Section 3, we describe the MATLAB functions contained in the toolbox. Some of the demo applications included in the toolbox are discussed in Section 4. Finally, in Section 5, we conclude this paper with some general remarks.

2. TIME-SCALE MODIFICATION

Most TSM procedures follow a common basic strategy which is sketched in Figure 1. Given an original audio signal x as an input, the first step of most TSM algorithms is to split up the waveform into short overlapping *analysis frames* which are spaced apart by an *analysis hopsize* H_a. In a second step, these frames are relocated on the time-axis to have a *synthesis hopsize* H_s and furthermore suitably adapted. While the relocation accounts for the actual modification of the time-scale of the audio signal, the objective of the adaption is to reduce possible artifacts introduced by the frame relocation. The modified frames, also known as *synthesis frames*, are then superimposed to form the output of the algorithm. The output signal is a time-scale modified version of the input signal x, altered in length by a constant *stretching factor* of $\alpha = H_s/H_a$.

The main differences between most procedures are therefore the strategies of how the analysis frames are chosen and how they are modified to form the synthesis frames. In the following, we review some of these strategies.

2.1. Overlap-Add (OLA)

One of the most basic TSM algorithms is known as *Overlap-Add* (OLA). In OLA, the synthesis frames are computed by just windowing the analysis frames with a window function w and not processing them any further. Although OLA is very efficient, adding up the unmodified synthesis frames usually introduces phase discontinuities into the output signal. Periodic, and therefore harmonic structures in the input signal are not preserved (see Figure 2). Perceptually, this manifests itself as strong harmonic artifacts in the output signal. However, especially when choosing the length of the analysis frames to be very short, OLA is particularly successful in preserving percussive sounds. This can be seen for example in Figure 3. Note that the sharp peak-like onsets which are visible in the original waveform (see Figure 3a) are preserved well by OLA (see Figure 3b).

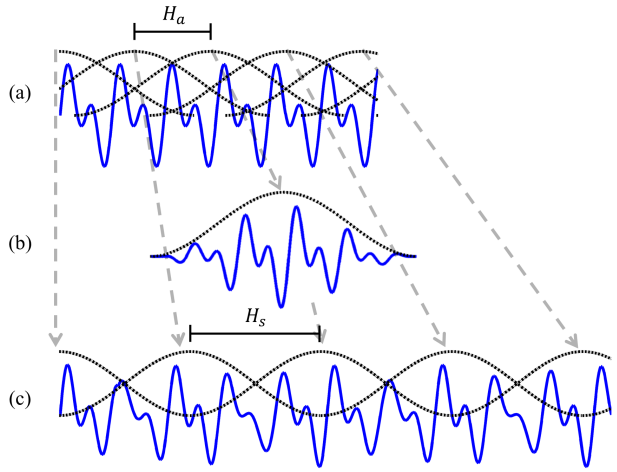


Figure 2: The principle of OLA TSM. **(a)**: Input signal x (solid line). The analysis frames are indicated by the window functions (dotted lines). **(b)**: One synthesis frame. **(c)**: Output signal as sum of all synthesis frames.

2.2. Waveform Similarity Overlap-Add (WSOLA)

One way of avoiding phase discontinuities as introduced by OLA is to choose the analysis frames such that successive synthesis frames better fit together when adding them up. The *Waveform Similarity Overlap-Add* algorithm (WSOLA) [4] achieves this by introducing an *analysis frame position tolerance* $\Delta_{\max}$. The position of each analysis frame in the input signal may be shifted on the time-axis by some $\Delta \in [-\Delta_{\max}:\Delta_{\max}]$ such that the waveforms of two overlapping synthesis frames are as similar as possible in the overlapping regions. Afterwards, the frames are windowed as in OLA and added up to form the output signal. Note that WSOLA reduces to OLA when using $\Delta_{\max} = 0$. The introduced tolerance for the analysis frames strongly reduces artifacts resulting from phase discontinuities. However, especially at transients in the input signal, the algorithm introduces noticeable *stuttering artifacts* in the output signal. These artifacts originate from shifted frame positions which tend to *cluster* around transients in the input signal. In the output signal, the transients

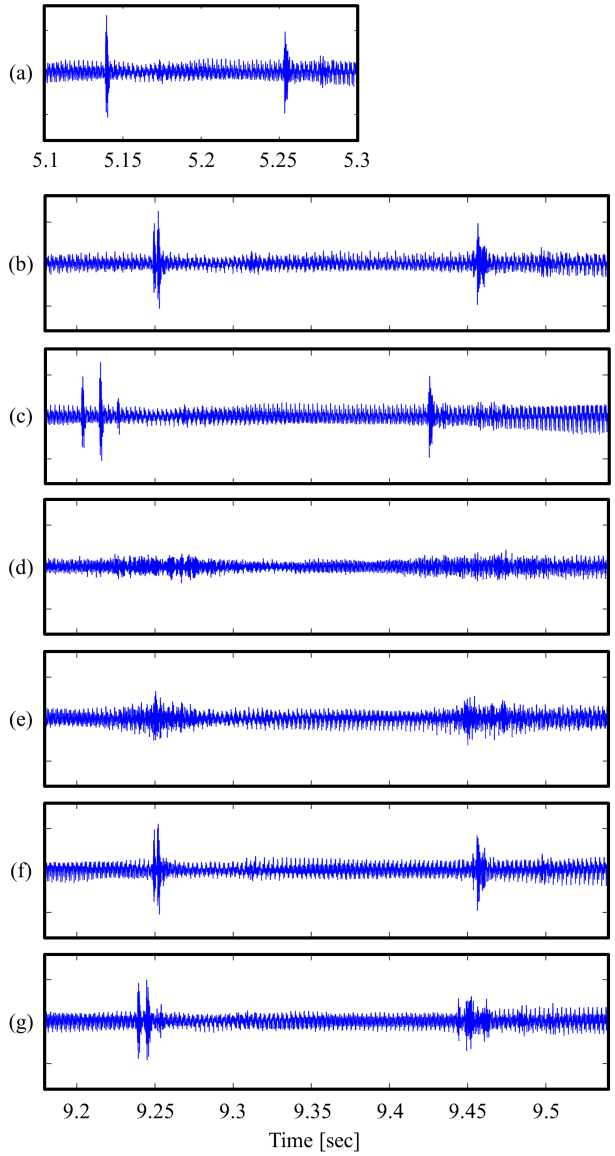


Figure 3: TSM results of different algorithms for an audio recording of a violin and castanets. **(a)**: Original waveform. **(b)**: OLA. **(c)**: WSOLA. **(d)**: Phase vocoder. **(e)**: Phase vocoder with identity phase locking. **(f)**: TSM based on HPSS. **(g)**: TSM based on the commercial *élastique* algorithm.

are therefore duplicated several times which results in the stuttering sound. For example, in Figure 3c, the first transient is repeated three times with different amplitudes.

2.3. Phase Vocoder

While WSOLA approaches the problem of phase discontinuities in the time-domain, the problem can also be targeted in the frequency-domain. The core idea of the *phase vocoder* [5, 6] is to see each analysis frame as a weighted sum of sinusoids with known frequency and phase. The synthesis frames are then computed by adapting the phases of these sinusoids such that no phase discontinuities are introduced when adding up the relocated synthesis frames.

In the first step of the procedure the Fourier transform is applied to every analysis frame resulting in a sequence of frequency spectra. Each frequency bin of a spectrum represents a sinusoid that contributes to the original signal. Afterwards, the *instantaneous frequencies* of the spectrum's frequency bins are computed from the phase differences of successive spectra, see [11]. Knowing the instantaneous frequencies and the synthesis hopsize H_s, the phases of the spectra can be adapted accordingly. Finally, all spectra are brought back to the time-domain by applying the inverse Fourier transform with the resulting waveforms constituting the synthesis frames. Note that the term "phase vocoder" generally describes the technique to estimate the instantaneous frequencies in an audio signal. However, the term is also frequently used to name the TSM algorithm.

By design, the phase vocoder guarantees phase continuity of all sinusoidals contributing to the output signal, which is also known as *horizontal phase coherence*. However, the *vertical phase coherence*, meaning the phase relationships of sinusoidals within one frame, is usually destroyed in the phase adaption process. Transients, which are highly dependent on preserving the vertical phase coherence of the signal, are therefore often *smeared* in phase vocoder TSM results, see Figure 3d for an example. The loss of vertical phase coherence also causes a very distinct sound coloration of phase vocoder TSM results known as *phasiness* [12].

2.4. Phase Vocoder with Identity Phase Locking

To reduce the loss of vertical phase coherence in the phase vocoder, Laroche and Dolson proposed a modification to the standard phase vocoder TSM algorithm [13]. Their core idea is to not adapt the phases of all frequency bins in the short-time Fourier spectra independently of each other. Instead, bins which contribute to the same partial of the audio signal are grouped. A peak in the magnitude spectrum is assumed to represent one partial of the audio signal, while the bins surrounding the peak are assumed to contribute to this partial as well. In the phase adaption process, only the frequency bins which contain spectral peaks are updated in the usual phase vocoder fashion. The phases of the remaining frequency bins are then *locked* to the phase of the closest spectral peak and the vertical phase coherence is therefore locally preserved. This technique, also known as *identity phase locking* leads to reduced phasiness artifacts and also to less transient smearing, see Figure 3e for an example.

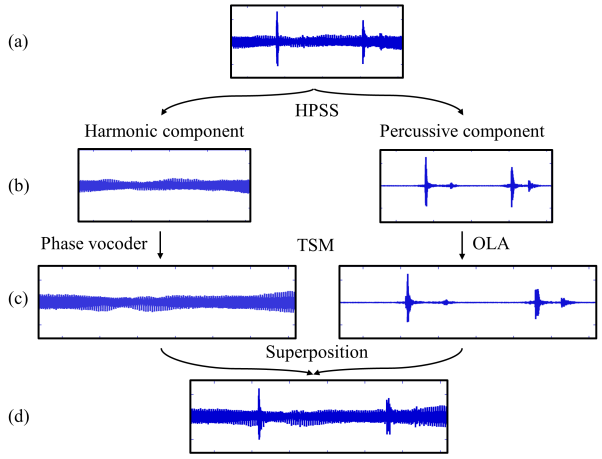


Figure 4: Overview of TSM based on HPSS. **(a)**: Input audio signal. **(b)**: Separation in harmonic component (left) and percussive component (right). **(c)**: TSM results for the harmonic component using the phase vocoder (left) and for the percussive component using OLA (right). **(d)**: Superposition of the TSM results from (c).

2.5. Combined TSM based on HPSS

TSM algorithms like the phase vocoder work particularly well for audio signals with harmonic content, while other algorithms like OLA are well suited for percussive signals. The authors of [7] therefore proposed a combined TSM approach using harmonic-percussive source separation (HPSS) techniques.

In HPSS, the goal is to decompose a given audio signal into a signal consisting of all harmonic sound components and a signal consisting of all percussive sounds. Fitzgerald [14] proposed a simple and effective HPSS procedure. This method exploits the fact that in a spectral representation of a signal, harmonic sounds form structures in time direction, while percussive sounds yield structures in frequency direction. By applying a median filter of length ℓ_h in time direction and a median filter of length ℓ_p in frequency direction to the magnitude spectrogram of the input signal, the respective structures are enhanced. Afterwards, by comparing the two filtered spectra element wise, each time-frequency instance of the signals spectrum can be assigned to either the harmonic or the percussive portion of the signal. This yields in the end the desired components.

After having decomposed the input signal using this HPSS method, the authors of [7] apply the phase vocoder with identity phase locking to the harmonic component and OLA to the percussive component. By treating the two components separately, both the characteristics of the harmonic sounds as well as the percussive sounds of the input signal can be preserved. The superimposed TSM results of both procedures finally form the output of the algorithm (see Figure 4). Note that there also exist other approaches to preserve both characteristics. For example, algorithms employing *transient preservation* aim for explicitly identifying the time positions of percussive events in the audio signal and giving them a special treatment in the TSM process [15, 16]. Such a strategy can also easily be integrated into the MATLAB code provided in the TSM toolbox.

2.6. TSM based on élastique

Besides these publically known TSM algorithms, there also exists a number of proprietary commercial products. One of these commercially available TSM algorithms, called *élastique*, has been developed by zPlane [17]. This algorithm, which is integrated in a wide range of music software like *Steinberg Cubase*[1] or *Ableton Live*[2], can be considered the state-of-the-art in the field of commercial TSM algorithms. An example of an audio signal stretched with *élastique* is shown in Figure 3g. In addition to the usual licensing model for their algorithm, the developers also offer a web-based interface called *sonicAPI*[3], which allows users to compute the TSM results for *élastique* over the internet. At least for the time being, this service is free of charge for personal usage. A MATLAB wrapper function for this webservice is included in the TSM toolbox.

3. TOOLBOX

The TSM algorithms as described in Section 2 form the core of our TSM toolbox, which is freely available at the website [8] under a GNU-GPL license. Table 1 gives an overview of the main MATLAB functions along with the most important parameters. Note that there are many more parameters and additional functions not discussed in this paper. However, for all parameters there are default settings such that none of the parameters need to be specified by the user.

To demonstrate how the TSM algorithms contained in our toolbox can be applied, we now discuss the code example shown in Table 2, which is also contained in the toolbox as script `demoTSMtoolbox.m`. Our example starts in lines 1-4 with specifying an audio signal as well as a time-stretch factor α. Furthermore, the audio signal is loaded from the hard disk using the MATLAB function `wavread` and stored in the variable `x` while its sampling rate is stored in `sr`.

The first TSM algorithm which is applied to the loaded signal is OLA in lines 6-10. Since OLA is a special case of WSOLA, this is done by calling the `wsolaTSM.m` function with a specialized set of parameters. In line 6, the analysis frame position tolerance $\Delta_{\max}$ of WSOLA is set to 0, turning WSOLA into OLA. Afterwards, the synthesis hopsize

[1] `http://www.steinberg.net`
[2] `https://www.ableton.com`
[3] `http://www.sonicapi.com/`

Filename	Main parameters	Optional parameters	Description
`wsolaTSM.m`	x, α	synHop$\hat{=}H_s$, win$\hat{=}w$, tolerance$\hat{=}\Delta_{\max}$	Application of OLA & WSOLA.
`pvTSM.m`	x, α	synHop$\hat{=}H_s$, win$\hat{=}w$, phaseLocking	Application of the phase vocoder (with or without identity phase locking).
`hpTSM.m`	x, α	hpsFilLenHarm$\hat{=}\ell_h$, hpsFilLenPerc$\hat{=}\ell_p$, pvSynHop, pvWin, olaSynHop, olaWin	Application of TSM based on HPSS.
`elastiqueTSM.m`	x, α	–	MATLAB wrapper for the *élastique* algorithm.
`win.m`	ℓ, β	–	Generates a $\sin^\beta$ window function of length ℓ.
`stft.m`	x	anaHop, win	Short-time Fourier transform of x.
`istft.m`	spec	synHop, win	Inversion of a short-time Fourier transform, see [18].
`hpSep.m`	x	filLenHarm$\hat{=}\ell_h$, filLenPerc$\hat{=}\ell_p$	Harmonic-percussive source separation.
`pitchShiftViaTSM.m`	x, n	algTSM	Pitch-shifting the signal x by n cents.
`visualizeWav.m`	x	fsAudio, timeRange	Visualization of TSM results.
`visualizeSpec.m`	spec	fAxis, tAxis, logComp	Visualization of a short-time Fourier transform.
`visualizeAP.m`	anchorpoints	fsAudio	Visualization of a set of anchorpoints.

Table 1: Overview of the main MATLAB functions contained in the TSM toolbox [8] and the most important parameters.

H_s is set to 128 samples in line 7. In lines 8 and 9, a $\sin^\beta$-window of length $\ell = 256$ samples and $\beta = 2$ is generated by calling `win.m`. The size of the generated window specifies at the same time the size of the analysis and synthesis frames. Together with the synthesis hopsize of 128 samples this means that in the output of the TSM algorithm the synthesis frames will have a half-overlap of 128 samples. Finally the actual TSM algorithm is applied to the input signal x with the stretching factor α and the specified set of parameters in line 10. The resulting waveform is stored in the variable `yOLA`.

Next, in lines 12-16, the WSOLA algorithm is applied. We first set the analysis frame position tolerance $\Delta_{\max}$ to 512 in line 12. Since WSOLA works optimally for medium sized frames which are half-overlapped, we set the synthesis hopsize H_s to 512 in line 13 and chose a $\sin^\beta$-window of length $\ell = 1024$ samples and $\beta = 2$ in lines 14 and 15. Finally, the function `wsolaTSM.m` is called in line 16.

In lines 18-22 the standard phase vocoder is applied by a call of `pvTSM.m`. To this end, we first specify that no phase locking should be applied (line 18). Being a frequency-domain TSM algorithm, the phase vocoder is dependent on a high frequency resolution of the used Fourier transform and therefore on a large frame size. Furthermore, also a large overlap of the synthesis frames is beneficial for the quality of the output signal as well as a sin-window function. We therefore set the synthesis hopsize H_s to 512 (line 19) and chose a $\sin^\beta$-window of length $\ell = 2048$ samples and $\beta = 1$ (lines 20 and 21), resulting in a 75% frame overlap. The actual function call is then executed in line 22. For the application of the phase vocoder with identity phase locking in lines 24-28, the only difference is the `phaseLocking` parameter set to one (line 24).

The TSM algorithm based on HPSS, which is applied in lines 30-38, is a combination of multiple techniques. First, we set the length of the median filters ℓ_h and ℓ_p used in the HPSS procedure both to 10 (lines 30 and 31). Then, the synthesis hopsizes and windows, which are used in the two TSM algorithms OLA and phase vocoder with identity phase locking, are set separately in lines 32-37. In line 38 the algorithm is then executed by a call of `hpTSM.m`.

The last TSM algorithm is the MATLAB wrapper for *élastique*. Since this function requires a *sonicAPI* access id as well as the additional tool `curl`, the function call in line 44 is commented out by default. However, when supplying the additional sources the algorithm can be applied by a call to `elastiqueTSM.m`. Since *élastique* is a proprietary procedure it is not possible to tweak the algorithm with additional parameters.

In lines 46 and 47, the visualization of the original input signal takes place. First, the segment of the input audio signal to be visualized is set to the section of the waveform between second 5.1 and 5.3 (line 46). Afterwards the visualization function `visualizeWav.m` is applied to `x` in line 47. To visualize the corresponding stretched audio segment, the segments boundaries are just multiplied with the stretching factor α in line 48. Afterwards, the visualization function is called again exemplarily for OLA's TSM result in line 49. Finally, in line 50, the TSM result of OLA is also written to the hard disk using the MATLAB function `wavwrite`.

4. APPLICATIONS

In this section, we discuss some additional functionalities of the TSM toolbox, including some demo applications.

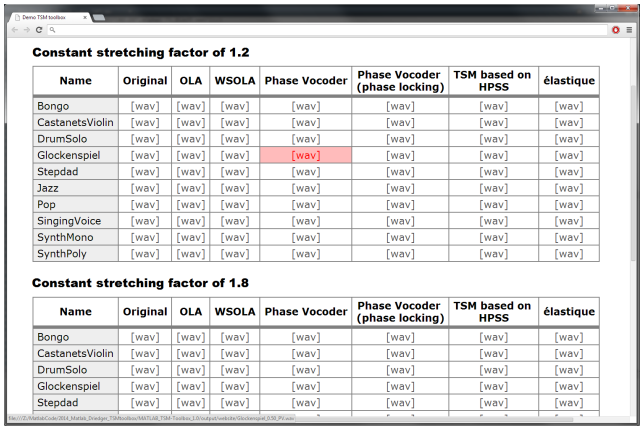
4.1. Interface Generation

The most common way of comparing the quality of different TSM algorithms is by performing listening experiments. To this end, one usually generates time-stretched versions

```
filename = 'CastanetsViolin.wav';
alpha = 1.8;

[x,sr] = wavread(filename);

paramOLA.tolerance = 0;
paramOLA.synHop = 128;
len = 256; beta = 2;
paramOLA.win = win(len,beta);
yOLA = wsolaTSM(x,alpha,paramOLA);

paramWSOLA.tolerance = 512;
paramWSOLA.synHop = 512;
len = 1024; beta = 2;
paramWSOLA.win = win(len,beta);
yWSOLA = wsolaTSM(x,alpha,paramWSOLA);

paramPV.phaseLocking = 0;
paramPV.synHop = 512;
len = 2048; beta = 1;
paramPV.win = win(len,beta);
yPV = pvTSM(x,alpha,paramPV);

paramPVpl.phaseLocking = 1;
paramPVpl.synHop = 512;
len = 2048; beta = 1;
paramPVpl.win = win(len,beta);
yPVpl = pvTSM(x,alpha,paramPVpl);

paramHP.hpsFilLenHarm = 10;
paramHP.hpsFilLenPerc = 10;
paramHP.pvSynHop = 512;
len = 2048; beta = 1;
paramHP.pvWin = win(len,beta);
paramHP.olaSynHop = 128;
len = 128; beta = 2;
paramHP.olaWin = win(len,beta);
yHP = hpTSM(x,alpha,paramHP);

% To execute elastique, you will need
% an access id from http://www.sonicapi.com.
% Furthermore, you need to download 'curl'
% from http://curl.haxx.se/download.html.
% yELAST = elastiqueTSM(x,alpha);

paramVis.timeRange = [5.1 5.3];
visualizeWav(x,paramVis);
paramVis.timeRange = [5.1 5.3] * alpha;
visualizeWav(yOLA,paramVis);
wavwrite(yOLA,sr,'Output_OLA.wav'
```

Table 2: Code example for computing TSM results of various TSM algorithms, generating the visualizations, and writing the TSM results to the hard disk.

of several audio items using different TSM algorithms and stretching factors. This results in large amounts of audio data. To be able to compare the generated TSM results, interfaces which allow a user to order and access the audio signals in a convenient way are of great help. With the script `demoGenerateTSMwebsite.m`, which is contained in the TSM toolbox, we provide the code for generating such a HTML-based interface automatically (see Figure5). The toolbox also includes the set of audio items listed in Table 3, which has been already used for evaluation purposes in the context of TSM in [7, 19].



Constant stretching factor of 1.2

Name	Original	OLA	WSOLA	Phase Vocoder	Phase Vocoder (phase locking)	TSM based on HPSS	élastique
Bongo	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
CastanetsViolin	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
DrumSolo	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
Glockenspiel	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
Stepdad	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
Jazz	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
Pop	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
SingingVoice	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
SynthMono	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
SynthPoly	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]

Constant stretching factor of 1.8

Name	Original	OLA	WSOLA	Phase Vocoder	Phase Vocoder (phase locking)	TSM based on HPSS	élastique
Bongo	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
CastanetsViolin	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
DrumSolo	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
Glockenspiel	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]
Stepdad	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]	[wav]

Figure 5: Screenshot of the interface generated using the function `demoGenerateTSMwebsite.m` of the TSM toolbox.

Item name	Description
Bongo	Regular beat played on bongos.
CastanetsViolin	Solo violin overlayed with castanets.
DrumSolo	A solo performed on a drum set.
Glockenspiel	Monophonic melody played on a glockenspiel.
Jazz	Synthetic polyphonic sound mixture of a trumpet, a piano, a bass and drums.
Pop	Synthetic polyphonic sound mixture of several synthesizers, a guitar and drums.
SingingVoice	Solo male singing voice.
Stepdad	Excerpt from *My Leather, My Fur, My Nails* by the band *Stepdad*.
SynthMono	Monophonic synthesizer with a very noisy and distorted sound.
SynthPoly	Sound mixture of several polyphonic synthesizers.

Table 3: List of audio items included in the TSM toolbox.

4.2. Non-linear Time-Scale Modification

In addition to stretching audio signals in a linear fashion by a constant stretching factor α, the implementations contained in the TSM toolbox (except for *élastique*) are also capable of stretching input signals in a non-linear way. To this end, one needs to define a *time-stretch function* which defines the mapping between time-positions in the input signal and the output signal of the TSM algorithm. A very convenient way of defining such a time-stretch function is by specifying a set of *anchorpoints*. An anchorpoint is a pair of time positions where the first entry specifies a time-position in the input signal and the second entry a time-position in the output signal. The actual time-stretch function is then obtained by a linear interpolation between the anchorpoints. In Figure 6, one can see an example of such a non-linear modification. In Figure 6b, we see the waveforms of two recorded performances of the first five measures of Beethoven's Symphony No. 5. The corresponding time-positions of the note onsets are indicated by red arrows. Ob-

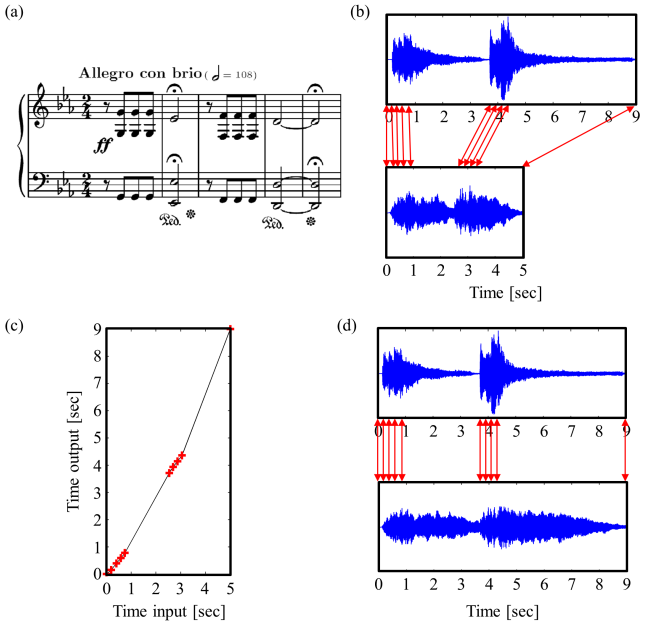


Figure 6: **(a)**: Score of the first five measures of Beethoven's Symphony No. 5. **(b)**: Waveforms of two performances. Corresponding onset positions are indicated by the red arrows. **(c)**: Set of anchorpoints. **(d)**: Onset-synchronized waveforms of the two performances, where the second performance was modified.

viously, the two performances differ strongly in their length. However, the tempo of the two performances does not differ by some constant factor. In fact, the tempo of the eighth notes in the first and third measure are played at almost the same tempo in both performances. Contrary, the durations of the half notes with fermata in measures two and five differ strongly in the two recordings. The mapping between the note onsets of the two performances is therefore non-linear. We define eight anchorpoints, which map the onset positions of the second performance to the onset positions of the first performance (plus two additional anchorpoints, which align the beginning and the end of the waveforms). Based on these anchorpoints, we then apply one of the TSM algorithms in the TSM toolbox to the second performance to obtain a version of the recording which is onset-synchronized with the first performance, see Figure 6d. The MATLAB code for this example, which also generates sonifications of the synchronization result, is also contained in the TSM toolbox in the file `demoNonlinearTSM.m`. In this example, the anchorpoints were chosen manually. However, one can also compute alignments between two recordings automatically and derive anchorpoints from them, see for example [20]. This functionality can, for example, be used in scenarios like *automated soundtrack generation* [21] or *automated DJing* [1, 2].

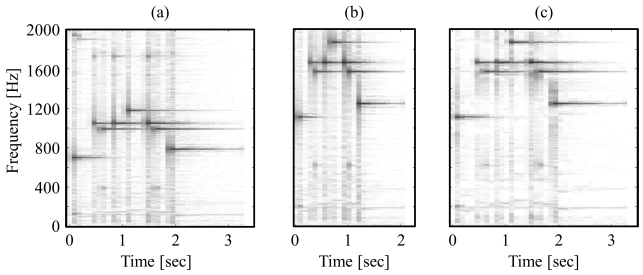


Figure 7: Pitch-shifting via resampling and TSM. **(a)**: Spectrogram of an input audio signal. **(b)**: Spectrogram of the resampled signal. **(c)**: Spectrogram after TSM application.

4.3. Pitch-Shifting

Pitch-shifting is the task of changing the pitch of an audio recording without altering its length. It can therefore be seen as the dual problem to TSM. While there exist specialized pitch-shifting algorithms [22, 23], it is also possible to approach the problem by combining TSM algorithms with resampling. Here, the core observation is, that stretching or compressing the whole waveform of an audio signal changes the length and the pitch of the signal at the same time. With vinyl records, this can for example be simulated by changing the rotation speed of the record player. In the world of digital audio signals, the same effect can be achieved by resampling a given signal. To this end, a given audio signal, sampled at a frequency of f_{in}, is resampled to have a new sampling frequency f_{out}. When playing back the resampled signal at the old sampling frequency f_{in}, this changes the pitch of the signal by $\log(f_{\text{in}}/f_{\text{out}})/\log(\sqrt[12]{2})$ semitones, as well as its length by a factor of $f_{\text{out}}/f_{\text{in}}$. To demonstrate this, we show an example in Figure 7. Here, the goal is to apply a pitch-shift of 8 semitones to the input audio signal. The original signal has a sampling frequency of f_{in}=44100 Hz (Figure 7a). To achieve a pitch-shift of 8 semitones, the signal is resampled to f_{out}=27781 Hz (Figure 7b). One can see, that the resampling changed the pitch of the signal as well as its length. While the change in pitch is desired, the change in length needs to be compensated. This can be done using a TSM algorithm at hand (Figure 7c). However, the quality of the pitch-shifting result crucially depends on the quality of the TSM algorithm. The MATLAB function `pitchShiftViaTSM.m`, which employs the above described strategy for pitch-shifting, is contained in the TSM toolbox. Furthermore, the script `demoPitchShift.m` gives an example of how this function can be applied.

5. CONCLUSIONS

In this paper, we have introduced the TSM toolbox, a unifying MATLAB framework which contains several TSM al-

gorithms, various code examples for demo applications, as well as audio material that has already been used for evaluating TSM algorithms. We hope that this toolbox not only provides a solid code basis to work in the field of TSM, but also helps to raise the awareness for potential problems of classical TSM algorithms, to foster the development of new TSM techniques, and to ease the design of listening experiments. Finally, we would like to encourage developers and researchers in the field of audio processing and music information retrieval to use the toolbox to realize their ideas of applications involving TSM of audio signals.

6. REFERENCES

[1] Dave Cliff, "Hang the DJ: Automatic sequencing and seamless mixing of dance-music tracks," Tech. Rep., HP Laboratories Bristol, 2000.

[2] Hiromi Ishizaki, Keiichiro Hoashi, and Yasuhiro Takishima, "Full-automatic DJ mixing system with optimal tempo adjustment based on measurement function of user discomfort," in *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)*, Kobe, Japan, 2009, pp. 135–140.

[3] Alexis Moinet, Thierry Dutoit, and Thierry Dutoit Alexis Moinet, "Audio time-scaling for slow motion sports videos," in *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx)*, Maynooth, Ireland, September 2013.

[4] Werner Verhelst and Marc Roelands, "An overlap-add technique based on waveform similarity (WSOLA) for high quality time-scale modification of speech," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Minneapolis, USA, 1993.

[5] James L. Flanagan and R. M. Golden, "Phase vocoder," *Bell System Technical Journal*, vol. 45, pp. 1493–1509, 1966.

[6] M. R. Portnoff, "Implementation of the digital phase vocoder using the fast fourier transform," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 24, no. 3, pp. 243–248, 1976.

[7] Jonathan Driedger, Meinard Müller, and Sebastian Ewert, "Improving time-scale modification of music signals using harmonic-percussive separation," *Signal Processing Letters, IEEE*, vol. 21, no. 1, pp. 105–109, 2014.

[8] Jonathan Driedger and Meinard Müller, "TSM toolbox," `http://www.audiolabs-erlangen.de/resources/MIR/TSMtoolbox/`.

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

[10] Daniel P. W. Ellis, "A phase vocoder in Matlab," `http://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/`, 2002, Web resource, last consulted in February 2014.

[11] Mark Dolson, "The phase vocoder: a tutorial," *Computer Musical Journal*, vol. 10, no. 4, pp. 14–27, 1986.

[12] Jean Laroche and Mark Dolson, "Phase-vocoder: about this phasiness business," in *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics, 1997*, October 1997.

[13] Jean Laroche and Mark Dolson, "Improved phase vocoder time-scale modification of audio," *IEEE Transactions on Speech and Audio Processing*, vol. 7, no. 3, pp. 323–332, 1999.

[14] Derry Fitzgerald, "Harmonic/percussive separation using median filtering," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, Graz, Austria, 2010, pp. 246–253.

[15] Frederik Nagel and Andreas Walther, "A novel transient handling scheme for time stretching algorithms," in *127th Audio Engineering Society Convention 2009*, New York, NY, 2009, pp. 185–192.

[16] Shahaf Grofit and Yizhar Lavner, "Time-scale modification of audio signals using enhanced WSOLA with management of transients," *IEEE Transactions on Audio, Speech & Language Processing*, vol. 16, no. 1, pp. 106–115, 2008.

[17] zplane development, "élastique time stretching & pitch shifting SDKs," `http://www.zplane.de/index.php?page=description-elastique`, Web resource, last consulted in August 2013.

[18] Daniel W. Griffin and Jae S. Lim, "Signal estimation from modified short-time Fourier transform," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 32, no. 2, pp. 236–243, 1984.

[19] Jonathan Driedger, Meinard Müller, and Sebastian Ewert, "Accompanying website: Improving time-scale modification of music signals using harmonic-percussive separation," `http://www.audiolabs-erlangen.de/resources/2014-SPL-HPTSM/`, Web resource, last consulted in March 2014.

[20] Sebastian Ewert, Meinard Müller, and Peter Grosche, "High resolution audio synchronization using chroma onset features," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009, pp. 1869–1872.

[21] Meinard Müller and Jonathan Driedger, "Data-driven sound track generation," in *Multimodal Music Processing*, Meinard Müller, Masataka Goto, and Markus Schedl, Eds., vol. 3 of *Dagstuhl Follow-Ups*, pp. 175–194. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2012.

[22] Azadeh Haghparast, Henri Penttinen, and Vesa Välimäki, "Real-time pitch-shifting of musical signals by a time-varying factor using normalized filtered correlation time-scale modification," Bordeaux, France, September 2007, pp. 7–14.

[23] Christian Schörkhuber, Anssi Klapuri, and Alois Sontacchi, "Audio pitch shifting using the constant-q transform," *Journal of the Audio Engineering Society*, vol. 61, no. 7/8, pp. 562–572, 2013.