SpectMorph: Morphing the Timbre of Musical Instruments

Stefan Westerfeld
Freiburg, Germany
stefan@space.twc.de

Abstract

SpectMorph is an open source software which performs morphing of the timbre of musical instruments. This allows creating sounds that smoothly transition from the timbre of one instrument to the timbre of another instrument. There are three steps necessary to obtain the final sound. In the analysis, we use the fourier transform to create models of the spectrum of the input samples. During synthesis a time domain signal can be obtained from these data. An algorithm for morphing the spectral models of multiple instruments is the core of our method. Synthesis and morphing can be done in real-time. After the description of the theoretical background, we provide an overview of the features of the SpectMorph plugin.

Keywords

Morphing, timbre, audio, spectral modelling

1 Introduction

The starting point for SpectMorph[1], our morphing software, are recordings of musical instruments. Typically samples for many different notes per instrument are used, to provide natural sound quality for different notes. From these samples we build spectral models, which are a description of the timbre of each instrument.

Once the analysis data is available, the software can combine the timbre of multiple instruments. A simple use case would be a smooth transition from a pan flute to a trumpet sound. Since we really combine spectral models, this is usually better than crossfading the samples, and does not have the undesirable phase cancellation a direct time domain approach has.

Combining the sounds of instruments can be done in different ways, and support for morphing more than two instruments is implemented. The software has been carefully optimized to allow real-time usage. For Linux, the usual plugin formats, LV2 and VST are supported, as well as a standalone JACK client and a plugin for BEAST. The VST plugin is also available for 64-bit Windows. At the time this paper was written, a port for macOS is being developed, but is not yet ready for end users.

Our goal is that musicians should be able to work with whatever tools they usually use, and the real-time morphing should integrate with these tools.

In addition to this paper, [Westerfeld, 2017] (german) provides a much more detailed description of how SpectMorph works.

2 Analysis of the Samples

In [Serra, 1989] and [Serra and Smith, 1990], the authors present a method called spectral modelling synthesis, which is the theoretical foundation of our analysis step. This produces a spectral model of the sound as sum of a deterministic (sine components) and a stochastic (noise) part.

2.1 Splitting the Signal into Frames

The perceived timbre of samples of musical instruments slowly changes over time. Our goal is to model the structure of the spectrum, as a morphable representation of the timbre. To capture the slow gradual change, the first analysis step is to split our input signals into frames of constant length.

Typically we use overlapping frames of 40ms duration, but for low notes the frames will be longer. If we look at a plot of one single pan flute frame as shown in figure 1, we can see that the signal is almost periodic within these 40ms.

The next analysis step is designed to capture this regularity by representing the frame signal as sum of (periodic) sine functions.

2.2 Modelling the Frame as Sum of Sine Waves

Since our signal is almost periodic, it can almost be represented as a sum of a number of sine

[1] `http://www.spectmorph.org`

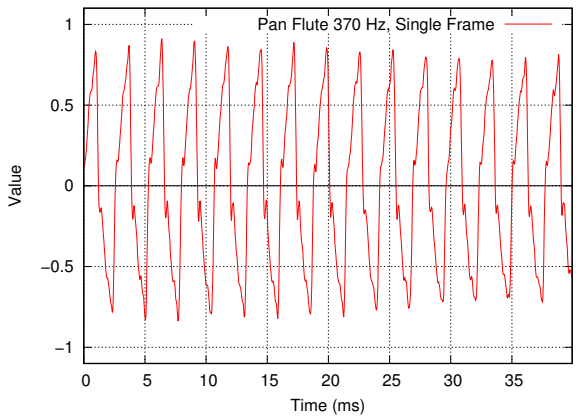

Figure 1: Single analysis frame of the pan flute

waves. In this analysis step, we try to find parameters to decompose the frame signal, called $x(t)$ into a periodic part $d(t)$ and some non-periodic rest $e(t)$.

$$x(t) = d(t)+e(t) = \sum_{p=1}^{P} A_p cos\left(2\pi\frac{F_p}{F_s} + \Phi_p\right)+e(t)$$

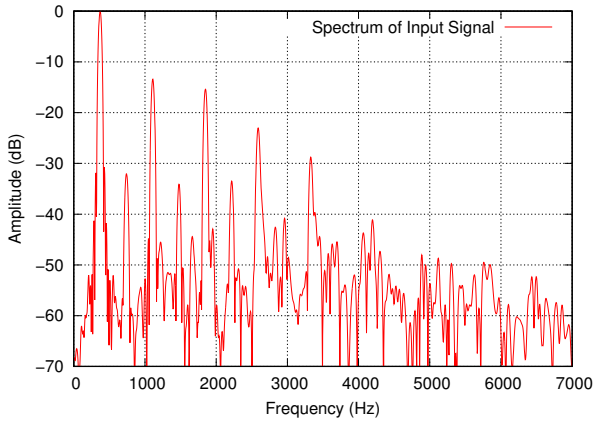

Figure 2: Spectrum of pan flute analysis frame

Figure 2 shows the spectrum of our pan flute analysis frame. Each sine component corresponds to one peak in the spectrum. In other words, if we say that the sound is made up of partials, we want to find the frequency F_p, amplitude A_p and phase Φ_p of each of these partials, and in the next step deal with whatever remains ($e(t)$).

To do this, we compute the (Hann-) windowed fourier transform of each frame signal. We zero-pad the input signal to get a higher frequency resolution, and use a power-of-2 FFT for efficiency reasons. The parameters for frequency, amplitude and phase can be derived by picking the peaks from the spectrum. A peak is a local maximum in the spectrum, however only some peaks are relevant (that is, correspond to a sine component).

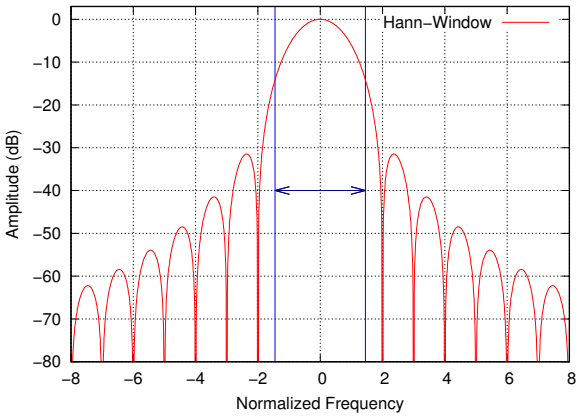

Figure 3: Peak Width and Hann-Window Transform

Since we have multiplied our input data with a hann window before using the FFT, an ideal sine component would look like figure 3 in the spectrum. This would correspond to a peak width of four. In practice a sine component will never be completely ideal, and we also have to consider that multiple sine components added together interfere. Still, we found that checking for a peak width[2] of at least 2.9 provides a good criterion for picking the relevant peaks on the different instrument samples we tested.

As a second (global) criteria we compare the magnitude of the peak with the biggest peak that we found in all frames. If the relative peak magnitude is less than -90 dB, we also consider the peak irrelevant. Note that these two criteria are intentionally chosen too permissive (rather than too strict), because keeping more peaks than necessary will not affect overall sound quality, whereas ignoring too many peaks as irrelevant would.

Finally, each peak corresponds to a sine component with a certain amplitude, frequency and phase, found by interpolation of the three values around the FFT bin with the peak maximum (as for instance described in [Serra, 1989]). At this point, the spectrum of the sum of all sine signals will be very similar to the input spectrum. Figure 4 shows this spectrum. These

[2]The peak width is computed based on how many bins the peak occupies from the local minimum before to the local minimum after the center. This value is normalized relative to frame length, fft size and zeropadding.

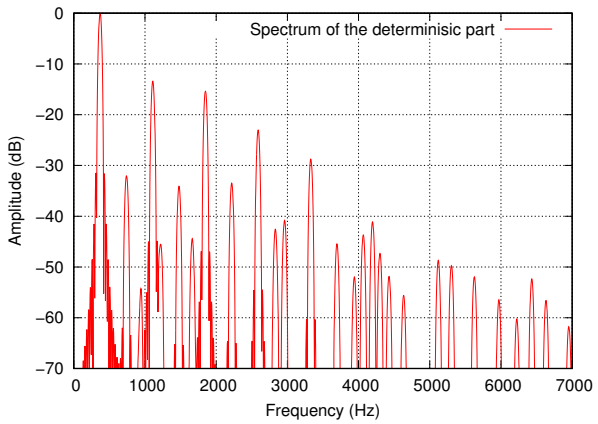

Figure 4: Spectrum of the sum of the sine waves

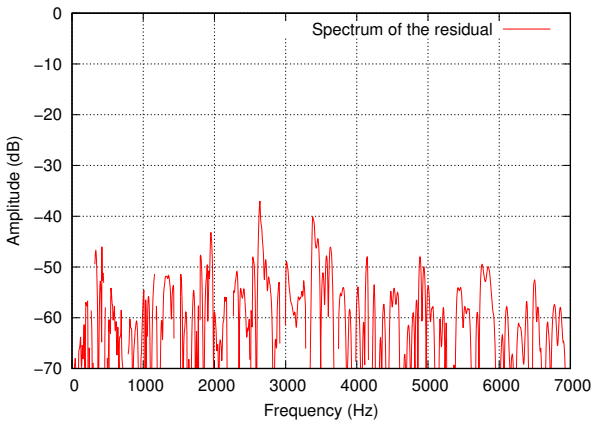

Figure 5: Spectrum of the residual

partials make up most of the sound, and therefore are the most important part of the model of the timbre. So a good answer to the question how a pan flute sounds (at a certain point in time), is: as a sum of a number of sine waves with these frequencies, amplitudes and phases.

2.3 Modelling the Residual

The sine signals usually make up most of the sound, but they cannot represent noisy aspects of the sound. For instance, a flute sample will have some breath or air noise, and a violin has some noise created by the bow. So far, we have only modelled the deterministic part $d(t)$ of the signal. To get the part of the signal that we did not yet describe, we simply subtract the spectrum of the sum of all sine waves from the original spectrum.

If our sine signal $d(t)$ perfectly matches our original signal $x(t)$, nothing would remain. However, if we missed something, the subtracted spectrum will contain just the missing part. For our pan flute frame, the residual spectrum is shown in figure 5.

As we have a model of the periodic part already, we assume that what remains is some kind of noise. So to complete our timbre model, we use 32 perceptually spaced frequency bands and store just the average level of the noise that we find in in each of these bands. The noisy part of each frame is then stored, along with the sine parameters, and provides a spectral model that includes both, $d(t)$ and $e(t)$.

2.4 Issues with Transients

The analysis algorithm described so far produces very good results for many different input signals. However, if the signal contains transients, the quality can be low at the time of the transient. Transients are fast changes in the signal. For instance for a piano attack sound, there is silence, and then suddenly there is a loud signal. This attack happens much faster than the size of one frame. But we only store parameters per frame, so the sharp attack of the original signal gets blurred over one analysis (and later synthesis) frame. The piano resynthesis will have a much softer attack than the original.

So far, we have not found a good strategy for handling transients. As a brief example consider the following method: do analysis as mentioned before, but for frames with transients, keep original sample data. While this is not too complicated to implement, and while this definitely will improve the quality (and preserve the sharp attack of a piano), the problem is that for these frames we would not be able to do proper morphing, as the description of the signal is no longer parametric in a form that allows combining multiple input signals.

Ideally, we would have an analysis strategy that preserves transients, but in a way that still allows morphing. Fortunately, even without special casing transients, there are many instrument sounds that only change slowly so that there are no quality issues caused by transients.

3 Synthesis

The goal of the synthesis is to compute a time domain signal from a sequence of spectral models. These spectral models consist of a set of sine frequencies, amplitudes and phases, and 32 noise bands. Similar to the analysis step, synthesis takes place in synthesis frames, which are overlapping, and added together to produce a

time domain signal.

3.1 Additive Synthesis and Inverse FFT

Since we want to use the synthesis in real-time, performance is important. Although we could theoretically simply add up all sine waves of each frame, to get $d(t)$, this could easily result in 100 or more sine computations and additions per output sample value. Instead, we compute the spectrum of the frame by adding one peak per sine component, and then use an inverse FFT, as described in [Rodet and Depalle, 1992].

The computation of the noise part of the output consists in setting up a spectrum of suitably chosen random values according to the 32 perceptual bands, and performing an inverse FFT. In SpectMorph, the sine and noise part are computed together, so we only need one single inverse FFT per synthesis frame.

3.2 Reconstruction of the Phase

Before, we used frequencies F_p, amplitudes A_p and phases Φ_p to describe the sine components that are part of one analysis frame. A more or less technical detail is that we can (and have to) avoid using the phase Φ_p completely during synthesis. This is also described in [McAulay and Quatieri, 1984] and [Serra, 1989].

To compute a phase value for a sine component that is to be synthesized in the current synthesis frame, we look at the last synthesis frame. If a component with similar frequency[3] can be found in the last synthesis frame, then the phase in this synthesis frame is chosen to continue the sine component of the previous frame. This avoids interference and possible cancellation of sine components of adjacent synthesis frames, while only using F_p and A_p for synthesis.

Any sine component in the current frame that was not found in the last synthesis frame starts with a phase of zero.

4 Morphing

4.1 Input and Output Parameters

Once we have transformed our samples to spectral models during analysis, the input for the morphing algorithm is the description of two spectral models, each with the parameters shown in table 1. The two input frames are from two sources, source A and source B, so we use superscript α and β for the parameters, for instance F_1^α (first frequency of source A) or A_1^β (first amplitude of source B).

	Parameters
Frequencies	$F_1, \ldots, F_P$
Amplitudes	$A_1, \ldots, A_P$
Noisebands	$NOISE_0, \ldots, NOISE_{31}$

Table 1: Spectral Model Parameters for one Frame

From this input, the morphing stage should produce one single spectral model with frequencies, amplitudes and noise band values. A parameter $\lambda \in [0, 1]$ controls the morphing. A value of $\lambda = 0$ means that only source A is audible, $\lambda = 1$ means that only source B is audible, $\lambda = 0.5$ corresponds to a 50%/50% mix, and so forth.

4.2 The Stochastic Part

Since the computation of the stochastic part (noise part of the signal) is simple, we start with this. The 32 noise band parameters of the output can be computed as

$$NOISE_b = (1-\lambda) \cdot NOISE_b^\alpha + \lambda \cdot NOISE_b^\beta, \qquad \text{for } b \in [0, 31]$$

For $\lambda = 0$, only the noise component of source A is used. For $\lambda = 1$, only the noise component of source B is used. If λ is between 0 and 1, the amplitude of the corresponding noise bands is interpolated linearly.

4.3 Matching corresponding Partials

Figure 6 is one example for the positions of the partials of a frame from source A and a frame from source B. To be able to perform the morphing, the first step is to find matching components. If partials match, they are assigned to each other. However each entry is at most used once, no partial is assigned to more than one entry.

To get good results, our algorithm starts with the louder partials. Since they will be clearly audible in the output, it is important that they get a close match.

We also use a frequency similarity criteria. Let G be the fundamental frequency of the note, we ensure that

$$\delta = |F_q^\beta - F_p^\alpha| \leq \frac{G}{2}$$

[3] We consider two frequencies to be similar, if the frequency difference is less than than 5%.

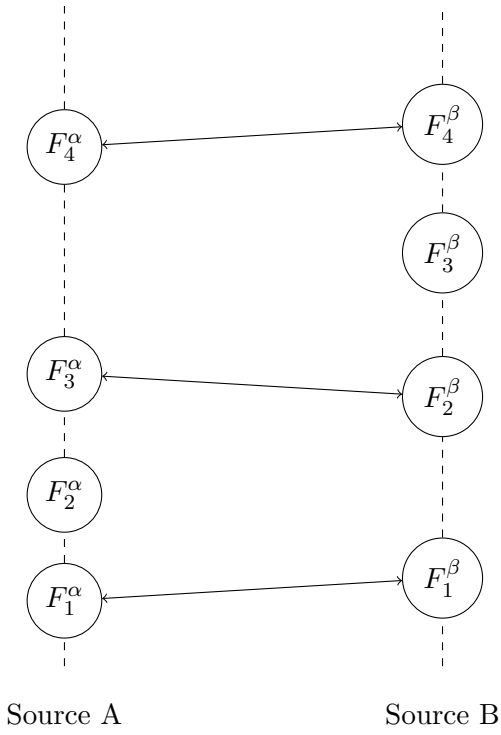

Figure 6: Matching Partials of the two input frames from Source A and Source B

which means that partials can only be assigned if they are closer to each other than half the fundamental frequency.

At the end of this stage, some partials are assigned to each other (like F_1^α and F_1^β), whereas other partials remain without a matching frequency in the other frame (like F_2^α).

4.4 Computing the Amplitudes

Once we have assigned the partials of the frames from source A and source B to each other, the output amplitudes can be found using

$$A = (1 - \lambda) \cdot A_p^\alpha + \lambda \cdot A_q^\beta$$

for partials p and q which have been assigned in the previous step.

For partials that remain without matching entry in the other frame, we simply use zero as amplitude for the interpolation.

We also implement an alternative way of dealing with amplitudes, which should be closer to how human loudness perception works: dB-linear amplitude interpolation. To do this, the amplitudes are converted to dB before the interpolation step, and converted back afterwards.

4.5 Computing the Frequencies

After the previous description of how noise band parameters and amplitudes are computed, the first idea would be to use the same strategy here, so

$$F = (1 - \lambda) \cdot F_p^\alpha + \lambda \cdot F_q^\beta$$

and keep the frequency exactly as it is for partials that have not been assigned.

However, this leads to one undesirable effect: for partials from the analysis stage that are not very loud, it does not matter much if their frequency is wrong. They are inaudible anyway.

If such a partial gets assigned to a very loud partial, the output frequency can easily get wrong, for if for instance $\lambda = 0.5$, half of the frequency output value F is determined by the almost inaudible partial.

So in practice, if partials do not have the same volume, we ensure that the louder partial has more influence on the output frequency.

Let A_p^α be the louder partial ($A_p^\alpha \geq A_q^\beta$), then we use as frequency:

$$F = F_p^\alpha + m\lambda(F_q^\beta - F_p^\alpha)$$

where m is a factor that depends on both amplitudes:

$$m = \frac{A_q^\beta}{A_p^\alpha}$$

If both amplitudes are equal, so that $m = 1$, we get the same result as the approach at the start of the section.

$$F = F_p^\alpha + 1\lambda(F_q^\beta - F_p^\alpha) = (1 - \lambda)F_p^\alpha + \lambda F_q^\beta$$

If one amplitude is a lot louder than the other, we have $m \approx 0$:

$$F \approx F_p^\alpha + 0\lambda(F_q^\beta - F_p^\alpha) = F_p^\alpha$$

So if a stable (louder) partial is combined with an almost inaudible partial, the factor m will ensure that the louder partial almost completely determines the frequency.

4.6 Grid Morphing

For grid morphing, instruments are placed on grid points of an WxH (width W, height H) grid. The simple case is that we have a $2x2$ grid, with four instruments A, B, C and D. In this setup, we now have two control parameters that correspond to the X- and Y-position on the plane.

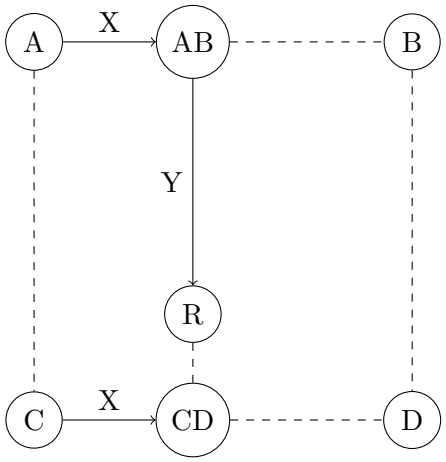

Figure 7: Grid Morphing of four Instruments

Our job is to compute a resulting output sound R as result of setting our X- and Y-position. This is shown in figure 7. To compute R we proceed as follows: as a first step we combine instrument A and B with the control value X, which produces sound AB. Then we combine instrument C and D, again with control value X to get CD.

As last step we can combine AB and CD with the control value Y to R. To summarize this algorithm: we can morph of four instruments on a plane using the morphing of two input frames, which we already described. To do it, we simply use this algorithm three times.

5 Plugin Features

In the last sections we've given the theoretical background how SpectMorph works internally. We'll now summarize some of the relevant topics for end users, which will usually use one of the SpectMorph plugins, LV2, VST or BEAST (or the JACK client).

5.1 Standard Instrument Set

As we've seen, SpectMorph itself is based on sample data. It can morph instrument sounds, but only after an instrument has been described as a set of samples, and analysis of these samples was performed. Typically we need one sample per semi-tone, or at least one sample every few semi-tones, in order to produce good reproduction quality. The tools required to build instruments from samples are distributed as a part of SpectMorph. However, it is a bit of work to create your own instruments.

To address this issue, SpectMorph currently ships with 14 ready-to-use instruments, like trumpet, oboe, pan-flute, saxophone and so on. All of the samples we used were free. Many were taken from the *Iowa Musical Instruments Samples*[4], we also recorded some samples ourselves, and added some instruments from the *Fluid R3 SoundFont*[5].

5.2 Using Morphing

The user interface of SpectMorph supports the two use cases mentioned before. The simple case involves combining two instruments using morphing, in the UI this is called "Linear Morph". For a linear morph, the user can choose the instruments from a list of instruments, usually from the standard instrument set. In the simplest case, an UI slider is used to control the morphing, so dragging the slider will gradually change the sound from the first to the second instrument.

Grid morphing as mentioned previously is also supported, which allows using an X/Y control pad to control the position on the grid with the mouse.

5.3 Automation / Control

If users create music using the plugin in sequencers, it is often desirable to exactly specify how the morphing should be performed, alongside with the notes to be played (rather than controlling the morphing with the mouse like it could be done during live performances). So we support automating the control value, so that the timbre can be controlled by the sequencer. For X/Y morphing, two control values can be used, to automate the position on the plane.

Besides these possibilities, SpectMorph implements an LFO operator (low frequency oscillator), which will change the control value periodically. This feature is also useful for live performances, to get interesting slowly changing sounds.

5.4 After Morphing

So far, we've described how a sound is produced, usually combining two or more standard instruments using some control values. This can be used as it is, or be modified with some optional additional steps before the output sound is generated.

One refinement is the unison effect, which adds up a few detuned copies of the spectral

[4] `http://theremin.music.uiowa.edu/MIS.html` - public domain license

[5] packaged by many linux distributions, for instance ubuntu: `https://launchpad.net/ubuntu/xenial/+package/fluid-soundfont-gm` - MIT license

model of the sound. This makes the sound more fat, and can be seen as imitation of multiple musicians playing the same notes using the same instrument.

Another refinement is using an ADSR envelope, which adds a custom volume envelope, replacing the natural volume envelope the sound has. Finally we implemented support for portamento and vibrato.

Although these optional post-morphing operations, that modify the output sound, can produce interesting possibilities, it is also obvious that using such post-morphing refinements has a cost: the sound will no longer be as natural as possible. For instance, giving a trumpet a quick exponential volume decay is a new variant of the sound, but it will sound less like a trumpet.

6 Conclusions

The algorithms presented in this paper, implemented in SpectMorph, produce realistic reproduction of instruments, based on building spectral models of samples. For many musical instruments, such as bassoon, trumpet, saxophone, oboe and so forth, the quality of the analysis step will be very high.

There are however some cases, in which the spectral modelling approach does not work well. Whenever the peaks in the spectrum are too close, the peak finding algorithm will not work properly. While spectral models for natural sounds usually provide good quality, typical synthetic sounds, such as a synthetic saw waves with unison cannot be analyzed properly.

We already discussed that there are issues with transients, such as the sharp attack of a piano, in section 2.4. For all sounds where the analysis step provides good quality, the morphing steps described in section 4 provide realistic transitions between the timbre of the instruments. This provides composers with a way of creating sounds that is not available with samplers or synthesizers.

In SpectMorph, much care has been taken to ensure not only good quality of the sounds, but also fast computation. Morphing and synthesis are reasonably fast so that high polyphony is available during real-time usage.

The SpectMorph LV2/VST/BEAST plugin (and the JACK client) supports creating new sounds by morphing existing ones, and includes many standard instruments. The plugin integrates into whatever sequencer or live performance environment the composer wants to use, and provides flexible ways of controlling the morphing parameters, as well as post-morphing refinements.

References

Robert J. McAulay and Thomas F. Quatieri. 1984. Magnitude-only reconstruction using a sinusoidal speech model. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 27.6.1–27.6.4.

X. Rodet and P. Depalle. 1992. Spectral envelopes and inverse FFT synthesis. In *Audio Engineering Society Convention 93*.

Xavier Serra and Julius O. Smith. 1990. Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal*, 14(4):12–24.

Xavier Serra. 1989. A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition. Dissertation STAN-M-58, Center for Computer Research in Music and Acoustics, Stanford University.

Stefan Westerfeld. 2017. Morphing der Klangfarbe von Musikinstrumenten durch Spektralmodellierung. `http://edoc.sub.uni-hamburg.de/informatik/volltexte/2018/236/`.