

Stationary and non-stationary sinusoidal model synthesis with phase vocoder and FFT⁻¹

Clément Cazorla, Vincent Chrun, Bastien Fundaro, Clément Maliet
ENSEEIHT - 3EN TSI 2016-2017

March 7, 2017

Abstract

The present document is to serve as both a technical report and a documentation for the code produced during the long project. As such the first part will explain the theoretical framework and the state-of-the-art of the field. The second part will give some insight about the code structure and the conventions that were adopted and last but not least, the third part will serve as an actual documentation and details every class, methods and attributes.



Contents

I	Introduction	4
1	The company	4
2	Objective	4
3	Context of the project	5
4	Work environment and project management	5
4.1	Work environment	5
4.2	Project management (Gantt chart)	6
II	General analysis/synthesis approach	7
5	Additive Synthesis (Time Domain)	7
6	General framework	8
6.1	Phase Coherence	10
III	Sinusoidal model synthesis in the frequency domain	11
7	Stationary Case	11
8	Quasi-Stationary Case	12
8.1	Theory	12
8.2	phase vocoder	12
9	Non-Stationary Case	12
9.1	Theory	12
9.2	phase vocoder	12
IV	Result	13
10	Stationary Case	13
11	Non-Stationary Case	13
V	Code structure and conventions	14

12 Conventions	14
12.1 Naming conventions	14
12.2 Spectrum and sinusoids parameters classes	15
12.2.1 Spectrum	15
12.2.2 Parameters	16
12.2.3 NonStationaryParameters	16
13 Code structure	17
13.1 General structure	17
13.2 Class structure	17
 VI Documentation	 18
14 Core module	18
14.1 Synthesizer	18
14.2 StationarySynthesizer	19
14.3 NonStationarySynthesizer	19
15 Spectrum generation module	20
15.1 SpectrumGenerator	20
15.2 StationarySpectrumGenerator	20
15.3 NonStationarySpectrumGenerator	21
15.4 LobeGenerator	21
15.4.1 StationnaryLobeGenerator	21
15.4.2 NonStationaryLobeGenerator	22
16 Phase Vocoder module	22
16.1 PhaseVocoder	22
16.2 StationaryPhaseVocoder	23
16.3 NonStationaryPhaseVocoder	23
A Phase advance and propagation along the signal	25
A.1 On the first attempt at Phase Vocoder use	25
A.1.1 "Pure" synthesis	25
A.1.2 "Parametered" synthesis	26
A.2 Theoretical phases advance	27
A.2.1 Stationary case	27
A.2.2 Non-stationary case	27

Part I

Introduction

1 The company

For this project we work with Audiogaming. Audiogaming is a start-up company which creates audio plug-in for movies and video games. They are base in Toulouse and Paris.

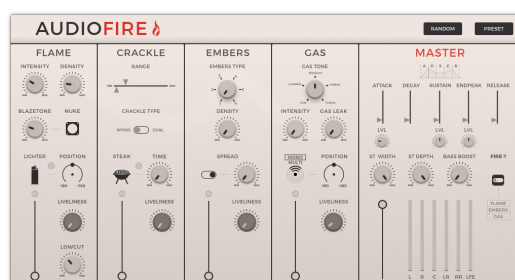


Figure 1: Audiofire: audio plug-in that recreates fire sound

2 Objective

The objective of the project is to use inverse Fourier transform to synthesize a sound. There are two parts in this process: the Analysis and the Synthesis.

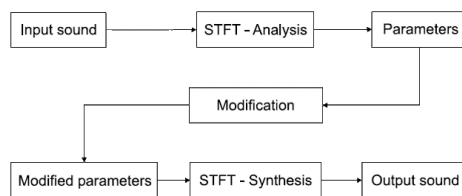


Figure 2: General approach for modifying a sound in the spectral domain

Our objectives were to focus on the synthesis. We supposed that the analysis part is done and that we know all the parameters. The work is divided into two parts : the stationary synthesis and the non-stationary synthesis.

3 Context of the project

We had 6 weeks to do this work. As a beginning, we had the work of the previous group which worked on this project in 2015. They had done an analysis estimator of sinus parameters and sinus generation with those parameters (only stationary) in Python and some research on the Non-stationary synthesis with the LUT of lobes in Matlab. We imposed to ourself an Object Oriented Programming tree structure in Python, in that way the code will be easier to use by the client and the next group which will work on this project. This structure will be described in details in this report.

4 Work environment and project management

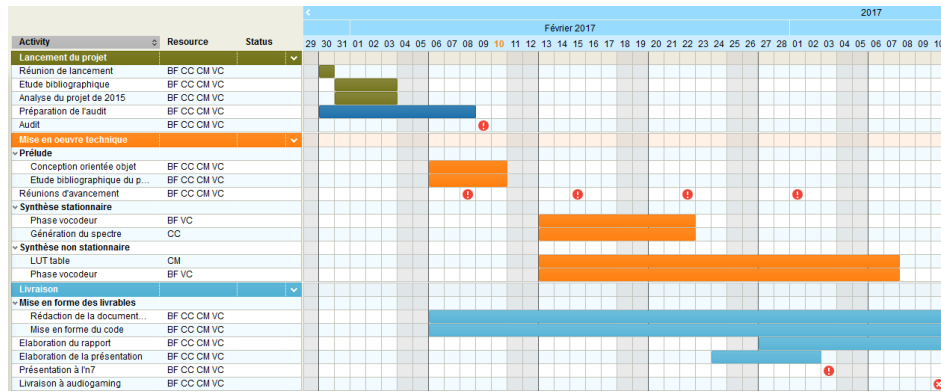
4.1 Work environment

We used PyCharm to program in Python, Slack to communicate with our supervisor (ask questions, send some quick results, plan a meeting...). An other important tool was GitHub, we used it to share our codes and to work all at the same time in the same project.



Figure 3: *PyCharm* as Python IDE , *Slack* to communicate, *GitHub* to stock the codes and have a versionning, *Freedcamp* to plan the project events

4.2 Project management (Gantt chart)



Part II

General analysis/synthesis approach

5 Additive Synthesis (Time Domain)

In signal processing, about as much work has been done in analysing sounds as in synthesizing them. The most common approach is to assimilate the sound to a sum of time varying sinusoids, which are called partials [1]:

$$x(t) = \sum_{n=1}^N a_n \sin(2\pi f_n t + \phi_n) \quad (1)$$

This is motivated by the fact that a great amount of research has been done towards analysing such sound models [2, 3]. The most common and obvious approach consists in adding a number of independent oscillator to reproduce the sound, each one of them being controlled independently.

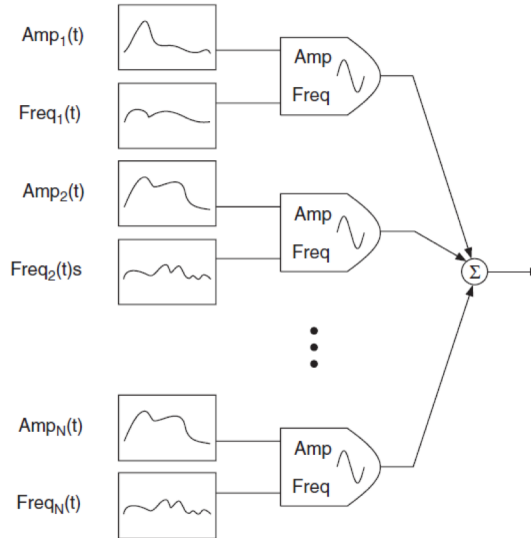


Figure 4: *The additive synthesis*

The problem of this method is that it is very costly to implement, so it is impossible to use in real-time applications. A solution would be to generate the sines in frequency domain, using the parameters given by the analysis.

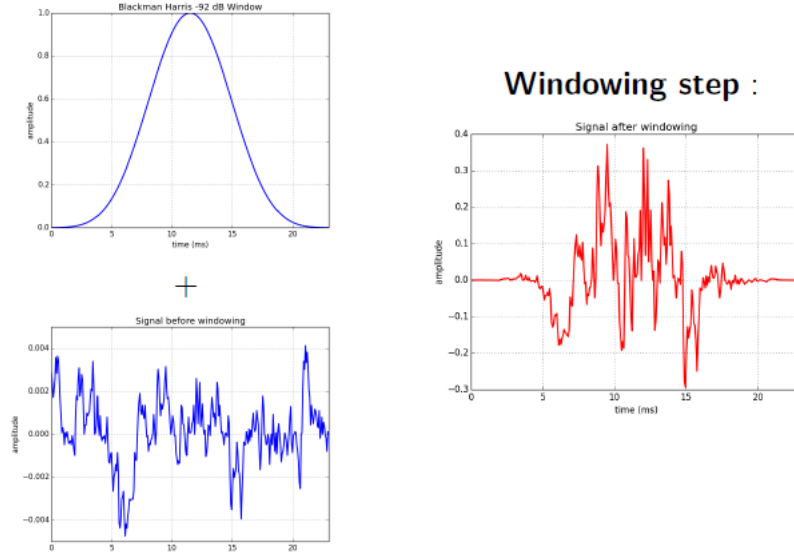


Figure 5: *Windowing step*

6 General framework

The first step of the method is the analysis. The aim is to extract the parameters of the signal (magnitude, phase, frequency). First of all, we window the signal in order to maximize the energy in the main lobe, using a 92 dB Blackman-Harris window for instance:

Then, we apply a STFT (Short Time Fourier Transform) to extract the coefficients. We assume each peak represents a sinusoid, and we use a particular case of STFT called STPT, that detects peaks in the signal and that applies the STFT in the areas that are close to the peaks and sets the rest of the signal to zero. It is a simplest version of STFT, because we neglect the residual part of the signal by only considering the periodic part of the signal. This is less costly and works quite well, except in case of transitional signal because it is very difficult to model with a finite sum of sines.

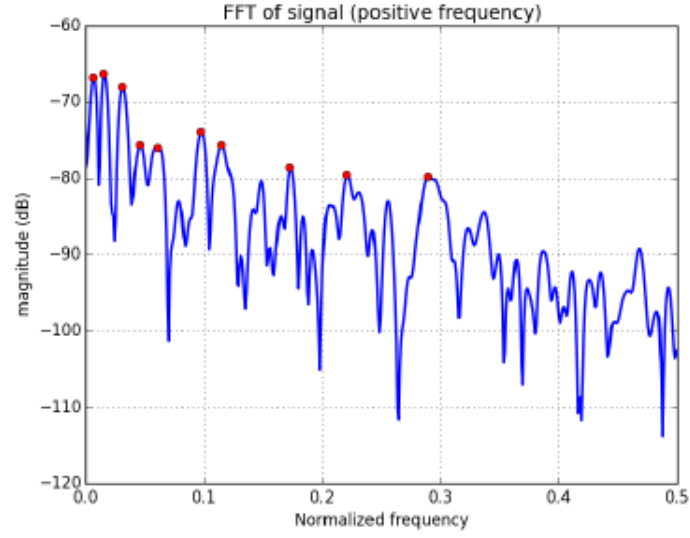


Figure 6: *Peaks detection*

Finally, we synthesize the new signal by FFT^{-1} and we compare it to the original signal we applied the analysis to.

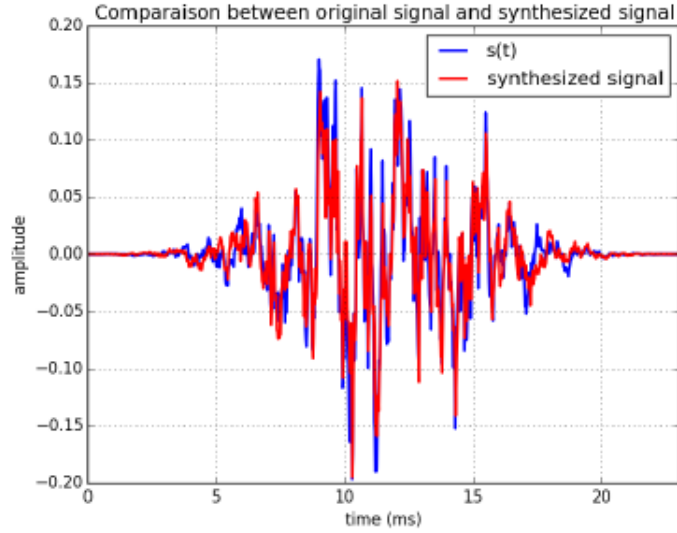


Figure 7: *Synthesized frame vs Original frame*

6.1 Phase Coherence

The phase becomes a problem when we are synthesizing a signal, even a simple sinusoid, because for each frame we create a "different" sinusoid with the same parameters. That is not what we want, we want to create a single sinusoid :

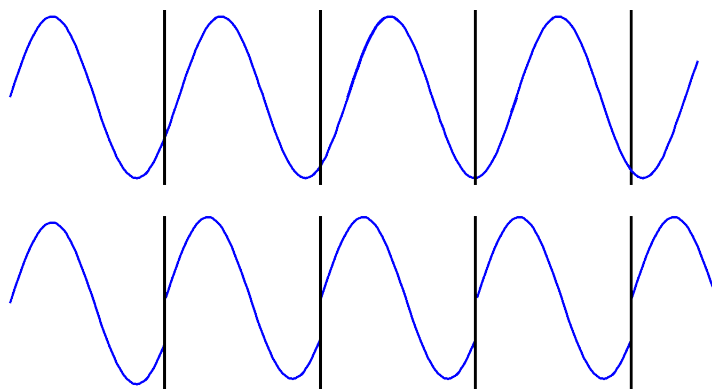


Figure 8: Coherent phase (above) vs. Incoherent phase (below) between frames

When we listen to a signal which phase is incoherent between frames (vertical jump) it will sound as "pops". This is something very unpleasant to listen to. We need to do something to make sure there is no discontinuity. Thus, correcting the phase is absolutely necessary.

In the stationary case, it is something easy to do. Knowing the exact frequency and for each frame, we advance the phase of $2\pi\tilde{f}R_a$ ¹.

¹To see details of the theoretical phase advance for the stationary case, please refer to the appendix A.2 page 27.

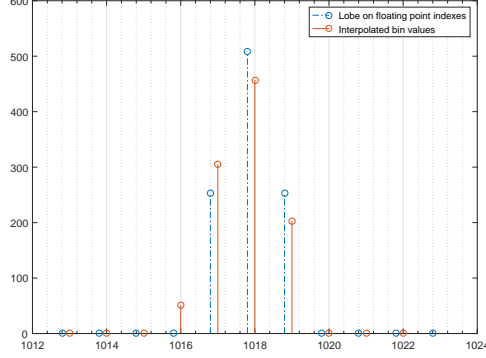


Figure 9: Interpolation of stored lobe values onto the FFT bins.

Part III

Sinusoidal model synthesis in the frequency domain

7 Stationary Case

Most stationary signals can be decomposed in a sum of partials with seldom loss in perception :

$$s(n) = \sum_{i=1}^K \alpha_i \sin(2\pi \tilde{f}_i n + \phi_i) \quad (2)$$

If all three parameters α_i , \tilde{f}_i and ϕ_i are known for each partial, perfect reconstruction is thus possible.

It is first needed to compute the Fourier transform of a given frame. Let $w_{t_a}(n)$ be an analysis window starting at $t_a = kR_a$ where R_a is called the *hop size*. The resulting spectral frame is:

$$S_{t_a}(m) = \text{DFT}[s](m) * W_{t_a}(m) \quad (3)$$

Where $W_{t_a}(m)$ is the DFT of the window w_{t_a} . Under the right assumptions on the window phase², the spectrum consists in main and secondary lobes (which originates directly from $W_{t_a}(m)$) convolved at the sinus amplitude, phase and frequency.

However since most of the energy is concentrated in the main lobe³, one can omit the side lobes with little to no error, ever more so since most of the error is

²Which involves circular shift to cancel the phase shift [4].

³At the condition that w_{t_a} is not rectangular.

made on the edges of the temporal frame where the amplitude is small. In the stationary case, every main lobes are identical in shape, this means that one can synthesise a spectrum by copying a precomputed lobes onto the relevant bins of the FFT. This makes the process computationally light since only 9 points of the main lobes are sufficient [2], although our method used 11 for further precision.

Interpolation is necessary to fill the bins because the sinusoid frequency bin \hat{f}_i is unlikely to fall on an integer bin number ($\hat{f}_i = \tilde{f}_i \times N$ with N the fft size) as illustrated in figure 9, this step allows for a good precision in frequency without the need for zero-padding the spectrum, thus increasing performance.

The phase is set at a common value for every bins of the lobe, on the rational that they all represents the contribution of the same sinusoid. Note that the phase used is the original phase ϕ_i corrected from the windowing effects and advanced by the time passed, to ensure phase coherence as explained in section 6.1, that is to say that:

$$\phi_{lobe} = \hat{\phi}_i + k \times 2\pi\tilde{f}_i R_a \quad (4)$$

Where $\hat{\phi}_i$ denote the window-corrected phase, $2\pi\tilde{f}_i R_a$ the phase advance for a single hop at frequency \tilde{f}_i and k s.t $t_a = k \times R_a$.

8 Quasi-Stationary Case

8.1 Theory

8.2 phase vocoder

9 Non-Stationary Case

9.1 Theory

9.2 phase vocoder

Part IV

Result

10 Stationary Case

11 Non-Stationary Case

Part V

Code structure and conventions

12 Conventions

In this section we remind the reader of a few coding convention necessary to ensure a seamless work flow and a bug free program as much as possible. Files should contains an entire module (as described in 13.1) not just a single class to limit the number of files and ease the bug tracking. Imports in files should be kept to a minimum and left in namespaces (do not use the **from** module **import** * syntax). It is preferable to import a whole module if more than three elements from the module are needed in the file, otherwise consider the **from** module **import** element1 , element2 syntax to avoid unnecessary memory flooding. If conflicts exists, notwithstanding the number of elements needed, the whole modules are to be loaded with a namespace. Namespaces may be abbreviated to the programmer's convenience however some abbreviation are to be universally respected :

- (i) `numpy` should always be imported as `np`
- (ii) `matplotlib.pyplot` should always be imported as `plt`

Finally math functions should always come from the `numpy` module and not python's `math` module to guarantee a universal behaviour across the program.

12.1 Naming conventions

Naming conventions are freely adapted from Python recommended conventions defined in PEP8 [5], as such :

- (i) *Class* should be named in **CapitalizedWord**
- (ii) *Methods* and *functions* should be named in **lower_case_with_underscores**
- (iii) *Attributes* and *variables* should be names in **lower_case_with_underscores**
- (iv) *Instantiation* following the fact that everything is an object in python should be named as *variables*.

Moreover during class declaration, the following principles should be adopted :

- (i) Non-public methods and attributes should use one leading underscore.
- (ii) Elements that conflicts with python reserved name should use one trailing underscore rather that simplification or a misspelling.
- (iii) Accessors or mutators using one leading underscore should be interpreted as properties of their associated attribute. As such it should be guaranteed that they induce a low computational cost.

- (iv) Non-public elements that should not be inherited or may cause conflicts during inheritance should use two leading underscore and make use of Python name-mangling.

To seamlessly manipulate both *stationary* and *non stationary* models, class that are inherited in two versions are preceded with either **Stationary** are **NonStationary** respectively.

12.2 Spectrum and sinusoids parameters classes

Because many spectra, main lobes and sinusoidal model parameters have to be traded between modules we created two classes, respectively **Spectrum** and **Parameters**. They mainly serve as containers, holding the data and returning them in a point wise fashion. This way we can prevent conflicts and errors that would come from a non-uniform data sharing protocol and as well ensure that every operation performed on either spectra or parameters are made following the same principles and algorithms.

12.2.1 Spectrum

Spectrum
<pre> _amplitude : np.array _phase : np.array _nfft : int ----- __init__(self, amplitude, phase) __add__(self, other) __iadd__(self, other) __mul__(self, other) __imul__(self, other) from_complex_spectrum(cls, complex_spectrum) @classmethod void_spectrum(cls) @classmethod set_spectrum(self, amplitude, phase, start_bin=None, stop_bin=None) set_complex_spectrum(self, complex_spectrum, start_bin=None, stop_bin=None) get_amplitude(self, k) get_phase(self, k) get_nfft(self) </pre>

The Spectrum class stores a spectrum in amplitude and phase, however it may be created or changed from a complex **np.array** respectively with the class method **from_complex_spectrum** and the method **set_complex_spectrum**. Those two methods may take optional parameters **start_bin** and **stop_bin** if one need to update only a part of the spectrum, for example a single lobe. The class checks that the given data are consistent upon instantiation. The **+** operation as well as the **+** = operation have been defined between two **Spectrum** objects and between a **Spectrum** object and an array of complex numbers.

The \times operation as well as the $\times =$ operation have been defined between a **Spectrum** object and an array of complex numbers. Addition and multiplication attempts between other data type will result in a **NotImplementedError** exception.

12.2.2 Parameters

Parameters
<pre> _amplitudes : np.array _frequencies : np.array _phases : np.array _number_sinuses : int ----- __init__(self, amplitudes, frequencies, phases) get_amplitude(self, k) get_frequency(self, k) get_phase(self, k) get_number_sinuses(self) </pre>

The **Parameters** class is more of a structure than a class and only contains the stationary sinusoidal model parameters and their respective accessors. It also stores the number of sinuses and checks that the given data are consistent upon instantiation.

In the stationary sinusoidal model the signal $s(t)$ is defined as follow¹ :

$$s(n) = \sum_{i=1}^{N_{sinus}} \alpha_i \sin(2\pi \tilde{f}_i n + \phi_i)$$

with $\tilde{f}_i = \frac{f_i}{f_s}$ the normalised frequency.

We then store the parameters as follows :

_amplitudes stores the α_i

_frequencies stores the \tilde{f}_i

_phases stores the ϕ_i

12.2.3 NonStationaryParameters

NonStationaryParameters(Parameters)
<pre> _acrs : np.array _fcrs : np.array ----- __init__(self, amplitudes, frequencies, phases, acrs, fcrs) get_acr(self, k) get_fcr(self, k) </pre>

The stationary sinusoidal model can be extended to the first order development

to better model fast amplitude and frequency change over time. The signal $s(t)$ can then be expressed as a sum of linearly varying chirps¹ :

$$s(n) = \sum_{i=1}^{N_{sinus}} (\alpha_i + \mu_i \cdot nT_s) \sin(2\pi \tilde{f}_i n + \frac{\psi_i}{2} (nT_s)^2 + \phi_i)$$

Where we define the **Amplitude Change Rate** μ and the **Frequency Change Rate** ψ .

Thus we inherit the **Parameters** class to add the two additional parameters as follow :

`_acrs` stores the μ_i

`_fcrs` stores the ψ_i

13 Code structure

13.1 General structure

13.2 Class structure

¹Please look up section 9 page 12 for more details

Part VI

Documentation

14 Core module

14.1 Synthesizer

Synthesizer
<pre>_window_size : float _window_type : float _nfft : float _analysis_hop : float _synthesis_hop : float _current_parameters : Parameters _past_parameters : Parameters _fs : float _past_spectrum : Spectrum _current_spectrum : Spectrum __init__(self, window_size, window_type, zero_padding_factor, analysis_hop, synthesis_hop, current_parameters, fs=None) __del__(cls) reset_synthesizer (self, window_size, window_type, zero_padding_factor, analysis_hop, synthesis_hop, current_parameters, fs) get_next_frame(self) set_next_frame(self, next_parameters) _set_window_size(self, window_size) _set_window_type(self, window_type) _set_analysis_hop(self, analysis_hop) _set_synthesis_hop(self, synthesis_hop) _set_current_parameters(self, current_parameters) _get_current_spectrum(self) _set_current_spectrum(self, new_spectrum)</pre>

14.2 StationarySynthesizer

StationarySynthesizer
<pre>_window_size : float _window_type : float _nfft : float _analysis_hop : float _synthesis_hop : float _current_parameters : Parameters _past_parameters : Parameters _fs : float _past_spectrum : Spectrum _current_spectrum : Spectrum _spectrum_generator : StationarySpectrumGenerator ----- __init__(self, window_size, window_type, zero_padding_factor, analysis_hop, synthesis_hop, current_parameters, fs=None) get_next_frame(self) inverse_fft(self, current_spectrum)</pre>

14.3 NonStationarySynthesizer

NonStationarySynthesizer
<pre>_window_size : float _window_type : float _nfft : float _analysis_hop : float _synthesis_hop : float _current_parameters : NonStationaryParameters _past_parameters : NonStationaryParameters _fs : float _past_spectrum : Spectrum _current_spectrum : Spectrum _spectrum_generator : StationarySpectrumGenerator ----- __init__(self, window_size, window_type, zero_padding_factor, analysis_hop, synthesis_hop, current_parameters, fs=None) get_next_frame(self) inverse_fft(self, current_spectrum)</pre>

15 Spectrum generation module

15.1 SpectrumGenerator

SpectrumGenerator
<pre><code>_parameters : Parameters _nfft : np.float _spectrum : Spectrum _window_size : np.float _analysis_hop : np.float ----- __init__(self, window_size, parameters, nfft, analysis_hop) _add_lobe(self, k, lobe) _set_window_size(self, window_size) _set_window_type(self, window_type) _get_parameters(self, new_parameters) _get_spectrum(self)</code></pre>

The aim of this class is to generate a synthetic spectrum from known parameters (amplitudes, phases, frequencies). This class is divided into two subclasses that generate stationary and non-stationary spectrums.

15.2 StationarySpectrumGenerator

StationarySpectrumGenerator (SpectrumGenerator)
<pre><code>_parameters : Parameters _nfft : np.float _spectrum : Spectrum _window_size : np.float _analysis_hop : np.float _lobe_generator : StationaryLobeGenerator ----- __init__(self, window_size, parameters, nfft, analysis_hop) _add_lobe(self, k, lobe)</code></pre>

15.3 NonStationarySpectrumGenerator

NonStationarySpectrumGenerator (SpectrumGenerator)
<pre>_parameters : Parameters _nfft : np.float _spectrum : Spectrum _window_size : np.float _analysis_hop : np.float _lobe_generator : NonStationaryLobeGenerator _regular_lut : np.array __init__(self, window_size, parameters, nfft, analysis_hop) _add_lobe(self, k)</pre>

This class has not been made yet, but the goal is the same as for the StationarySpectrumGenerator.

15.4 LobeGenerator

LobeGenerator
<pre>_window_type : np.float _window_size : np.float _nfft : np.float _window : np.array _lobe : Spectrum __init__(self, window_type, window_size, nfft) _set_window_size(self, window_size) _set_window_type(self, window_type) _gen_lobe(self) _get_lobe</pre>

This class generates a 11 points main lobe.

15.4.1 StationnaryLobeGenerator

StationaryLobeGenerator(LobeGenerator)
<pre>_window_type : np.float _window_size : np.float _nfft : np.float _window : np.array _lobe : Spectrum _gen_lobe : Spectrum __init__(self, window_type, window_size, nfft) _gen_lobe(self)</pre>

15.4.2 NonStationaryLobeGenerator

NonStationaryLobeGenerator(LobeGenerator)
<pre>_abscisse : np.array _ordonnee : np.array _interpolated_lobe : np.array _regular_grid : np.array _domain : np.array _number_acr : np.float _number_fcr : np.float _number_points : np.float _LUT : np.array _gen_lobe : Spectrum ----- __init__(self, regular_grid, acr_domain, fcr_domain, number_acr, number_fcr, window_type, window_size, nfft, fs=None, method_a=None, method_p=None, method_f=None) _gen_uniform_lut(self) _gen_non_uniform_lut(self) _gen_lobes_legacy(self, i, j, acr, fcr, t, n) _gen_lobe(self) _get_lobe(self) _interpolate_lobe(self, acr, fcr, method_a=None, method_p=None, method_f = None))</pre>

The aim is to generate a LUT with a uniform or a non-uniform grid, and to generate a lobe for a given ACR/FCR couple by interpolating with existing lobes of the LUT. The user must chose the kind of interpolation he or she wants to use for magnitude, phase and frequency. Besides, he can choose the size of the LUT to build by giving the number of ACRs and FCRs.

16 Phase Vocoder module

16.1 PhaseVocoder

PhaseVocoder
<pre>_analysis_hop : int _synthesis_hop : int _omega : np.array _past_analysis_spectrum : Spectrum _past_synthesis_spectrum : Spectrum _current_analysis_spectrum : Spectrum _current_synthesis_spectrum : Spectrum ----- __init__(self, analysis_hop, synthesis_hop, current_synthesis_spectrum) _get_region(self, k) _get_pv_spectrum(self, k)</pre>

This module is not used !

The PhaseVocoder file gathers both the Stationary Phase Vocoder and the Non-Stationary Phase.

16.2 StationaryPhaseVocoder

StationaryPhaseVocoder

get_pv_spectrum(self)

The Phase Vocoder algorithm is located here. It will correct the phase at each frequency bin k (remember that this module is not used at the end). To do so it proceed in 5 steps :

- Get the phase difference,
- Remove the expected phase difference,
- Map the phase shift to $[-\pi, \pi]$ range,
- Get the true frequency,
- And finally get the final phase.

For the details, please refer to [6–8] and annex .

16.3 NonStationaryPhaseVocoder

NonStationaryPhaseVocoder

get_pv_spectrum(self)

This module is not complete ! The detailed algorithm called Scaled-Phase Locking is also described in the paper [6]. The idea is to correct the phase on the peaks bin and then apply this correction on each bin of the region containing the peak. To do so, a peak trajectory detection is needed to know between two frames if the peak is the same one on the other frame, because it could have changed his frequency bins due to the non-stationary case. We also need to get the region (bins near the peak bin). And then we apply on this peak the phase vocoder and the corrected phase shift is applied on all bins of the region corresponding to this peak. A Matlab implementation can be found in [9].

Conclusion

A Phase advance and propagation along the signal

A.1 On the first attempt at Phase Vocoder use

In the stationary case, the Phase Vocoder is expressed in 3 equations as follow :

$$\Delta\Phi_k^u = \angle X(t_a^u, \Omega_k) - \angle X(t_a^{u-1}, \Omega_k) - R_a\Omega_k \quad (5)$$

$$\hat{\omega}_k(t_a^u) = \Omega_k + \frac{\Delta_p\Phi_k^u}{R_a} \quad (6)$$

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^{u-1}, \Omega_k) + R_s\hat{\omega}_k(t_a^u) \quad (7)$$

What (5) means is that we look for the true phase shift (the analysis phase shift) during the frame u and $u-1$ which is $\angle X(t_a^u, \Omega_k) - \angle X(t_a^{u-1}, \Omega_k)$ and we compute the error in phase shift, that is to say the difference between the *true* phase shift and the *expected* phase shift.

In (6) we use the error in phase shift to compute the deviation in frequency $\frac{\Delta_p\Phi_k^u}{R_a}$ from the expected frequency Ω_k and thus compute the *true* frequency at which the bin was excited between t_a^{u-1} and t_a^u .

Finally (7) assume the correct synthesis phase shift will be $R_s\hat{\omega}_k(t_a^u)$ that is the true frequency times the synthesis hop.

The issue with such an approach in our case is that we dropped the analysis. We want to use the Phase Vocoder to *create* phase shifts in our spectra, but the Phase Vocoder is in fact nothing but a fancy way to copy existing phase shifts while taking into account a different hop during analysis and synthesis. I will try and break down the issues I have into two cases.

A.1.1 "Pure" synthesis

We wish to synthesize a stationary or a sum of stationary sinusoid from scratch. For simplicity's sake and without loss of generality we will take the one sinusoid case. That is to say that we want, without prior knowledge to generate $s(n)$ such as :

$$s(n) = \alpha \cos(2\pi\tilde{f}n + \phi) \quad (8)$$

knowing only α , \tilde{f} and ϕ .

We will also, to ease the process, assume that we know the application⁴ $f_{w,\tilde{f}}(\phi) : \phi \mapsto \tilde{\phi}$ which takes into account the effect of windowing on the phase of the frame spectrum's phase⁵.

⁴And this this is a very strong hypothesis in the sense that it will never be true, but this is not the core issue at stake here.

⁵For more details on the theory, please read part II page 7

The first step is then to generate a synthetic spectrum with the desired parameters. To do this we only generate a main lobe derived from the Fourier transform of the normalized window w supposedly⁶ used during analysis, and place it at the right position on the spectrum. This involves to interpolate the relevant bins value if by any chance the wanted frequency \tilde{f} is not exactly on a bin, that is to say if $\tilde{f} \notin \{\frac{2k\pi}{N}\}_{k=0\dots N-1}$. We then multiply the generated lobe by $\frac{A}{2}$ and set the lobe phase to $\tilde{\phi} + 2\pi\tilde{f}R_a$ ⁷. We then wished to use the phase vocoder to advance the phase (compute the needed phase shift). To get the temporal frame, we theoretically only have to compute the inverse Fourier transform of the generated spectrum.

In order to use the Phase Vocoder we assumed the generated spectrum to be equivalent to the analysis spectrum $X(t_a^u)$ and the antecedently phase corrected spectrum to be equivalent to the past synthesis spectrum $Y(t_a^{u-1})$.

At the first iteration :

- $X(t_a^{u-1})$ is void because by hypothesis, nothing happened before.
- $X(t_a^u)$ is the freshly generated spectrum
- $Y(t_a^{u-1})$ is also void for the same reasons

Equation 5 gives, for $k \in 1\dots N-1$ s.t $X(t_a^u, \Omega_k)$ is a bin of the lobe a phase shift error of $\tilde{\phi}$.

Then after 6 and 7 we obtain :

$$\begin{aligned}\angle Y(t_s^u, \Omega_k) &= \angle Y(t_s^{u-1}, \Omega_k) + R_s \hat{\omega}_k(t_a^u) \\ &= \angle Y(t_s^{u-1}, \Omega_k) + \frac{R_s}{R_a} \tilde{\phi} + R_s \Omega_k\end{aligned}$$

If $R_s = R_a$ we have a perfect reconstruction of the time synthesized overlap-add test signal. However, in that case, the Phase Vocoder is perfectly irrelevant to the synthesis, indeed, since we have no *actual* analysis phase, we only need to modify R_a to change the length of the final signal.

At the following iteration, we have to update the phase of the generated spectrum to $\tilde{\phi} + 2 \times 2\pi\tilde{f}R_a$ instead of $\tilde{\phi} + 2\pi\tilde{f}R_a$ (as computed in (11)), recursively, we can define the phase of the lobe the i th generated spectrum as :

$$\begin{cases} \tilde{\phi}_i = \tilde{\phi}_{i-1} + 2\pi\tilde{f}R_a \\ \tilde{\phi}_0 = \tilde{\phi} \end{cases} \quad (9)$$

A.1.2 "Parametered" synthesis

In this case, we will not synthesize a truly stationary signal but we assume that the signal is quasi-stationary, which is to say that given a small enough analysis

⁶Because no actual analysis happened

⁷This is because we wish to generate frame spaced by R_a so we have to compensate the *expected* phase shift by hand. In fact, in the purely stationary case, the expected phase shift is the theoretical phase shift.

window, it can be considered stationary within that frame.

$$s(n) \cdot \mathbb{1}_{[t_a^{u-1}, t_a^u]} \simeq \alpha \cos(2\pi \tilde{f}n + \phi) \quad (10)$$

Note that the initial phase ϕ is constant from one frame to the other *by definition*, indeed, a change of phase is equivalent to a sweep in frequency. Also this is more of a constrain of stability on frequency than it is on amplitude because of the nature of the overlap-add process.

We can assume that under the right conditions the method developed in subsection A.1.1 above still holds, if we update the parameters \tilde{f} and α in the spectrum generation.

A.2 Theoretical phases advance

A.2.1 Stationary case

For stationary signals expressed as in (8) the phase advance from one frame to the other is elementary to compute:

$$\begin{aligned} \Delta \Phi_f^{t_a} &= \Phi_f^{t_a + H_a} - \Phi_f^{t_a} \\ &= 2\pi f(t_a + H_a) + \phi - 2\pi f t_a - \phi \\ &= 2\pi f H_a = 2\pi \tilde{f} R_a \end{aligned} \quad (11)$$

Where H_a is the hop-size in seconds that is $H_a = \frac{R_a}{f_s}$.

A.2.2 Non-stationary case

We can not say much about the phase shift of non-stationary signals without making a few assumptions about the frequency modulation.

We assume that given a small enough window the non-stationary signal can be expressed as the following Taylor expansion:

$$s(t) \mathbb{1}_{[t_a^{u-1}, t_a^u]} \simeq (\alpha + \mu t) \sin(2\pi f t + \frac{\psi}{2} t^2 + \phi_a) \quad (12)$$

This is equivalent to assume that the signal is a linear chirp in the window, with f the instantaneous frequency and α the instantaneous amplitude at the start of the analysis window, and ϕ_a is the phase at the start of the window.

The phase advance is thus computed in the same way as in the stationary case:

$$\begin{aligned} \Delta \Phi_f^{t_a} &= \Phi_f^{H_a} - \Phi_f^0 \\ &= 2\pi f H_a + \frac{\psi}{2} (H_a)^2 + \phi_a - \phi_a \\ &= 2\pi f H_a + \frac{\psi}{2} H_a^2 \end{aligned} \quad (13)$$

References

- [1] T. F. Quatieri and R. J. McAulay, “Audio signal processing based on sinusoidal analysis/synthesis,” in *Applications of digital signal processing to audio and acoustics*. Springer, 2002, pp. 343–416.
- [2] X. Rodet and P. Depalle, “Spectral envelopes and inverse fft synthesis,” in *Audio Engineering Society Convention 93*. Audio Engineering Society, 1992.
- [3] P. Depalle, G. Garcia, and X. Rodet, “Tracking of partials for additive sound synthesis using hidden markov models,” in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, vol. 1. IEEE, 1993, pp. 225–228.
- [4] D. Arfib, F. Keiler, and U. Zölzer, “Time-frequency processing,” *DAFX: digital audio effects*, pp. 237–297, 2002.
- [5] G. Van Rossum, B. Warsaw, and N. Coghlan, “Style guide for python code,” Aug. 2013.
- [6] T. Karrer, E. Lee, and J. O. Borchers, “Phavorit: A phase vocoder for real-time interactive time-stretching.” in *ICMC*, 2006.
- [7] D. Barry, D. Dorran, and E. Coyle, “Time and pitch scale modification: A real-time framework and tutorial,” in *Conference papers*, 2008, p. 16.
- [8] F. Hammer, “Time-scale modification using the phase vocoder,” *Institute for Electr. Music and Ac, Graz Univ. of Dramatic Arts, Graz*, 2001.
- [9] J. Grünwald, “Theory, implementation and evaluation of the digital phase vocoder in the context of audio effects,” *Institute for Electr. Music and Ac, Graz Univ. of Dramatic Arts, Graz*, 2010.