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

Part I
Introduction

1 The company

2 Objective

3 Context of the project

4 Work environment and project management

Part II
Method Overview

5 Additive Synthesis (Time Domain)

6 Method Overview

Part III
The additive synthesis in frequency domain

7 Stationary Case

7.1 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 :

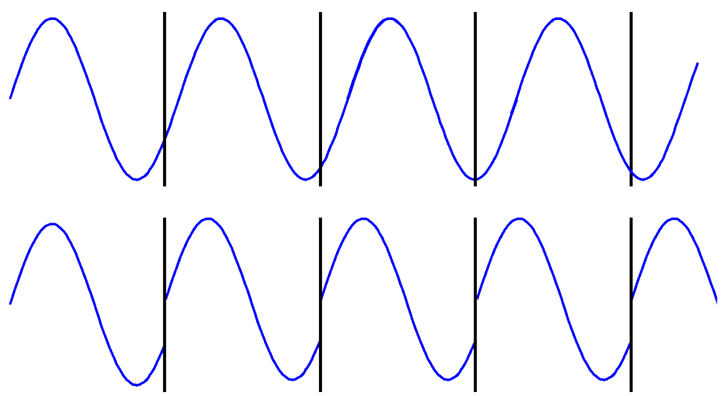

Figure 1: Coherent phase (above) vs. Incoherent phase (below) between frames

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 $2\pi\tilde{f}R_a$. To see details of the theoretical phase advance for the stationary case, please refer to the appendix (A.2).

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 [1], 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
```

```
_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)
```

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
_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)

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 follow[1] :

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

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)
_acrs : np.array _fcrs : np.array
__init__(self, amplitudes, frequencies, phases, acrs, fcrs) get_acr(self, k) get_fcr(self, k)

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 chirps[1] :

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

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

[1] Please look up section 9 page 7 for more details

Part VI
Documentation

14 Core module

14.1 Synthesizer

14.2 StationarySynthesizer

14.3 NonStationarySynthesizer

15 Spectrum generation module

15.1 SpectrumGenerator

SpectrumGenerator
_parameters : np.array _nfft : np.float _spectrum : np.array _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)

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)
_parameters : np.array _nfft : np.float _spectrum : np.array _window_size : np.float _analysis_hop : np.float _lobe_generator : np.array
__init__(self, window_size, parameters, nfft, analysis_hop) _add_lobe(self, k, lobe)

15.3 NonStationarySpectrumGenerator

```
NonStationarySpectrumGenerator (SpectrumGenerator)
```

```
_parameters : np.array
_nfft : np.float
_spectrum : np.array
_window_size : np.float
_analysis_hop : np.float
_lobe_generator : np.array
_regular_lut : np.array
--------------------------------------------------------
__init__(self, window_size, parameters, nfft, analysis_hop)
_add_lobe(self, k)
```

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

15.4 LobeGenerator

```
LobeGenerator
```

```
_window_type : np.float
_window_size : np.float
_nfft : np.float
_window : np.array
_lobe : np.array
--------------------------------------------------------
__init__(self, window_type, window_size, nfft)
_set_window_size(self, window_size)
_set_window_type(self, window_type)
_gen_lobe(self)
_get_lobe
```

This class generates a 11 points main lobe.

15.4.1 StationnaryLobeGenerator

```
StationaryLobeGenerator(LobeGenerator)
```

```
_window_type : np.float
_window_size : np.float
_nfft : np.float
_window : np.array
_lobe : np.array
_gen_lobe : np.array
--------------------------------------------------------
__init__(self, window_type, window_size, nfft)
_gen_lobe(self)
```

15.4.2 NonStationaryLobeGenerator

```
NonStationaryLobeGenerator(LobeGenerator)
```

```
_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 : np.array
- - - - - - - - - - - - - - - - - - - - - - - -
__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))
```

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
```

```
_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)
```

This module is not used by the mains yet. Please look up section **??** page **??** to know what this module do. The PhaseVocoder file gathers both the Stationary Phase Vocoder and the Non-Stationary Phase

16.2 StationaryPhaseVocoder

16.3 NonStationaryPhaseVocoder

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 :

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

$$\hat{\omega_k}(t_a^u) = \Omega_k + \frac{\Delta_p\Phi_k^u}{R_a} \tag{2}$$

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^{u-1}, \Omega_k) + R_s\hat{\omega_k}(t_a^u) \tag{3}$$

What (1) 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 (2) 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 (3) 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 :

$$s(n) = \alpha\cos(2\pi\tilde{f}n + \phi) \tag{4}$$

knowing only α, $\tilde{f}$ and ϕ.

We will also, to ease the process, assume that we know the application[1] $f_{w,\tilde{f}}(\phi) : \phi \mapsto \tilde{\phi}$ which takes into account the effect of windowing on the phase of the frame spectrum's phase[2].

[1] And 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.

[2] For more details on the theory, please read part II page 5

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 supposedly[3] 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 $\tilde{f} \notin \{\frac{2k\pi}{N}\}_{k=0...N-1}$. We then multiply the generated lobe by $\frac{A}{2}$ and set the lobe phase to $\tilde{\phi} + 2\pi\tilde{f}R_a$ [4]. 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 1 gives, for $k \in 1...N-1$ s.t $X(t_a^u, \Omega_k)$ is a bin of the lobe a phase shift error of $\tilde{\phi}$.
Then after 2 and 3 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 (7)), recursively, we can define the phase of the lobe the ith generated spectrum as :

$$\begin{cases} \tilde{\phi}_i = \tilde{\phi}_{i-1} + 2\pi\tilde{f}R_a \\ \tilde{\phi}_0 = \tilde{\phi} \end{cases} \tag{5}$$

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

[3]Because no actual analysis happened

[4]This 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.

$$s(n) \cdot \mathbb{1}_{[t_a^{u-1}, t_a^u]} \simeq \alpha \cos(2\pi \tilde{f} n + \phi) \tag{6}$$

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 (4) the phase advance from one frame to the other is elementary to compute:

$$\begin{aligned} \Delta\Phi_f^{t_a} &= \Phi_f^{t_a+H_a} - \Phi_f^{t_a} \\ &= 2\pi f(t_a + H_a) + \phi - 2\pi f t_a - \phi \\ &= 2\pi f H_a = 2\pi \tilde{f} R_a \end{aligned} \tag{7}$$

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:

$$s(t)\mathbb{1}_{[t_a^{u-1}, t_a^u]} \simeq (\alpha + \mu t) \sin(2\pi f t + \frac{\psi}{2} t^2 + \phi_a) \tag{8}$$

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:

$$\begin{aligned} \Delta\Phi_f^{t_a} &= \Phi_f^{H_a} - \Phi_f^0 \\ &= 2\pi f H_a + \frac{\psi}{2}(H_a)^2 + \phi_a - \phi_a \\ &= 2\pi f H_a + \frac{\psi}{2} H_a^2 \end{aligned} \tag{9}$$

References

[1] G. Van Rossum, B. Warsaw, and N. Coghlan, "Style guide for python code," Aug. 2013.