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 9, 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.

AudioGaming
NATURAL BORN INTERACTIVE

INP TOULOUSE ENSEEIHT

Contents

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.

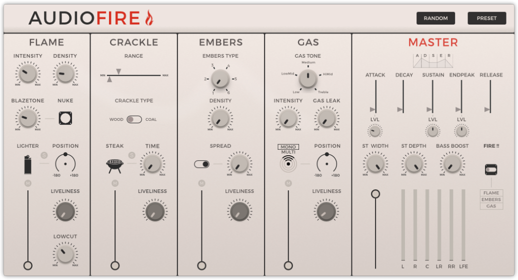


Figure 1: Audiofire: audio plug-in that recreates fire sound

2 Objective

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

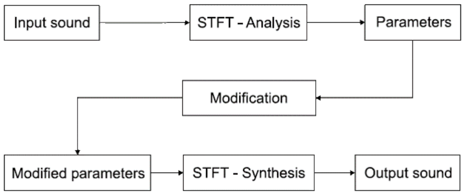


Figure 2: General approach for modifying a sound in the spectral domain

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.

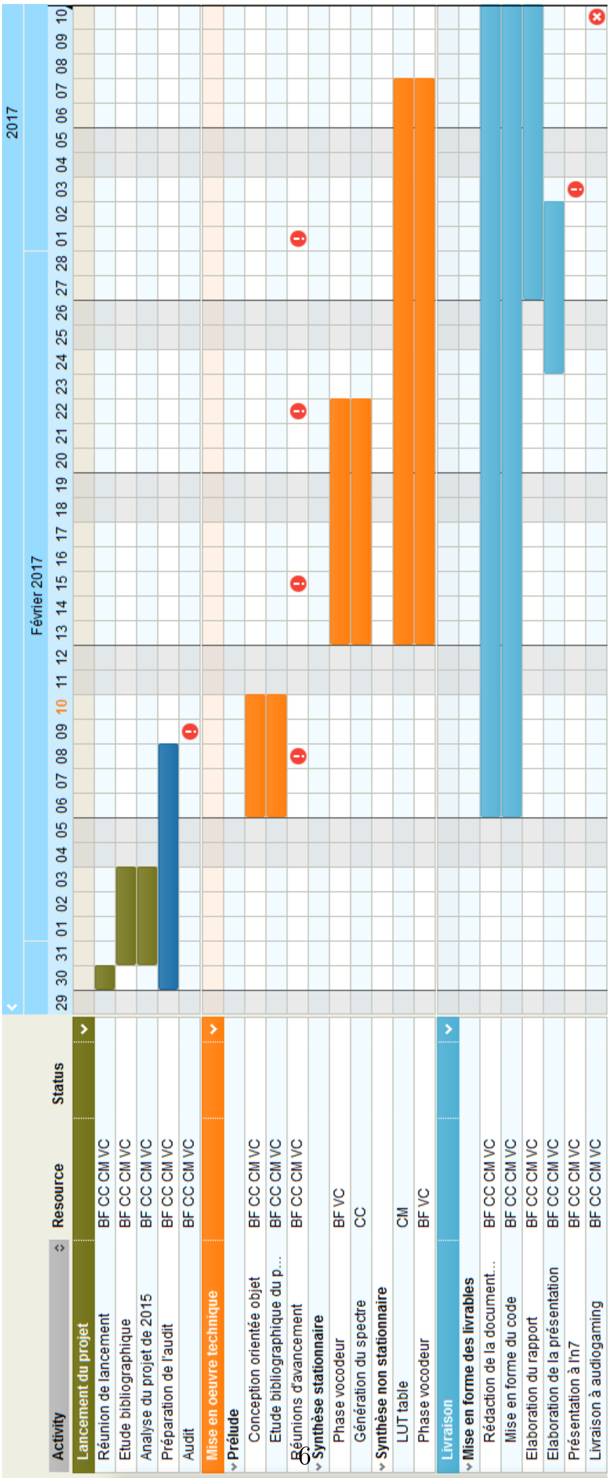
4 Work environment and project management

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)



Activity	Resource	Status
Lancement du projet		
Réunion de lancement	BF CC CM VC	
Etude bibliographique	BF CC CM VC	
Analyse du projet de 2015	BF CC CM VC	
Préparation de l'audit	BF CC CM VC	
Audit	BF CC CM VC	
Mise en oeuvre technique		
Prélude		
Conception orientée objet	BF CC CM VC	
Etude bibliographique du p...	BF CC CM VC	
Réunions d'avancement	BF CC CM VC	
Synthèse stationnaire		
Phase vocodeur	BF VC	
Génération du spectre	CC	
Synthèse non stationnaire		
LUT table	CM	
Phase vocodeur	BF VC	
Livraison		
Mise en forme des livrables		
Rédaction de la document...	BF CC CM VC	
Mise en forme du code	BF CC CM VC	
Elaboration du rapport	BF CC CM VC	
Elaboration de la présentation	BF CC CM VC	
Présentation à l'n7	BF CC CM VC	
Livraison à audiogaming	BF CC CM VC	

Part II
Sound synthesis in the frequency domain

5 General analysis/synthesis approach

5.1 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 finite sum of time varying sinusoids, which are called partials [1] added to a residual, that is, a stochastic process equivalent to a noise:

$$s(t) = \sum_{i=1}^{K} \alpha_i(t) \sin(2\pi f_i(t)t + \phi_i) + e(t) \tag{1}$$

Where the amplitude $\alpha_i(t)$ and the frequency $f_i(t)$ are smooth slowly varying functions.

Although in most cases, that is speech or musical instruments synthesis, the residual contribution is not negligible compared to the partials [1, 2] one can, in a first time, set the residual term to zero[1].

This synthesis method is also motivated by the fact that a great amount of research has been done towards analysing such sound models [2, 4]. 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.

However the main issue at stake 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. By doing so, we only stock

[1] In fact, the sinusoidal plus residual model is a direct consequence of the source/filter model [3], the synthesis of the residual term is thus more or less equivalent to filtering a white noise, which is, in its naive resolution, simpler than the sinusoids synthesis. This is why we mainly focus on the sinusoidal synthesis despite the non negligible nature of the residual term.

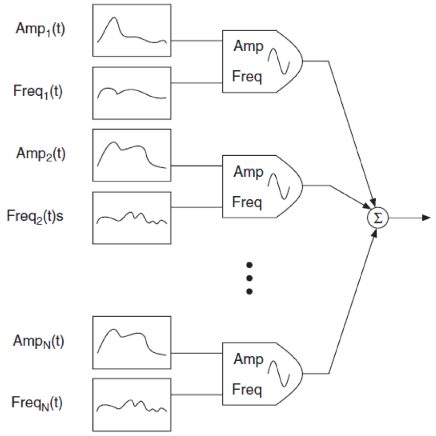


Figure 4: The additive synthesis

5.2 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, with Hanning, Blackman-Harris... window for instance (Figure 5):

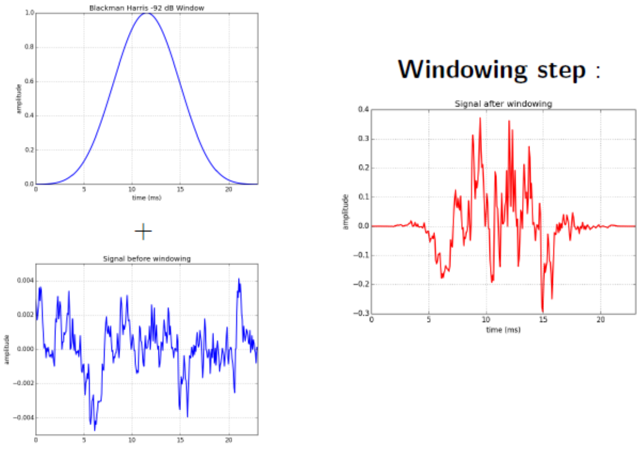


Figure 5: Windowing step

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.

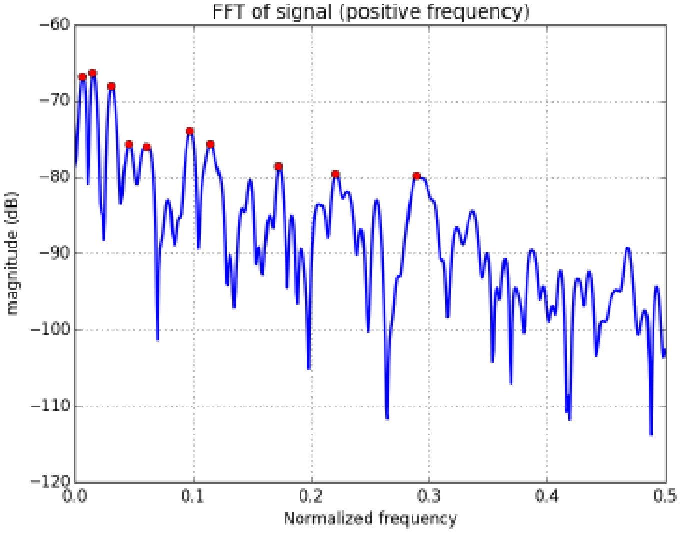


Figure 6: Peaks detection

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

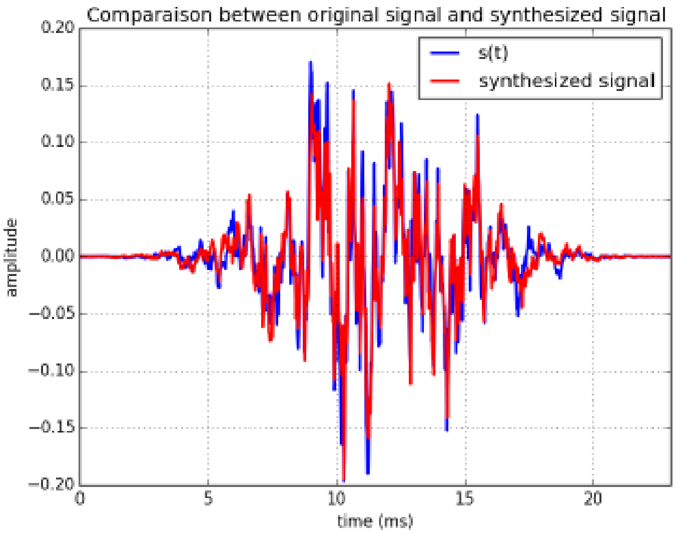


Figure 7: Synthesized frame vs Original frame

The image below is a little sum up of the method:

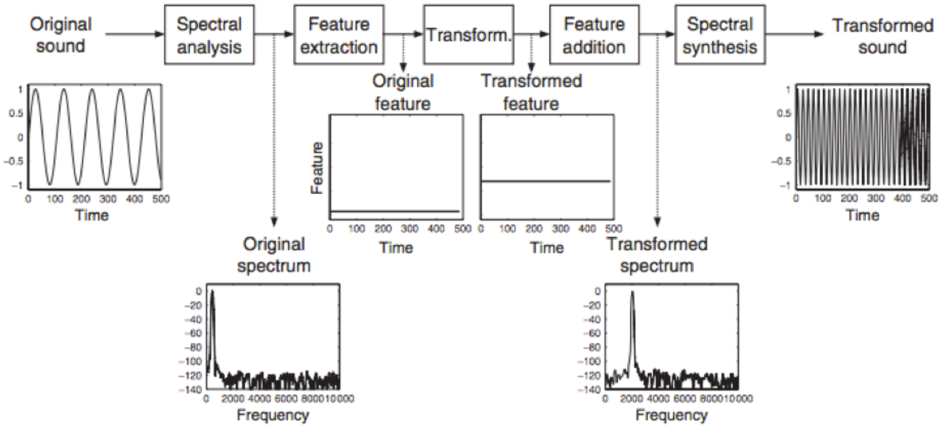


Figure 8: Block diagram of a higher-level spectral-processing framework

5.3 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 :

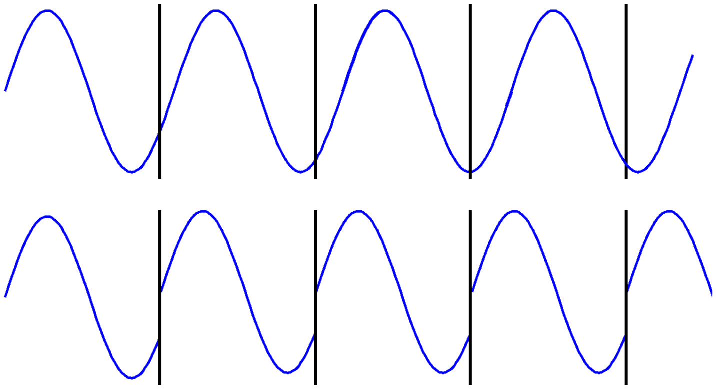


Figure 9: *Coherent phase (above) vs. Incoherent phase (below) between frames*

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 $2\pi\tilde{f}R_a$ [2].

6 Sinusoid synthesis in the frequency domain

6.1 Stationary Case

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

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

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:

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

[2] To see details of the theoretical phase advance for the stationary case, please refer to the appendix B.2 page 34.

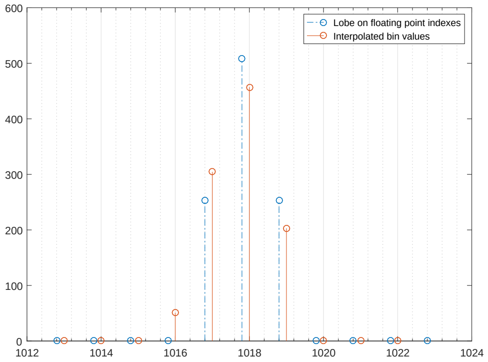


Figure 10: Interpolation of stored lobe values onto the FFT bins.

Where $W_{t_a}(m)$ is the DFT of the window w_{t_a}. Under the right assumptions on the window phase[3], 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 lobe[4], one can omit the side lobes with little to no error, ever more so since most of the error is 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 10, 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 5.3, that is to say that:

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

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 $t_a = k \times R_a$.

[3] Which involves circular shift to cancel the phase shift [5].

[4] At the condition that w_{t_a} is not rectangular.

6.2 Quasi-Stationary Case

Although most of what has been said before still holds in the quasi-stationary case, a few changes have to be made so that the synthesis work.

The signal is a non stationary signal which can be approximately considered as stationary given a small enough window:

$$s(n) \cdot \mathbb{1}_{[t_a^{u-1},t_a^u]} \simeq \sum_{i=1}^{K} \alpha_i^u \sin(2\pi \widetilde{f_i^u} n + \phi_i^u) \tag{5}$$

However, while the synthesised spectrum amplitude remain unchanged, the phase has to be corrected to take into account both the fact that the sinusoid initial phase ϕ_i has to be passed along the sinusoid track if its frequency where to change, and phase advance corrections due to the non-stationary nature of the signal.

Indeed, although we assume the signal to be stationary in a short frame, this mainly implies that the synthesised lobe corresponding to the sinusoid is identical to a stationary sinusoid lobe. However, because of the recursive nature of the synthesis phase, errors in phase advances due to either a loss of precision in frequency estimation[5] or applying stationary phase shift as seen in (4), may lead to significant phase deviation after a few iteration. This may in turn leads to significant phase de-coherence in the case of a note onset, that is the appearing of a new sinusoid.

One proposed method relies on a Phase Vocoder[6] to take into account and corrects unexpected phase shift due to non-stationarity in the pre-analysed reference signal. The phase is then set to the reference signal phase corrected for a possible difference in analysis and synthesis hop size. However, because we rely on the analysis phase to ensure correct phase propagation, the method loses the initial flexibility of a true sinusoidal model.

Another possible approach is to pass the true theoretical phase advances along the track. Although it implies that one is able to track the sinusoid evolution (the complexity of which is substantial [3,6,7]), this has the advantage of keeping a truly sinusoidal framework as in the stationary case.

The following has not been implemented, but the idea is to estimate coarsely the non-stationary *Frequency Change Rate* (ψ) between two frame and apply non-stationary sinusoid phase shift.

The linear chirp is written as follow[7]:

$$s(t) \propto \sin\left(2\pi\left(f + \frac{1}{2\pi}\frac{\psi}{2}t\right)t + \phi_a\right) \tag{6}$$

[5] As one will see in the true non-stationary case, amplitude and phase shifts have significant influence one the lobe appearance

[6] See annex A page 30 for more informations on Phase Vocoders.

[7] The non-stationary case hypotheses and framework can be found in the section 6.3 below.

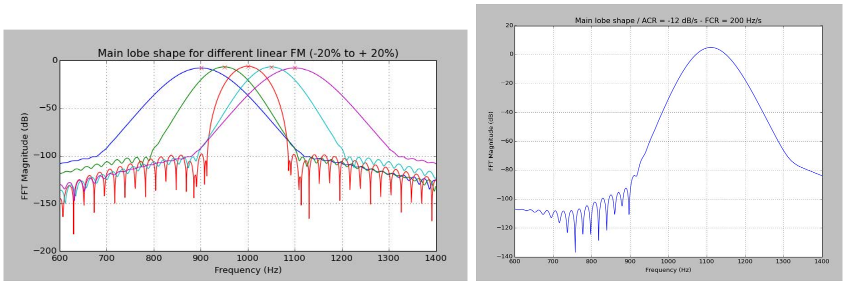


Figure 11: Frequency and amplitude modulation affects the shape of the lobe

Thus the frequency shift between two consecutive frames is:

$$\begin{aligned}\Delta f &= f_{H_a} - f_0 \\ &= f_0 + \frac{\psi}{4\pi} H_a - f_0 \\ &= \frac{\psi}{4\pi} H_a\end{aligned} \tag{7}$$

We can thus estimate ψ between two frames:

$$\hat{\psi}_{t_a} = \frac{4\pi}{H_a}\left(f_{(t_a+H_a)} - f_{t_a}\right) \tag{8}$$

The estimated phase advance (25) computed in annex B.2.2 page 34 is thus:

$$\Delta\Phi_f^{t_a} = 2\pi f_{t_a} H_a + \frac{\psi}{2} H_a^2 = 2\pi H_a \left(f_{t_a} + f_{(t_a+H_a)} - f_{t_a}\right) = 2\pi f_{(t_a+H_a)} H_a \tag{9}$$

6.3 Non-Stationary Case

The non-stationary case is the most general one. For simplicity and without loss of generality we will assume in the following that the signal consists in a single sinusoid, that is :

$$s(t) = \alpha(t)\sin(2\pi f(t)t + \phi) \tag{10}$$

With $\alpha, f \in \mathcal{C}^1$ such that they vary slowly compared to the sinusoid, and ϕ a constant corresponding to the phase at time 0.

When applying the STFT we may not, as opposed to section 6.2, assume the signal to be stationary in a small enough window, despite the slow variation of amplitude and frequency hypothesis. This is because the shape main lobe of the Fourier transform of the sinusoid is greatly dependant on amplitude and frequency modulation [8, 9], as showed on Figure 11.

What it implies is that we may no longer use the same lobe[8] for every sinusoid as in section 6.1. Because of time constraints it is not possible to compute the exact values of each lobes, the proposed method bellow was first mentioned and tested by the 2015 projects we built upon, it relies on interpolating lobe from a *Look-up Table* and provides both excellent precision and small computational and memory costs. However this implies a simplified mathematical model.

6.3.1 Mathematical Model

Given a small enough analysis window, the signal can be modelled [10] as:

$$s(t) = (\lambda + \mu t) \sin \left(2\pi f t + \frac{\psi}{2} t^2 + \phi \right) \tag{11}$$

This polynomial model can be viewed either as truncated Taylor expansion of more general amplitude and frequency modulations or as a linear chirp. We introduce two additional parameters, the *Amplitude Change Rate* μ and the *Frequency Change Rate* ψ, the latter has already been introduced in section 6.2.

It is important to note however that (11) builds upon the slow variation hypothesis introduced before, this means that, although it is correct for most real sounds, there exist a number of case where (11) can not be applied. Most notably, either in the cases of rapid harmonic sound onset or in the case of very fast frequency modulation, the model before is invalid and will raise poor results. Those two cases belong to what are called transients of a sound. Their synthesis is a very difficult issue and falls outside of the scope of the present document, however, it is important to point out the limits of our model.

6.3.2 The look-up table

The principle of the method is to create a regular or irregular ACR/FCR grid. For each point of the grid which is represented by an ACR/FCR couple, we store an eleven-point main lobe in the look-up table. Hence, the size of the look-up table is equal to number_of_ACR times number_of_FCR. Then, we can find the lobe for the couple of ACR/FCR we want by interpolating.

[8] One may argues that the main advantage of FFT^{-1} synthesis is lost because the computational cost gains offered by the method relied mainly on the systematic nature of lobe synthesis. However, while this imposes a non negligible additional computational strain, the number of points to compute is still one or two order of magnitudes less than in the temporal domain for a given frame.

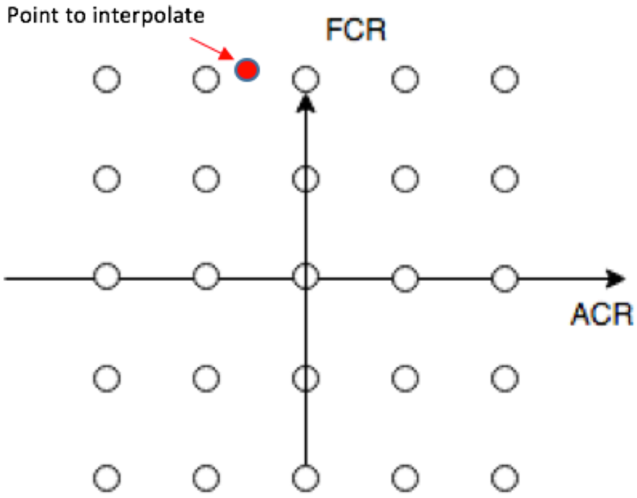


Figure 12: Regular ACR/FCR grid

The look-up table should not be too big so it does not take too much memory. As a consequence, we have to find a compromise between the size of the look-up table and the precision of the interpolation.

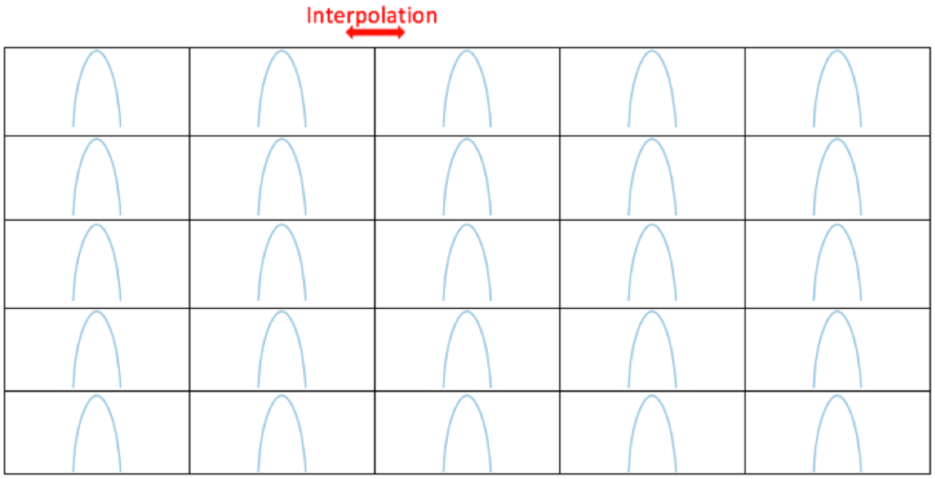


Figure 13: Interpolation of stored lobe values onto the FFT bins

We tested different kind of interpolation because the interpolation that gives the best results is not the same for the magnitude as for the phase. Finally, we figured out that the interpolation that works better for the magnitude is the cubic one, whereas the one that works better for the phase is the linear one which sounds quite logical giving the form of the curves. The user can choose the interpolation method he wishes for both amplitude and phase.

Finally, the non-stationary phase advance equation (25) computed in annex B.2.2 page 34 is:

$$\Delta\Phi_f^{t_a} = 2\pi f_{t_a} H_a + \frac{\psi}{2} H_a^2 \tag{12}$$

Part III

Results

7 Stationary Case

8 Non-Stationary Case

Part IV
Code structure and conventions

9 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 10.1) 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.

9.1 Naming conventions

Naming conventions are freely adapted from Python recommended conventions defined in PEP8 [11], 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.

9.2 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.

9.2.1 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 × operation as well as the × = 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.

9.2.2 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

9.2.3 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

10 Code structure

10.1 General structure

10.2 Class structure

[1] Please look up section 6.3 page 14 for more details

Part V
Documentation

11 Core module

11.1 Synthesizer

```
Synthesizer
_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_synthetizer (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)
```

The aim of the Synthesizer is to set all the parameters which are used to make the synthesis, that is to say the type and the size of the window, the FFT number and the hop size for the analysis and the synthesis. Moreover, it returns the spectrum of the signal.

11.2 StationarySynthesizer

```
StationarySynthesizer
-------------------------------------------------------------
_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)
```

11.3 NonStationarySynthesizer

```
NonStationarySynthesizer
-------------------------------------------------------------
_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)
```

12 Spectrum generation module

12.1 SpectrumGenerator

SpectrumGenerator
_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)

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.

12.2 StationarySpectrumGenerator

StationarySpectrumGenerator (SpectrumGenerator)
_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)

12.3 NonStationarySpectrumGenerator

NonStationarySpectrumGenerator (SpectrumGenerator)
_parameters : Parameters
_nfft : np.float
_spectrum : Spectrum
_window_size : np.float
_analysis_hop : np.float
_lobe_generator : NonSationaryLobeGenerator
_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.

12.4 LobeGenerator

```
LobeGenerator
```

```
_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
```

This class generates a 11 points main lobe.

12.4.1 StationnaryLobeGenerator

```
StationaryLobeGenerator(LobeGenerator)
```

```
_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)
```

12.4.2 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 : 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))
```

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 choose the kind of interpolation he or she wants to use for the magnitude, the phase and the frequency. Besides, he can choose the size of the LUT to build by giving the number of ACRs and FCRs.

13 Phase Vocoder module

13.1 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 !

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

13.2 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 [7, 12, 13] and annex .

13.3 NonStationaryPhaseVocoder

```
NonStationaryPhaseVocoder
```

```
get_pv_spectrum(self)
```

This module is not complete ! The detailed algorithm called Scaled-Phase Locking is also described in the paper [7]. 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 [14].

Part VI
Conclusion

Six weeks was a very short period for such a research subject. Indeed, we lost most of the time trying to understand the phase vocoder and its utility in the overall project. Moreover, we are not sure that it can work in real-time conditions because we could not evaluate the real performances of such a method. Indeed, we implemented it in Python which is a high-level interpreted programming language.

For now, the stationary synthesis works quite well, and we succeeded in creating the look-up table and in interpolating the non-stationary lobes with it.

An interesting thing to do would be to get a real **analysis** method to get the parameters from a real sound signal and then to listen to the synthesized sound. Indeed, without an analysis tool, we cannot really test our synthesis method.

In this project, we learned Python language and how to manage the codes with GitHub. We also studied the phase of signals, something that in signal processing we didn't study much and which we don't really bother with.

Part VII
Appendix

A Phase Vocoder

A.1 Simple Phase Vocoder algorithm

The phase vocoder is a popular algorithm for time-stretching audio without changing the pitch. In this project, our supervisor suggested we use a phase vocoder to correct ans advance the phase. ——- That is what the phase vocoder do, correct the phase. But it correct the phase when the hop size are different between the analysis part and the synthesis part (respectively R_a and R_s, or hop_a and hop_s).——— The approach is described in the scheme below :

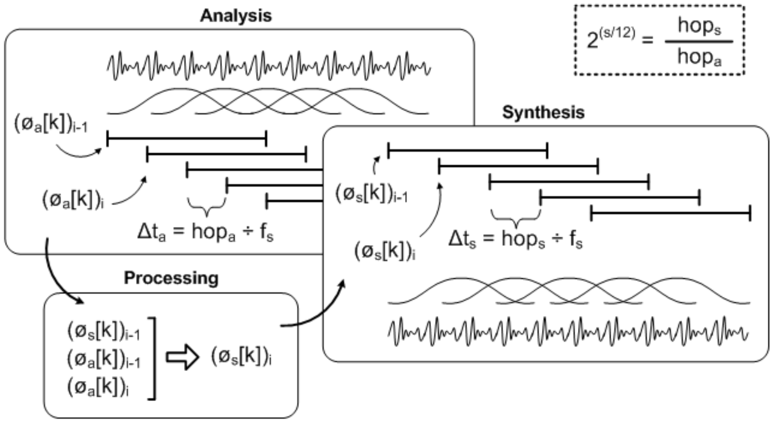


Figure 14: Phase Vocoder scheme

With different analysis (R_a) and synthesis (R_s) hop size, we can change the pitch of the original sound. This is not what we want to do in this project. We just want to synthesise with a correct phase between frames. The main idea of using a phase vocoder was to correct the phase with the two same hop size without bothering for the phase anymore. But it didn't work. We explain why the phase vocoder is useless in our case in the part below (B).

The general Phase Vocoder is expressed in 3 equations as follow :

$$\Delta\Phi_k^u = \angle X(t_a^u, \Omega_k) - \angle X(t_a^{u-1}, \Omega_k) - R_a\Omega_k \tag{13}$$

$$\hat{\omega_k}(t_a^u) = \Omega_k + \frac{\Delta_p\Phi_k^u}{R_a} \tag{14}$$

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^{u-1}, \Omega_k) + R_s\hat{\omega_k}(t_a^u) \tag{15}$$

What (13) 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 (14) 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 (15) 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.

A.2 Scale phase locking

We can't use the same phase vocoder in the non stationary case because, in this case : sine waves that switch from a frequency bin to another bin. The phase might change a lot from one frame to another. Thus we have to use a refined version of the phase vocoder, there are several algorithm to do this explained in [7]. Here we choose to use the Scaled-Phase Locking algorithm. This method takes into account the frequency trajectory of each lobes.

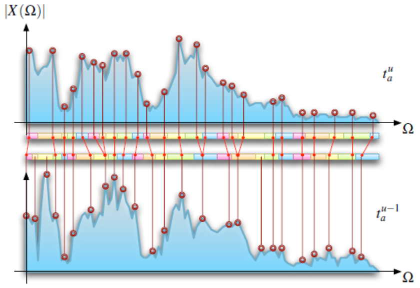


Figure 15: *Peak coherence from a frame to another*

We find each region corresponding to each peaks, and then we use the phase vocoder algorithm for each region. To choose the peak corresponding to a peak in the previous frame we take the nearest.

$$\hat{\omega}_{k_1}(t_a^u) = \Omega_k + \frac{\Delta_p \Phi_k^u}{R_a} \tag{16}$$

Where:

$$\Delta\Phi_{k_1}^u = \angle X(t_a^u, \Omega_{k_1}) - \angle X(t_a^{u-1}, \Omega_{k_0}) - R_a \Omega_{k_1} \tag{17}$$

Hence,

$$\angle Y(t_s^u, \Omega_{k_1}) = \angle Y(t_s^{u-1}, \Omega_{k_0}) + R_s \hat{\omega}_{k_1}(t_a^u) \tag{18}$$

To determine which peak in frame u $-$ 1 corresponds to the peak at Ω_k in frame u, the peak in frame u$-$1 that is closest to channel Ω_{k1} is identified. The non-peak bins are said to belong to the closest peak's region of influence, and are then phase-locked to that peak using the phase-locking equation:

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^u, \Omega_{k_1}) + \beta[\angle X(t_a^u, \Omega_k) - \angle X(t_a^u, \Omega_{k_1})] \tag{19}$$

where β is a phase scaling factor. In this way the phase relations between each peak channel and its neighboring nonpeak channels are carried over from the original signal to the time-stretched signal which helps to preserve vertical phase coherence.

B Phase advance and propagation along the signal

B.1 On the first attempt at Phase Vocoder use

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.

B.1.1 "Pure" synthesis

We wish to synthesise 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 :

$$s(n) = \alpha \cos(2\pi \tilde{f} n + \phi) \tag{20}$$

knowing only α, $\tilde{f}$ and ϕ.

We will also, to ease the process, assume that we know the application[9] $f_{w,\tilde{f}}(\phi) : \phi \mapsto \tilde{\phi}$ which takes into account the effect of windowing on the phase of the frame spectrum's phase[10].

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[11] 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

[9] 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.

[10] For more details on the theory, please read part 5 page 7

[11] Because no actual analysis happened

lobe by $\frac{A}{2}$ and set the lobe phase to $\tilde{\phi} + 2\pi\tilde{f}R_a$ [12]. 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

(13) 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 14 and 15 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 synthesised 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 (23)), recursively, we can define the phase of the lobe the ith generated spectrum as :

$$\begin{cases}\tilde{\phi}_i = \tilde{\phi}_{i-1} + 2\pi\tilde{f}R_a \\ \tilde{\phi}_0 = \tilde{\phi}\end{cases} \tag{21}$$

B.1.2 "Parametered" synthesis

In this case, we will not synthesise a truly stationary signal but we assume that the signal is quasi-stationary, which is to say that given a small enough analysis window, it can be considered stationary within that frame.

$$s(n) \cdot \mathbb{1}_{[t_a^{u-1}, t_a^u]} \simeq \alpha \cos(2\pi\tilde{f}n + \phi) \tag{22}$$

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

[12] 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.

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 B.1.1 above still holds, if we update the parameters $\tilde{f}$ and α in the spectrum generation.

B.2 Theoretical phases advance

B.2.1 Stationary case

For stationary signals expressed as in (20) the phase advance from one frame to the other is elementary to compute:

$$\begin{aligned}\Delta\Phi_f^{t_a} &= \Phi_f^{t_a+H_a} - \Phi_f^{t_a} \\ &= 2\pi f(t_a + H_a) + \phi - 2\pi f t_a - \phi \\ &= 2\pi f H_a = 2\pi \tilde{f} R_a\end{aligned} \tag{23}$$

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

B.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:

$$s(t)\mathbb{1}_{[t_a^{u-1}, t_a^u]} \simeq (\alpha + \mu t)\sin(2\pi f t + \frac{\psi}{2}t^2 + \phi_a) \tag{24}$$

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:

$$\begin{aligned}\Delta\Phi_f^{t_a} &= \Phi_f^{H_a} - \Phi_f^{0} \\ &= 2\pi f H_a + \frac{\psi}{2}H_a^2 + \phi_a - \phi_a \\ &= 2\pi f H_a + \frac{\psi}{2}H_a^2\end{aligned} \tag{25}$$

Please not however that although the phase advance term is rather simple, (24) implies that we have a sinusoidal continuation scheme at hand to ensure a correct propagation of the phase ϕ_a along the sinusoid track, and while much work has been done on the subject [7, 15], this is a complicated issue.

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] R. McAulay and T. Quatieri, “Speech analysis/synthesis based on a sinusoidal representation,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, no. 4, pp. 744–754, 1986.

[4] 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.

[5] D. Arfib, F. Keiler, and U. Zölzer, “Time-frequency processing,” *DAFX: digital audio effects*, pp. 237–297, 2002.

[6] X. Serra and X. Serra, “A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition,” 1989.

[7] T. Karrer, E. Lee, and J. O. Borchers, “Phavorit: A phase vocoder for real-time interactive time-stretching.” in *ICMC*, 2006.

[8] J. J. Wells, “Methods for separation of amplitude and frequency modulation in fourier transformed signals.”

[9] J. J. Wells and D. T. Murphy, “High accuracy frame-by-frame non-stationary sinusoidal modelling,” in *Proc. of the 9th International Conference on Digital Audio Effects (DAFx-06).* Citeseer, 2006, pp. 253–258.

[10] S. Marchand and P. Depalle, “Generalization of the derivative analysis method to non-stationary sinusoidal modeling,” *Addison Wesley*, 1994.

[11] G. Van Rossum, B. Warsaw, and N. Coghlan, “Style guide for python code,” Aug. 2013.

[12] D. Barry, D. Dorran, and E. Coyle, “Time and pitch scale modification: A real-time framework and tutorial,” in *Conference papers*, 2008, p. 16.

[13] F. Hammer, “Time-scale modification using the phase vocoder,” *Institute for Electr. Music and Ac, Graz Univ. of Dramatic Arts, Graz*, 2001.

[14] 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.

[15] X. Amatriain, J. Bonada, A. Loscos, and X. Serra, “Spectral processing,” *DAFX: Digital Audio Effects*, pp. 373–438, 2002.