VIRTUAL SUSTAIN PEDALLING FOR GUITAR BY SINUSOIDAL MODELLING

by

JOAN PRAT RIGOL

Department of Communication System
École Polytechnique Fédérale de Lausanne

2011-2012

Approved by:

Major Professor
Mihailo Kolundzija

Contents

List of Figures

Acknowledgements

I am very grateful to my supervisor and at the same time, my guide of this project. He has given me the ideas and the necessary initiative to carry on with this little project. I also want to thank him for his patience due to the fact of having a student whose English is not his strong point and the thousand and one times Mihailo had to repeat things so that I could understand.

I would also like to thank all the friends I have met in the Erasmus experience, because they have allowed me to enjoy the experience, and therefore I have been able to perform with enthusiasm in the project.

I want to thank you also to my "English teachers", Montserrat Voltà. She has helped me to correct my report and to improve my English.

And finally, I want to thank all my family who have always been by my side and have not felt the distance that separated us and they have let me perform one of the greatest experiences that undoubtedly it will keep forever as one of the best.

Agraïments

M'agradaria donar les molt sinceres gràcies al meu tutor, i a la vegada, el meu guia durant aquest projecte. Ell m'ha donat idees i l'empenta necessària per a poder tirar endavant aquest petit projecte. També jo vull agrair la paciència que ha comportat haver de tenir a un alumne del qual la llengua anglesa no era el seu fort i per les mil i una vegades que ell ha hagut de repetir les coses per a que les pogués entendre.

Voldria també donar les gràcies a tots els amics que he conegut en aquesta experiència Erasmus, ja que m'han permès gaudir d'ella, i per tant, poder rendir amb entusiasme en el projecte.

Vull donar les gràcies també a la meva "professora d'anglès", Montserrat Voltà. Ella m'han ajudat a corregir la meva memòria i a millorar el meu Anglès.

I per acabar, agrair a tota la meva família, que sempre han estat al meu costat fent que la distància que ens separava fos nul·la i permetent-me du a terme una de les més grans experiències de la que sense dubte, quedarà guardada per sempre com una de les millors.

Preface

In this project you will find a new effect for a guitar player called Virtual Guitar Sustain that allows the player to play a solo guitar while a chord is sustained in the background. We present a study on the guitar sound and we develop a program that will allow us sustain a guitar chord whenever we want to.

1. Introduction

1.1. Motivation of this project

The guitar is one of the most popular instruments in the world. The number of guitar players is always unknown because every day someone new picks up the guitar and leans to play. But we can conclude that this number is a magnitude of millions. And each one is always looking for new pedal distortions; new pedal effects that make a new sound for a guitar and also new styles to play the guitar.

Nowadays, the digital world is growing and gaining space through the musician's community. Not only for its cheap price compared with the analogue world, also for its versatility. You could have all these pedals reduced in one pedal, or one computer which is more practical, you can reduce space and have less problems saving the sound presets, and a lot of advantages.

The purpose of this work is to introduce a new digital pedal effect to add to the infinity of the effects in the guitar world effect. This effect called Virtual Sustain, generates a sound guitar based on a sample of the guitar played, its spectrum, to allow the player to play over the chord. Then, with only one guitar one can play the chord and then play over it. To achieve it, the hardware and the software must guarantee response within strict time constraints. Then, the implementation of the program in real time means that the incoming sound guitar has to be recorded, processed and then reproduced at the same time and with a delay than less than 10 milliseconds, which is a tolerable delay for a human being. A person cannot find out that there is a delay if it is less than 10 miliseconds.

1.1. Report Structure

This report is divided in 5 chapters, including the first introduction chapter and the final conclusions and future work lines. The remaining chapters follow the chronological order of the project development, from the guitar sound analysis to the development of the program in Matlab, which you can see on Appendix A.

Chapter 2 introduces the sound of a guitar, looking at its spectrogram and to extract the most significant characteristics of this sound and to understand why we are using Sinusoidal Modeling

Chapter 3 explains the Sinusoidal Model, starting for the formulation of this model which is a parametric one, going over the different parts that lead us to have the Sinusoidal model of the sound.

In Chapter 4 a different aspect of the guitar sound which was not find out in the previous ones is going to lead us to a more accurate model of the synthetic chord.

Finally, chapter 5 presents the conclusions reached with the program developed during this PFC project and some improvements and future work.

Additionally, in the appendix A the program developed in Matlab code is presented to the reader in order to figure out how everything that is explained in this PFC is implemented.

2. Guitar sound

EQUATION SECTION 2The guitar sound signal we have to synthesize has three principal characteristics that we should know in order to synthesize it. Its bandwidth depends on the kind of guitar that is in use, there are guitars with 7 strings, 12 strings, or guitars that are tuned in a different tune as usual. So we must take these things into account in order to use the program. We will discuss about the most common guitar with 6 strings and tuned as we can see

● Guitar Frets & Their Notes vs Frequencies

E — Note Name
329 — Frequency [Hz]

	OPEN	1F	2F	3F	4F	5F	6F	7F	8F	9F	10F	11F	12F	13F	14F	15F	16F	17F	18F	19F	20F
1st	E 329	F 349	F# 370	G 392	G# 415	A 440	A# 466	B 494	C 523	C# 554	D 587	D# 622	E 659	F 698	F# 740	G 784	G# 831	A 880	A# 932	B 988	C 1047
2nd	B 247	C 262	C# 277	D 294	D# 311	E 329	F 349	F# 370	G 392	G# 415	A 440	A# 466	B 494	C 523	C# 554	D 587	D# 622	E 659	F 698	F# 740	G 784
3rd	G 196	G# 208	A 220	A# 233	B 247	C 262	C# 277	D 294	D# 311	E 329	F 349	F# 370	G 392	G# 415	A 440	A# 466	B 494	C 523	C# 554	D 587	D# 622
4th	D 147	D# 156	E 165	F 175	F# 185	G 196	G# 208	A 220	A# 233	B 247	C 262	C# 277	D 294	D# 311	E 329	F 349	F# 370	G 392	G# 415	A 440	A# 466
5th	A 110	A# 117	B 123	C 131	C# 139	D 147	D# 156	E 165	F 175	F# 185	G 196	G# 208	A 220	A# 233	B 247	C 262	C# 277	D 294	D# 311	E 329	F 349
6th	E 82	F 87	F# 92	G 98	G# 104	A 110	A# 117	B 123	C 131	C# 139	D 147	D# 156	E 165	F 175	F# 185	G 196	G# 208	A 220	A# 233	B 247	C 262

Figure 2.1: Guitar Frets and their notes vs Frequencies

in Figure 2.1

The lowest guitar frecuency is 82 Hz, as Figure 2.1 shows. Then we should filter the sound to avoid wrong detection in low frequencies. The VGuitarSustain program has a parameter called `fc` to avoid peak detections under this cutoff frequency.

In Figure 2.1 we can also see that the worst case for frequency resolution is when the guitarist plays A from the 6^{th} string and A# from the 5^{th} string. Thus, a frequency resolution of approximately 7 Hz is required to detect all the notes of a chord. Then, for a sampling frequency of 44100 Hz and a Blackman window with a main lobe width of 6 bins leads us to a window length of 37800 (860 ms)! This leads us to a really good frequency resolution but a bad time resolution. We will discuss about it in the next chapter and in chapter 4.

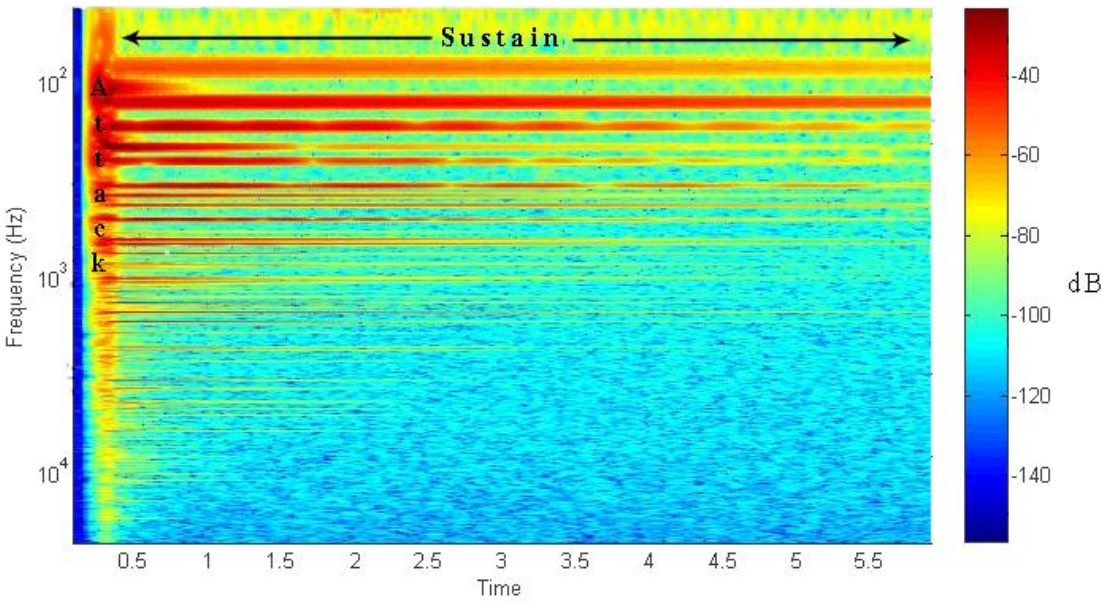

Figure 2.2: E chord's spectrogram.

Another aspect of the sound to synthesize is its evolution in time-frequency domain. Plotting the spectrogram of a chord, Figure 2.2, we can see a first part called attack or transient part, and then a stable part, called sustain, which has sinusoids and we cannot appreciate a change in frequency and the distribution is not homogenous, due to the fact that each string has its own harmonics added. So then, we have to synthesize an inharmonious (polyphonic) sound that is stable in time.

Therefore, a good method for analysing and synthesizing this type of sounds is a technique called sinusoidal modelling, which models a signal with a sum of sinusoids.

3. Sinusoidal Modeling

EQUATION SECTION (NEXT)Sinusoidal modeling is an STFT based analysis/resynthesis technique. It is based on modeling the time-varying spectral characteristics of a sound as sums of time-varying sinusoids. The sound $s(t)$ is modeled by

$$s(t) = \sum_{r=1}^{R} A_r(t)\cos[\theta_r(t)] \tag{3.1}$$

where $A_r(t)$ and $\theta_r(t)$ are the instantaneous amplitude and phase of the rth sinusoid, respectively, and R is the number of sinusoids.

To obtain a sinusoidal representation of a sound, an analysis is performed in order to estimate the instantaneous amplitudes and phases of the sinusoids. The estimation of these parameters begins generally by computing the STFT of the sound, then, detecting the spectral peaks, measuring the magnitude, frequency and phase of each one and finally tracking the peaks to obtain a time varying sinusoidal tracks.

3.1. STFT

The sound is a continuous signal and for the analysis of this signal we use the STFT. The papers [1] [2] talk about this STFT method and its mathematical formulation. Conceptually, it is the evolution in the frequency domain of the sound to analyse, thus, we have $X(f,t)$ that depends on time. In Figure 3.1 we can see the STFT for a different time moments (*t=0, t=4000 and t=9000*).. There are 3 parameters to take into account, the type of window, the window size and the period with which we take the FFT, called the hop size.

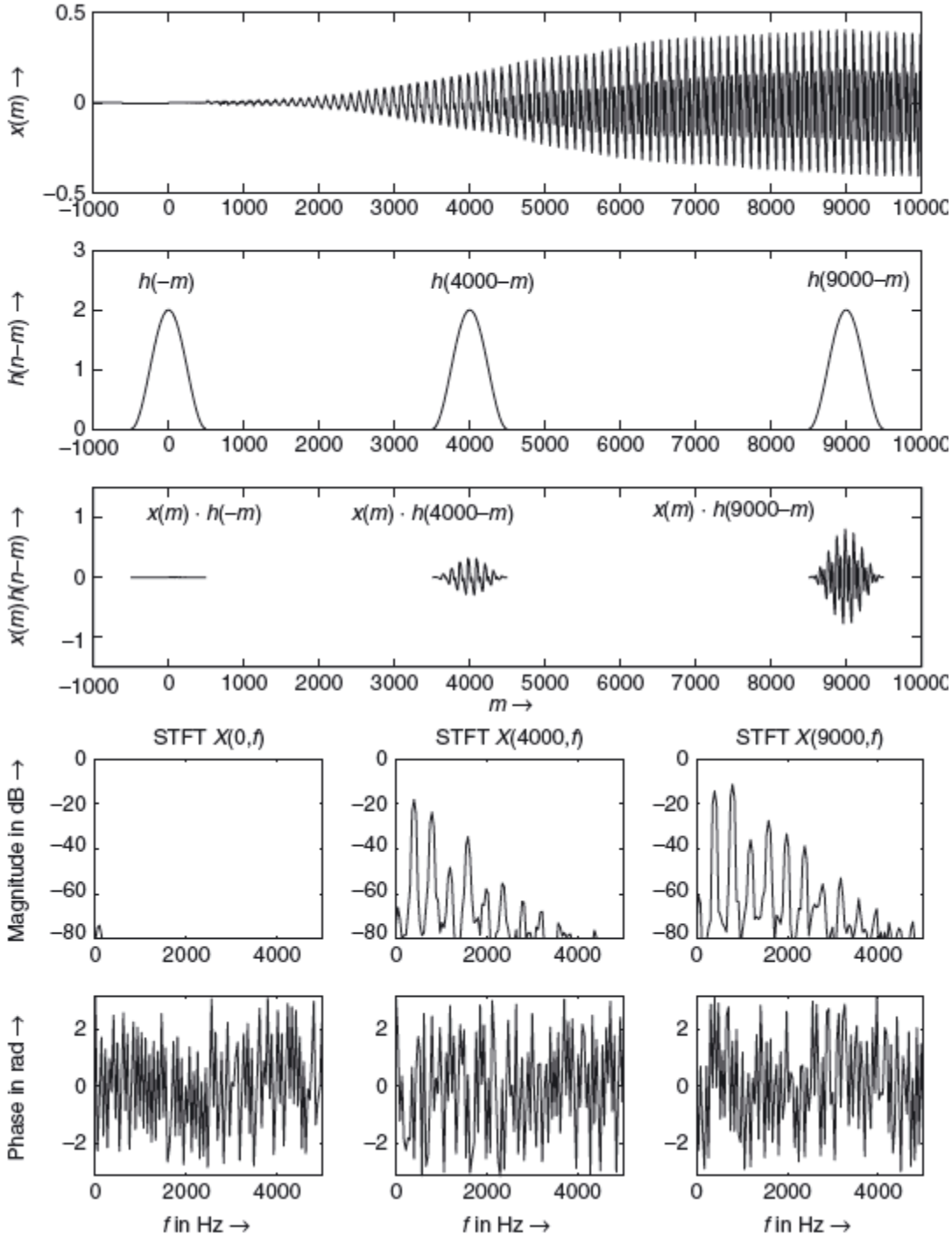

Figure 3.1: Sliding analysis window and short-time Fourier transform. Source: [5]

The choice of the window type is done according to [3]. There is a relationship between the window type and the zero-padding chosen to have a certain bias in frequency, amplitude and phase estimation. Hence, VGuitarSustain uses Blackamn window, which is a good choice to

have a low bias and a small zero padding (longer zero-padding increases the length of FFT, and consequently increases the computation time.).

One of the most crucial parameter choices is the length of the analysis window. This will control the width of the main lobe in the frequency domain and, as result, it will determine the frequency resolution of the analysis. As we were arguing in 2, the resolution that we would like to have is at least 7 Hz. This is not possible due to two reasons: nowadays we don't have enough computational power to calculate it in realtime so then, a big delay will appear in real time. The other issue is that if a sinusoid has an amplitude variation into the frame window, the measure will have some error.

In our experiment, we have observed that with a length of more than 8000 samples it is

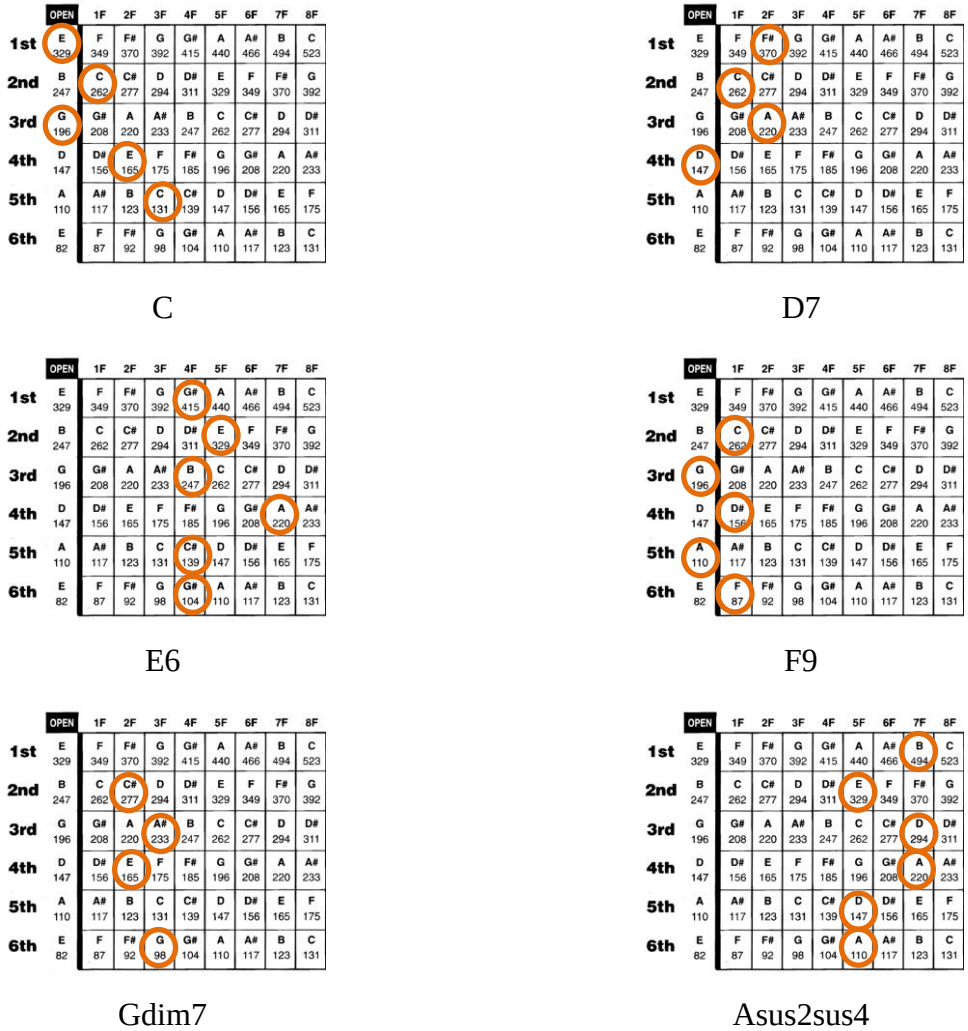

Figure 3.2: Different chords. The circles are the notes played in each chord

enough to detect the peaks without a big change on the synthesized sustain.

The resolution f_r in Hz is given by

$$f_r = \frac{\Delta w}{M} f_s \tag{3.2}$$

where f_s is the sampling rate and Δw is the desired window separation between the maximums peaks, in bins.

VGuitarSustain uses, as we have said in this chapter, the Blackamn window. The authors in [3] make an extensive study of the minimum allowable frequency separation (MAFS) for various window types. The MAFS is the smallest separation that allows the peak detection and does not introduce significant bias in the peak interpolation due to interference from a neighboring peak. With a spectral oversampling (zero padding) factor of 2, the MAFS for the Blackman window is 3.53 bins. VGuitarSustain uses an integer number of MAFS, 4 is the bin separation used. As a result of what has been said, the resolution obtained for M = 8191, f_s = 44100 Hz and Δw = 4 is 21.5 Hz which is a good resolution for practically all the chords. In Figure 3.2 there are different chords and their played notes. We can see how fundamental frequency separation notes are above 21.5 Hz.

The window type determines the hop size to choose. A maximum hope size has to be chosen to avoid aliasing, as described in [2]. As a consequence, for a Blackman window a maximum hope size of M/6 is required. In section 4 we will discuss about another restriction for the hope size.

3.2. Peak Detection

Several methods have been proposed in order to extract sinusoidal components by improving the frequency resolution of a quantized Fourier Transform (FT), called DFT. We can find 6 different methods to achieve it in [4]. VGuitarSustain uses the parabolic interpolation method because of its easy implementation, robustness and low error.

Given a window length M, the FFT size is $N = 2^{\lceil \log_2 M \rceil + 1}$, which results in spectrum oversampling by a minimum factor of 2, then with a peak separation of $4 f_s$, each main lobe peak will be sampled by at least 5 DFT bins. To aid the rejection of spurious peaks we may conservatively impose the restriction that the three magnitudes $|X(k_{n-1})|$, $|X(k_n)|$, $|X(k_{n+1})|$ of a

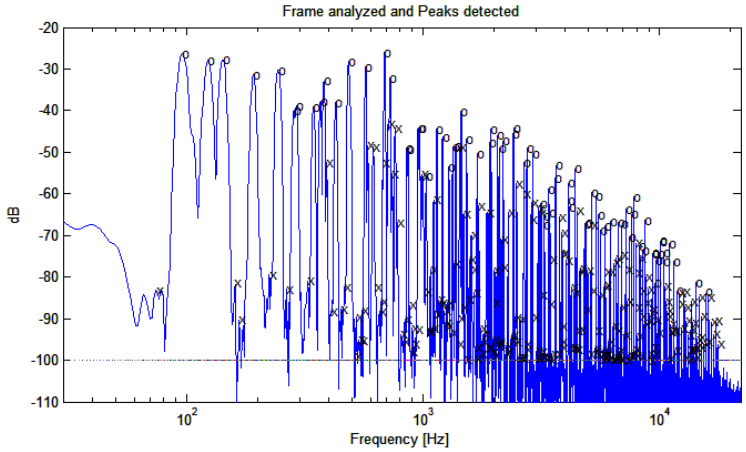

Figure 3.3: 'o' and 'x' peaks detected. 'x' peaks masked. 'o' peaks not masked, '-' lowest threshold detection.

parabolic peak must also all exceed the magnitude of either neighboring bin: $|X(k_{n-1})| > |X(k_{n-2})|$ or $|X(k_{n+1})| > |X(k_{n+2})|$. Once the program has all the possible quantized peaks, it only takes those which pass a threshold, and then compute the Parabolic Interpolation as described in [4] and [5] (section 10.3.3). After that, the program applies a partial pruning with all the peaks that are above the threshold to improve both the synthesis efficiency and the peak detection in order to eliminate spurious. This partial pruning uses the psychoacoustic phenomenon of masking that has been used for many years to remove redundant information from audio signals such as MPEG codecs. Only frequency masking is considered in VGuitarSustain. Using equation from [6] the program computes a frequency dependent Threshold for each peak found and then removes those which are under the threshold. In Figure 3.3 we can see all the peaks detected with the method described above and the masked peaks.

3.3. Partial Tracking

Once all the peaks from a STFT frame have been detected, the peaks must be organized so as to create continuous sinusoidal tracks, called partials. Several different techniques have been proposed for partial tracking. The simplest method is matching the peaks in the current frame with the nearest frequency peak in the previous STFT frames.

Peaks detected must be over a threshold to allow them to start a partial track. As the guitar spectrum decays in high frequency, we should equalize the signal to give equal weight to both high and low frequency components. However, there is an easy way explained in [7] where

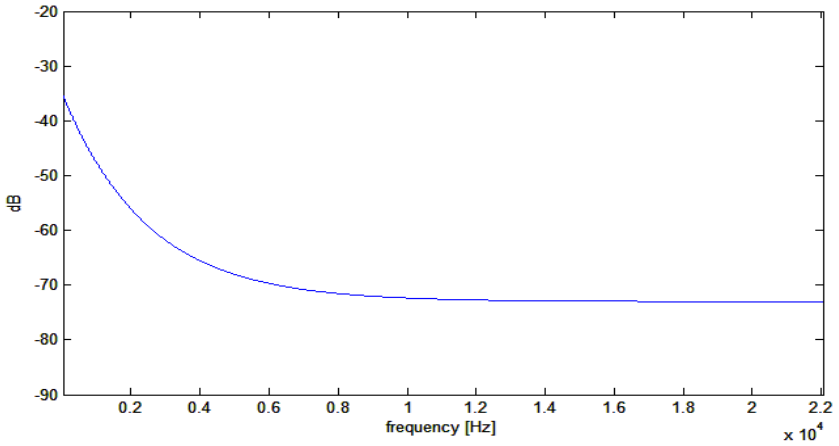

Figure 3.4: Threshold curve $A_t(f_p)$ with $b = 3{\cdot}10^{-4}$, $a_T = -35$,

they use a variable frequency dependent threshold, T_b. It is computed using a frequency dependent threshold curve, $A_t(f_p)$, in combination with the maximum bin amplitude of the analysis frame, $a_n^{\max}$. The threshold curve is given by

$$A_t(f_p) = a_T + a_L + \left(\frac{a_R}{b-1}\right) - \left(\frac{a_R}{b-1}\right) b^{f_P/20000} \tag{3.3}$$

where f_p is the frequency in hertz, b is a parameter controlling the shape of the curve, a_T is user a controllable threshold in dB, a_L is the amplitude offset at 0 kHz in dB, and a_R is the range of the curve in positive dB from 0 to 20 kHz. We can see in Figure 3.4 the threshold curve with the parameters adapted to values of the guitar sound.

Then T_b, given a maximum bin amplitude in dB for a frame, is computed as follows

$$T_b = a_n^{\max} + A_t(f_p) \tag{3.4}$$

If the peak is above T_b, a new partial track is considered. In consequence, a new structure is created for this track with the following information:

- Freq: frequency of the current track most recent peak
- Pw: Power in dB of the current track most recent peak
- Ph: Phase in rad of the current track most recent peak
- Ss: Start time of the current track to calculate decaying magnitude threshold
- Ts: Time of current track most recent peak

- Rs: Track number
- Ms: Current magnitude threshold limit

The tracking algorithm is:

```
for PreviousPeaks do
 [distance bestix] = min{|detectedPeaks.freq – previousPeak.freq|}
 if distance < freq_threshold then
   magdBStep = |previousPeak.Pw – detectedPeaks(bestix).Pw|
   if magdBStep > previouspeak.Ms then
     track not matched, remove detectedPeaks(bestix)
   else
     track matched, save new values.
     newPartial.Ts = time
   end if
 end if
 if (currentime - PreviousPeak.Ts) > deadSteps then
   remove
 end if
 for tracks not matched
   if detectedPeaks.Pw > Tb then
     create new track
     newPartial.Ts = time
   end if
 end for
end for
```

`PreviousPeaks` are the partials in the previous frames with all the structure information mentioned before. `DetectedPeaks` are the peaks detected in the current frame. The algorithm searches for the best matched `previousPeak` and it takes the distance and the index of the candidate (`bestix`). If the distance is close enough, it compares the power of the previous partial and the peak detected. If the step in amplitude is not too high, then this is the candidate for matching, otherwise the algorithm removes the peak found.

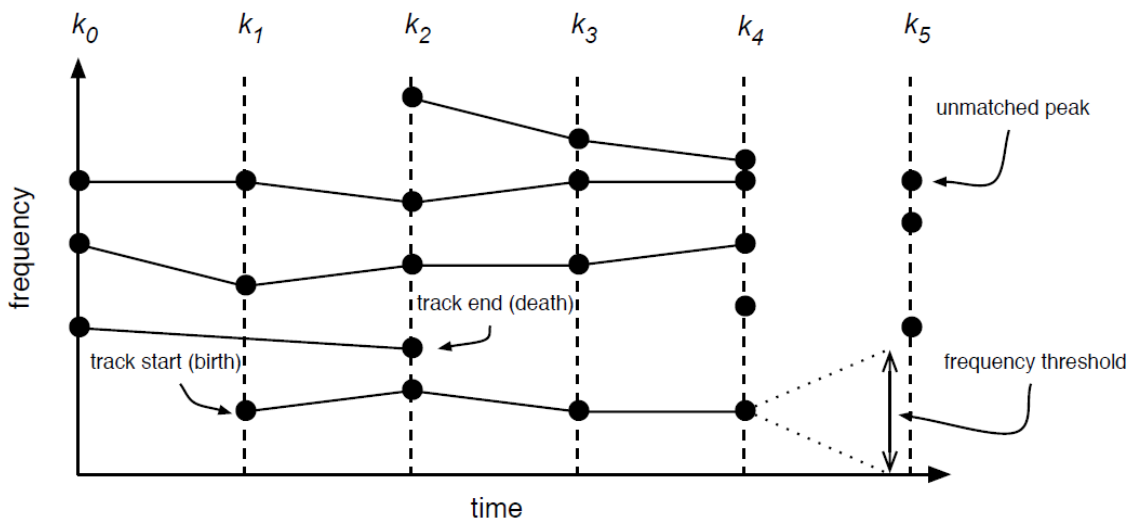

Figure 3.5: Connecting STFT peaks into sinusoidal tracks. Source: [7]

When a partial is not matched to the next frame, the algorithm leaves it as unknown, but if the number of consecutive unknown values exceeds `deadSteps`, the partial is considered as "death". In Figure 3.5 there is a graphical representation of the partials.

4. Partial Amplitude Modulation (PAM)

EQUATION SECTION (NEXT)Once one has all the present sinusoids in a chord one could synthesize it, but most of the chords are not stable in amplitude, apart from the decay, there is an amplitude fluctuation. In Figure 2.2 where there is the spectrum in 2D we could not appreciate this amplitude fluctuation in some partials. This amplitude modulation is called in literature as tremolo or amplitude vibrato. Normally, vibrato is usually used for a frequency variation whereas tremolo is usually used for an amplitude variation. We will refer to this tremolo as Partial Amplitude Modulation (PAM). Our purpose is to estimate this PAM.

In Figure 4.1 we can take a look at the evolution of amplitude partials during time. We can see different behaviour patterns in each chord. Some partials have higher slope, the more the frequency the higher the decay. This is natural in all the sounds because high frequencies normally have high absorption coefficients. Hence, we should take into account when synthesizing. Another thing to realize is that the PAM does not depend on the partial frequency, the high frequencies tend to have a higher frequency tremolo, but it is not possible to establish a clear pattern.

Besides this, it will be interesting to know why this PAM appears. It is caused by two effects. One of them is intrinsic in the instrument, and it is a characteristic of guitar sound. The chord is a linear combination of, at the most, 6 strings and each guitar string has its own behavior. We can observe this effect when we take a look in one string played, Figure 4.2, where one partial has this PAM.

Another cause of this tremolo in each partial, called PAM, is related to the sinusoidal fusion. When two sinusoids are close enough in frequency, the result sound can be interpreted as

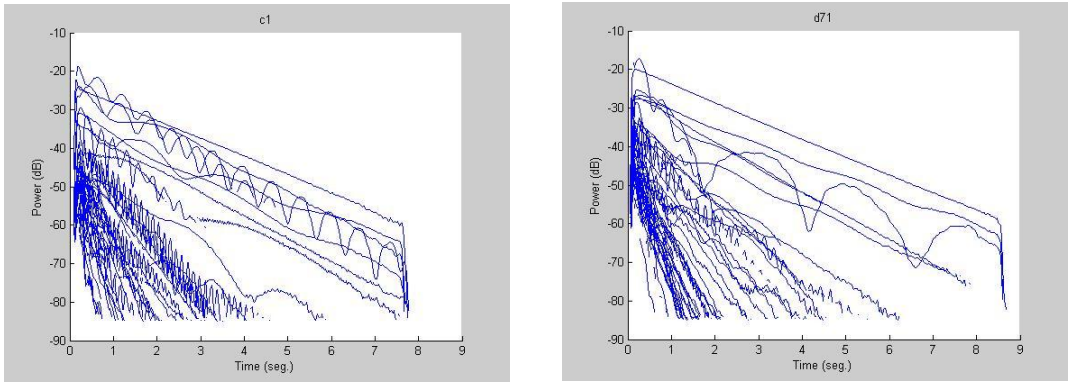

Figure 4.1: Partial Amplitude evolution. Left: C chord. Right D7 chord.

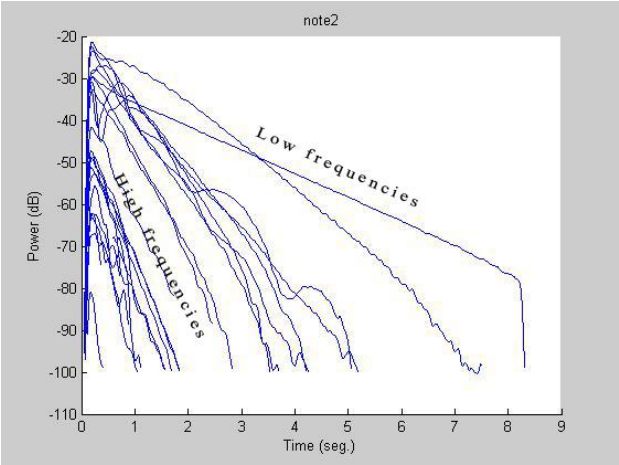

Figure 4.2: Partial Amplitude Evolution of one string

one sinusoid with an amplitude variation. The trigonometric identity (4.1) shows how a variation in frequency Δf becomes as an amplitude variation:

$$\sin(\omega \cdot n) + \sin\left(2\pi(f+\Delta f)\cdot n\right) = 2\cdot\sin\left(2\pi\frac{f\cdot n+(f+\Delta f)\cdot n}{2}\right)\cdot\cos\left(2\pi\frac{f\cdot n-(f+\Delta f)\cdot n}{2}\right) =$$

$$= 2\cdot\sin\left(2\pi\left(f+\frac{\Delta f}{2}\right)\cdot n\right)\cdot\cos\left(2\pi\frac{\Delta f}{2}n\right) \tag{4.1}$$

This sinusoidal fusion could not be suppressed because of the length of the window. We should have infinite length so that we would have a window with zero bandwidth. Moreover, the other effect is intrinsic to the sound, for this reason both effects are not suppressible and must be synthesized in order to have a real sound. The partial evolution can be modelled as:

$$P_i(sH) = A_{0i}e^{-\alpha_i sH}(1+m_i(sH)) \tag{4.2}$$

where A_i is the amplitude of the partial, α_i is the constant decay, m_i is the function of modulation (PAM), H is the hop size and s is the time variable. We are taking a sample of the guitar sound each $H\cdot T_s$, therefore, the frequency sampling of STFT is

$$f_{STFT} = \frac{f_s}{H} \tag{4.3}$$

The frequency resolution, as described in section 3.1, is 21.5 Hz; then each two sinusoids with less frequency difference than 21.5 Hz will become one sinusoid with time amplitude variation. This effect of merging of two sinusoids seen as one is called sinusoidal fusion. Taking a look in

(4.1), knowing that the largest deviation in frequency is $\Delta f = 21{,}5Hz$, we can extract that the Amplitude Modulation frequency will be 10.75 Hz, however one partial could have a higher frequency modulation due to its sound. Anyway, we are going to take the maximum merging frequency, $\Delta f/2$, as the more restrictive condition. So then, applying the Niquist criteria, and equation (4.3)

$$f_{STFT} > 2B \rightarrow \frac{f_s}{H} > 2B \tag{4.4}$$

Taking equation (3.2), where f_r is $2B$, then

$$H \leq \frac{M}{\Delta\omega} \tag{4.5}$$

Remembering the condition taken in section 3.1 which is $H < M/6$, watching the previous equation, where $\Delta\omega$ is the desired separation in bins of two windows and knowing that the main lobe width of a Blackman window is 6; we can extract that $H < M/6$ is the restrictive equation because you will always choose $\Delta\omega$ less than the main lobe width because of it is possible to detect two nearby components with less than the main lobe width and you will even have less window length.

Back to estimate the PAM, we could do an exponential regression to extract the modulation $m_i(sH)$ from (4.2) but in the exponential regression algorithm there is, first, a logarithm conversion and then a linear regression. Therefore we are going to convert it in a linear expression taking the logarithm of (4.2):

$$P_{idB}(sH) = \log_{10} P_i(sH) = \log_{10} A_{0i} - \alpha_i \log_{10}(e) sH + \log_{10}\left(1 + m_i\left(sH\right)\right) \tag{4.6}$$

Now it is possible to estimate all the parameters, once VGuitarSustain has the logarithm of the amplitude partial evolution. $P_{idB}(sH)$ is a line, so it is possible to estimate the amplitude A_{0i} and the α_i doing a linear regression ($LR(sH) = \hat{a} \cdot sH + \hat{b}$). With this parameters estimated, the program can remove $\log_{10} A_{0i} - \alpha_i \log_{10}(e) sH$ from (4.6) and leave a signal with only the PAM and the error:

$$\log_{10} \hat{x}_i(sH) = \log_{10}(1 + m_i(sH)) + \log_{10}(\varepsilon_i(sH)) \tag{4.7}$$

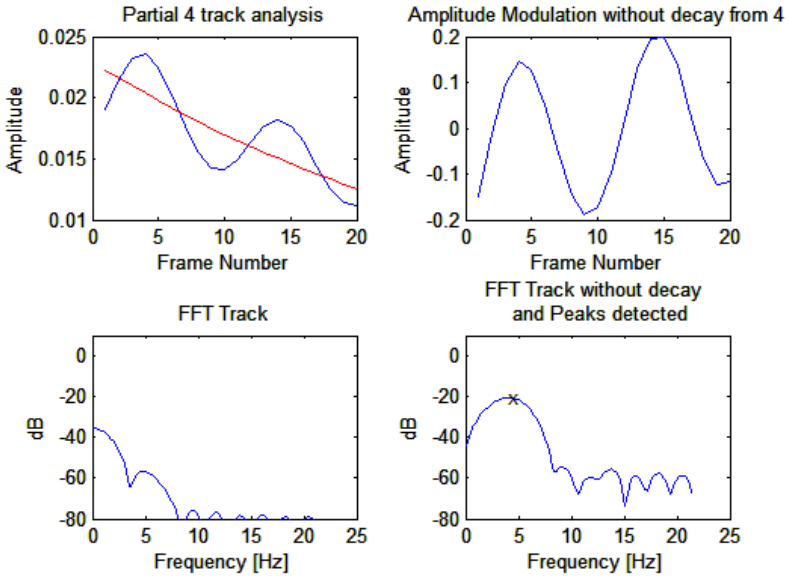

Figure 4.3: Partial Track Analysis

The synthesized chord might have different PAM in each partial, compared with the real chord, cause of the error; however it will sound as the same chord, with the same tone. Thus, we can allow some error $\varepsilon_i(sH)$.

Taking out the logarithm from (4.7) it obtains a signal with the modulation multiplied by the error. Considering that $\log_{10}(\varepsilon(sH)) \to 0$ then VGuitarSustain can extract the PAM

$$\hat{x}(sH) = (1 + m_i(sH)) \cdot \varepsilon(sH) \approx 1 + m_i(sH) \tag{4.8}$$

Then, VGuitarSustain does the FFT of m_i and takes the most significant peaks using peak detection and threshold. Therefore, the modulation is computed as a sum of sinusoids.

On top left of Figure 4.3 we can see the partial amplitude evolution (equation(4.2)). The red line is de linear interpolation. Below we can see the FTT of the top left graphic. On top right, we can see the modulation, $m_i(sH)$ once the linear regression is computed and extracted from the partial. The bottom right graph is the FFT of top right. The cross is the peak detected. VGuitarSustain has a threshold of -45 dB for peak detection of amplitude modulation to remove those peaks that represent insignificant amplitude modulation. Thus, any amplitude in $m_i(sH)$ less than 0.01 is taken as no variation in amplitude.

The number of samples of PAM to compute the FFT depends on the time one presses the pedal. To understand this, we are going to explain how VGuitarSustain works.

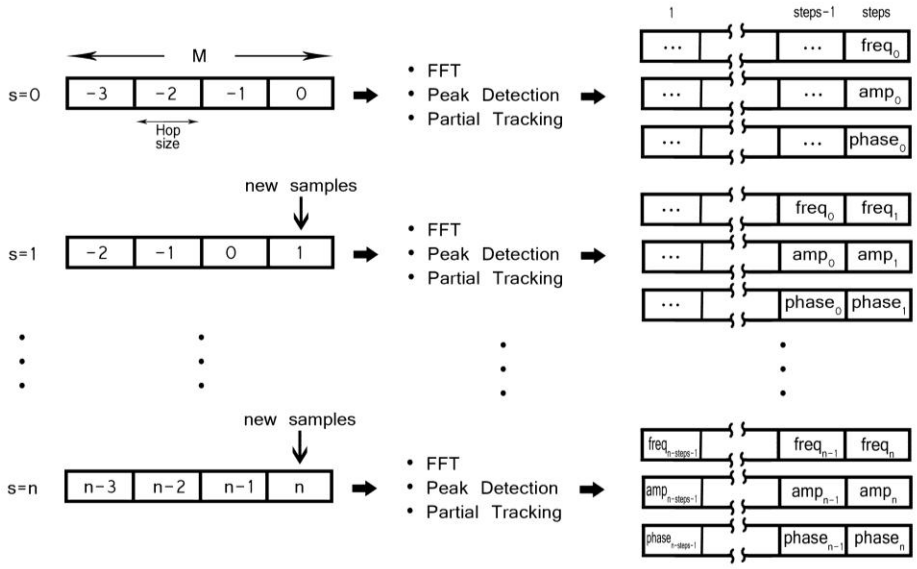

Figure 4.4: Buffer Iteration in VGuitarSustain

There is an input buffer with window size "M". Every time that the program has H samples, the hope size, the input buffer is shifted to the left and new samples are added to the buffer at input buffer right part. Then, zero-padding is added and FFT computed. After peak detection and partial tracking, those partials that are matched, its amplitude, frequency and phase are added at tracking buffers column "steps". These tracking buffers are three matrix (amplitude, frequency and phase) with as many rows as partials are being tracked, and "steps" columns. Each row of this amplitude matrix, $P_i(sH)$, is used to calculate the PAM.

Going back to the FFT number to estimate the frequency, amplitude and phase modulation, as we said, it will depend on the time the guitarist presses the pedal. Let's suppose the guitarist plays a chord on time 0 and then he presses the pedal after t_1 ms. Then, the number of samples of PAM would be $t_1 \cdot f_{STFT}$. For example, if the player presses the pedal after 400 ms, we would have 18 samples to compute the FFT. Executing the program with different values we realize that a good number of samples are more than 16 making the synthesized chord better with 20. The more samples the lower the error in the linear prediction and then, better estimation. But, what happens if the player presses the pedal before 400 ms? In Figure 4.5 we can see a wrong PAM detection. On top left the attack (crescendo part) of the chord appears. On maximum of the partial amplitude, the sustain part starts. It becomes in a bad PAM estimation due to the attack part, so the program detects an amplitude modulation which really there is not.

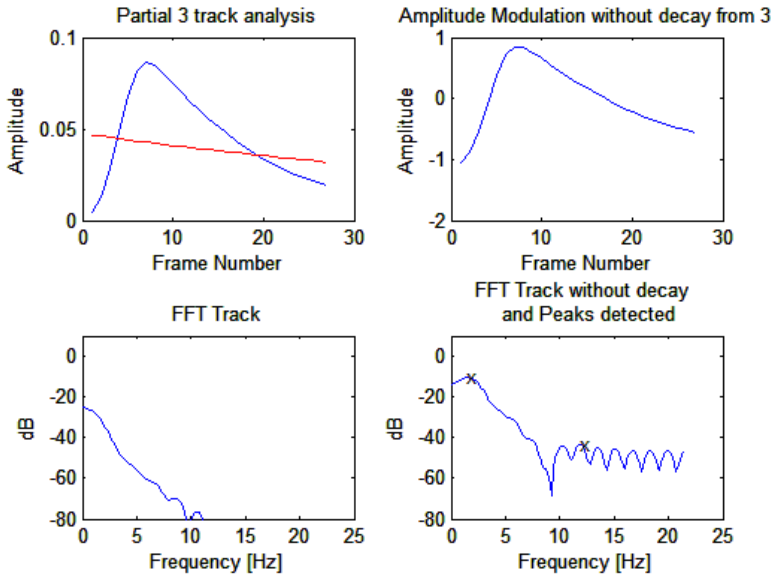

Figure 4.5: Bad PAM detection

To apply a good linear prediction we have to avoid the transient of the chord, the att6ack part, to estimate a good linear prediction. This is achieved with onset detection.

4.1. Onset Detection

The onset detection is important to avoid a bad estimation in PAM whenever the guitarist pushes the pedal for a chord synthesis. There are two ways, a part from others, to detect this onset detection:

- Energy Information
- Phase Information

The first one uses the energy difference between consecutive spectra. Hence, an introduction of a new chord leads to an increase in the energy of the signal. The other concept is based on the Phase-Vocoder; during the steady-state, part of the signal of these components will tend to have constant frequencies. Therefore, the difference between two consecutive unwrapped phase values must remain constant between frames. A complex domain method explained in [8] is used in VGuitarSustain which improves over previous energy-based and phase-based approaches by combining both types of information in the complex domain. In Figure 4.6 we can see the Detection Function and a threshold with a hysteresis for chord detection. The hysteresis is added to avoid wrong detections because of the small oscillations on Detection Function.

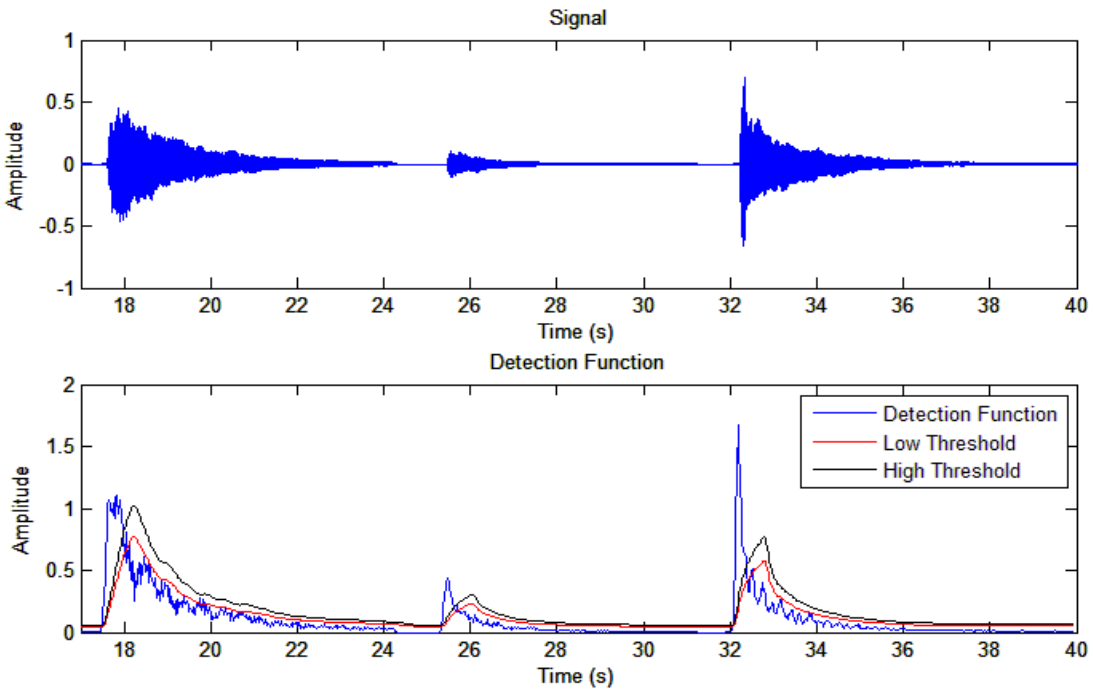

Figure 4.6: On top, signal to analyse. On bottom, the Detection Function and the thresholds for a chord detection

4.2. State Zones

We can remove this transient part that led VGuitarSustain to a bad PAM detection knowing where the chord starts. Then when a chord is detected a number of frames are removed to avoid this wrong detection. VGuitarSustain has a variable called `transSteps` to avoid the number of transient steps. The default value is 10 frames. However, we have seen in the previous section that for PAM's estimation we need 20 frames. Consequently, a total of 30 frames are needed in order to synthesize the chord with the estimated PAM. In Figure 4.7 we can understand what happens whenever a chord is detected. When the detection function passes the high threshold, a counter called `sustain` is set to zero. On the attack part (from 1381 to 1390), no partial tracking is applied, therefore no samples are added to the track buffer. If the guitarist presses the pedal in this zone, VGuitarSustain waits until it arrives in the "Random Modulation".

When the counter arrives in the "Random Modulation" section the program stars to do the tracking partial. If the guitarist presses the pedal in this zone or the zone before, as no PAM can be estimated, the program does the peak detection with one frame, takes the most significant peaks and computes a random modulation with a decay that follows the following expression:

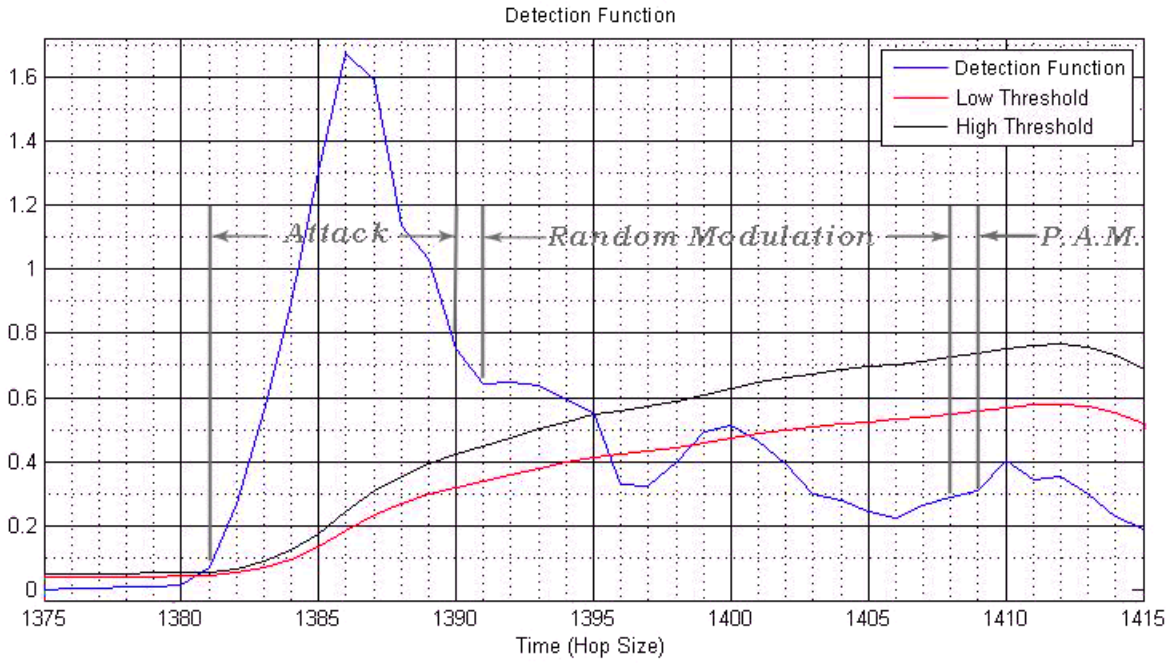

Figure 4.7: Detection Function. States of the program whenever a chord is detected

$$\tau(f) = De^{-(30-e^{|x|})f} \tag{4.9}$$

where D is the decay constant, f is the frequency in Hertz and x is a normal distribution $x \sim N(0,1)$. This random variable adds more interest in the sound and emulates the real sound. Each string has certain decay in each harmonic, and this is related to the frequency but also to the harmonic. So then a higher frequency could have less decay than another with less frequency.

If the guitarist presses the pedal in the PAM zone, the chord can be synthesized with the modulation. The synthesis is done by:

$$y(n) = \sum_{k=1}^{K} e^{-\tau_k n}(1 + A_k^m \cos(\omega_k^m n + \varphi_m^m))A_k \cos(\omega_k n + \varphi_k) \tag{4.10}$$

where K are the number of sinusoids detected, τ_k the decay constant generated, and the respective amplitude, frequency and phase of both, the modulation and the sinusoid. . Clearly we can see three different parts, the decay part, the modulation part, and the sinusoid generated.

5. Conclusion and Future Work

We can see in this work how it is possible to synthesize a chord in real time. However, we can only do it with a latency of 23 ms which is not really a good latency for playing in real time. The objective is to reduce the latency to less than 10 ms but this can only be improved with a high or mid-level programming language. Now, with 1024 buffer samples, which implies these 23 ms of latency, we have some glitches (samples that are lost) due to some function, like the function `interp1`, which takes a long time to compute the interpolation. The next step should be to program the code in a high or mid-level programming language like C or C++ and optimize it.

As we have said, `interp1` takes long time to compute whenever it has to interpolate unknown values. A possible solution should be to introduce a new partial tracking method based on linear prediction of the frequency and amplitude evolutions of the partials as [9]. In this case, the call of the function `interp1` could be removed.

Listening to the chord generated, we realise that the faster you press the pedal the higher frequencies appear in the sound. This is a consequence of the guitar sound. If we take a look into Figure 2.2 we can see how the onset of the chord the high frequencies are more present than in the middle of the spectrogram. Then as we said, the program computes the synthetic chord based on its spectrum when we are in the random part. So then a better equation than (4.9) could be applied in order to create a decay function that makes the synthetic chord more natural.

Another improvement to do that would lead us to a more efficient method. On the other hand, it would generate more accurate sinusoidal parameters (amplitude, frequency, and phase). This would be achieved by adding a Multiresolution Sinusoidal Modelling [10] that basically separates the spectrum in different frequency ranges and then computes the Sinusoidal Modelling. With this method we could reduce the window length M, improving the computational efficiency and we would have more accurate sinusoidal parameters. We must remember that now we have a window length of 8191 which is necessary to detect the peaks in the low frequency range, but for a high frequency it is not necessary, less length is enough to detect the frequency. The lower the frequency the higher the period, then for a low frequency we normally need a lot of samples to have, at least, one period of the sinusoid, whereas for a high frequency, the number of samples to sample a period are far fewer compared with a low frequency.

A lot of work is still open to finish this Virtual Sustain Pedalling for Guitar by Sinusoidal Modelling, we should optimize the code generated on Matlab (Appendix A) to be compiled in a high mid-level language. Then we should find a way to emulate this pedal work with the program and make an interface to be able to change the more important settings, like the sustain decay, the release, the number of peaks detected, and so on.

I encourage all who wish to continue the work already done.

Bibliography

[1] R. E. Crochiere, "A Weighted Overlap-Add Method of Short-Time Fourier Analysis/Synthesis," *Acoustics, Speech and Signal Processing, IEEE Transactions on,* vol. 28, no. 1, Feb 1980.

[2] J. Allen and L. Rabiner, "A unified approach to short-time Fourier analysis and synthesis," *Proceedings of the IEEE,* vol. 65, no. 11, Nov. 1977.

[3] Mototsugu Abe, and Julius O. Smith, "Design Criteria for Simple Sinusoidal Parameter Estimation based on Quadratic Interpolation of FFT Magnitude Peaks," in *117th Convention*, San Francisco, 28-31 Oct. 2004.

[4] F. Keiler, S. Marchand, "Survey on Extraction of Sinusoids in Stationary Sounds," in *Digital Audio Effects (DAFx-02)*, Hamburg, Jul 2008.

[5] U. Zölzer, DAFX: Digital Audio Effects, 2nd ed., Hamburg: A John Wiley and Sons, Ltd., Publication, 2011.

[6] G. Marentakis, K. Jensen, "Sinusoidal Synthesis Optimization," Copenhagen, 2002.

[7] M. Kateley, Spectral Analysis, Editing, and Resynthesis: Methods and Applications, Columbia University, 2009.

[8] C. Duxbury, J. P. Bello, M. Davies and M. Sandler, "Complex Domain Onset Detection For Musical Signal," in *Digital Audio Effects (DAFx-03)*, London, 2003.

[9] M. Lagrange, S. Marchand, M. Raspaud and J.B. Rault, "Enhanced Partial Tracking Using Linear Prediction," in *Digital Audio Effects*, London, 2003.

[10] T. S. V. J. O. S. I. Scott N. Levine, "Multiresolution Sinusoidal Modeling for Wideband Audio with Modifications," Center for Computer Research in Music and Acoustics, Standford, 1998.

Appendix A - Program Code

This Matlab code is made in Matlab Version R2010a. To use it you must install the utility Playrec, available in http://playrec.co.uk

```
function pedal()
%----------------------------------
% Pedal v 1.0
% Date: 17.01.2012
% Author: Joan Prat
%----------------------------------

clear all
close all

%% --- Initialization of variables ---
global Fs
global NFFT H N2
global verbose
global steps La
global ntrax
global maxPeaks fc MAFS maxPeaksMod
global Thres
global freqTracks ampTracks phaseTracks
global pageSize
global countSynth countRelease
global FAM  AM  PAM
global iftfreq iftamp iftphase
global Tau Lambda
global phaseFFTInput1 phaseFFTInput2
global magn1 magn
global chord sustain


%=== USER DATA ===
playDeviceID  = 9;      % Playrec parameter. http://www.playrec.co.uk/
playChanList  = [1 2];  % Playrec parameter. http://www.playrec.co.uk/
recDeviceID   = 10;     % Playrec parameter. http://www.playrec.co.uk/
recChan       = 1;      % Playrec parameter. http://www.playrec.co.uk/
Fs            = 44100;  % Sampling Frequency
pageSize      = 1024;   % size of each page processed (H = pageSize)
M             = 8191;   % Length of Analysis Window
steps         = 35;     % column dimension of tracks matrix
La            = 20;     % minimum values of amp to estimate parameters
transSteps    = 10;     % Number of STFT Samples to avoid after onset detection
maxPeaks      = 70;     % maximum peaks detected
Thres         = -100;   % Start Amplitude threshold for peak detection [dB]
TAdaptFactor  = 0.08;   % Thres Constant Adaption for Peak Threshold
ft            = 21500;   % frequency cutoff for Noise Calculation
fc            = 70;     % frequency high-pass cutoff [Hz]
MAFS          = 3.45;   % smallest separation allowed for peak
% detection and does not introduce significant bias in the peak
```

```
function pedal


% interpolation due to interference from a neighboring peak

% HCompThres & LCompThres are Threshold Histeresis for Chord Detection
HMinThres     = 0.005;  % High Threshold Chord Detection
LMinThres     = 0.004;  % Low  Threshold Chord Detection
HAdaptTFactor = 1.2;    % High Thres. Constant adaptation
LAdaptTFactor = 0.9;    % Low  Thres. Constant adaptation
Release       = 500;    % Release in miliseconds
RateSustain   = 4;      % Sustain Factor 1=> short sustain 10=> large sustain
PedalSwitch   = 512;    % Mouse 'X' Pixel Value to Switch On/Off the pedal
PedalOff      = 750;    % Mouse 'Y' Pixel Value to Stop VGuitarSustain
runMaxSpeed   = true;   % When true, the processor is used much more heavily
%(ie always at maximum), but the chance of skipping is reduced


%=== Definition of constants
NFFT        = 2^floor(log2(M)+1);
H           = pageSize;
maxPeaksMod = 1;
NFade       = min(1000,pageSize);
verbose     = 0;
N2          = NFFT/2;
window      = blackman(M);
window      = window'/sum(window);
countSynth  = -1;    % counter to follow the phase during the synthesize
inputBuffer = zeros(1,M);
nextPageSamples = zeros(1,pageSize);
phaseFFTInput1  = zeros(N2,1);
phaseFFTInput2  = zeros(N2,1);
magn        = zeros(N2,1);
magn1       = zeros(N2,1);
sustain     = 0;
lambdaMin   = log(100)/(Release*0.001*Fs);
cRelease    = ceil(0.001*Release*Fs/H);
HCompThres  = HMinThres;
LCompThres  = LMinThres;
compTrans   = zeros(1,steps);
ftBin       = round(ft/Fs*NFFT);
restart_track_values();
restart_synth_values()

%% ---  Audio Configuration ---
%Test if current initialisation is ok
if(playrec('isInitialised'))
  if(playrec('getSampleRate')~=Fs)
    fprintf('Changing playrec sample rate from %d to %d\n',...
      playrec('getSampleRate'), Fs);
    playrec('reset');
  elseif(playrec('getPlayDevice')~=playDeviceID)
    fprintf('Changing playrec play device from %d to %d\n',...
      playrec('getPlayDevice'), playDeviceID);
    playrec('reset');
  elseif(playrec('getRecDevice')~=recDeviceID)
    fprintf('Changing playrec record device from %d to %d\n',...
      playrec('getRecDevice'), recDeviceID);
```

```
    playrec('reset');
  elseif(playrec('getPlayMaxChannel')<max(playChanList))
    fprintf('Resetting playrec to configure device to use more output channels\n');
    playrec('reset');
  elseif(playrec('getRecMaxChannel')<recChan)
    fprintf('Resetting playrec to configure device to use more input channels\n');
    playrec('reset');
  end
end

%Initialise if not initialised
if(~playrec('isInitialised'))
  fprintf('Initialising playrec to use sample rate: %d, playDeviceID: %d and
recDeviceID: %d\n', Fs, playDeviceID, recDeviceID);
  playrec('init', Fs, playDeviceID, recDeviceID, max(playChanList),...
    max(recChan),1024);

  % This slight delay is included because if a dialog box pops up during
  % initialisation (eg MOTU telling you there are no MOTU devices
  % attached) then without the delay Ctrl+C to stop playback sometimes
  % doesn't work.
  pause(0.1);
end

if(~playrec('isInitialised'))
  error ('Unable to initialise playrec correctly');
end

if(playrec('pause'))
  fprintf('Playrec was paused - clearing all previous pages and unpausing.\n');
  playrec('delPage');
  playrec('pause', 0);
end
repeatCount = 1;
point = get(0,'PointerLocation');
%% Program execution
while point(2)< PedalOff

  %% Playrec interface
  pageNum = playrec('playrec', repmat(nextPageSamples',1,...
    length(playChanList)), playChanList, pageSize, recChan);

  if(repeatCount==1)
    %This is the first time through so reset the skipped sample count
    playrec('resetSkippedSampleCount');
    repeatCount = 2;
  end
  %playrec('getSkippedSampleCount')
  % runMaxSpeed==true means a very tight while loop is entered until the
  % page has completed whereas when runMaxSpeed==false the 'block'
  % command in playrec is used.  This repeatedly suspends the thread
  % until the page has completed, meaning the time between page
  % completing and the 'block' command returning can be much longer than
  % that with the tight while loop
  if(runMaxSpeed)
    while(playrec('isFinished', pageNum) == 0)
```

```
  end
else
  playrec('block', pageNum);
end


nextPageSamples = double(playrec('getRec',pageNum));
nextPageSamples = nextPageSamples';
playrec('delPage',pageNum);
%% Analysing
% shift the buffer
inputBuffer = circshift(inputBuffer,[0 -pageSize]);
inputBuffer(M-pageSize+1:end) = nextPageSamples;
inputWindowed = inputBuffer(1:M).*window;
fftInput = fft([inputWindowed, zeros(1,NFFT - M)]',NFFT);
logAbsFFTInput = 20*log10(abs(fftInput(1:N2)));
phaseFFTInput = angle(fftInput(1:N2));
%Thres = Thres + TAdaptFactor*(mean(logAbsFFTInput(ftBin:N2))-Thres);


if (countSynth == -1) || (countRelease > 0) % Do no process during synthesis
  % Phase Desviation
  phaseDev = (princarg(phaseFFTInput - 2*phaseFFTInput1 + phaseFFTInput2));
  % Complex Transcient+Energy Detection
  magn = abs(real(fftInput(1:N2)));
  func = sqrt(magn1.^2+magn.^2-2*magn1.*magn.*cos(phaseDev));
  compTrans = circshift(compTrans,[0 -1]);
  compTrans(1,steps) = sum(func);
  % Histeresis for chord detection
  if (compTrans(1,steps) > HCompThres)
    if chord == 0
      chord = 1;
      sustain = 0;
    end
  elseif (compTrans(1,steps) < LCompThres)
    if chord == 1
      chord = 0;
    end
  end
  sustain = sustain + 1;
  HCompThres =  HMinThres + HAdaptTFactor*mean(compTrans);
  LCompThres =  LMinThres + LAdaptTFactor*mean(compTrans);

  % Saving values for next iteration
  phaseFFTInput2 = phaseFFTInput1;
  phaseFFTInput1 = phaseFFTInput;
  magn1 = magn;

  if sustain >= transSteps
    move_tracks()
    extract_tracks(logAbsFFTInput,phaseFFTInput);
  end
end
point = get(0,'PointerLocation');
%% Algorithm of the program
if (point(1) < PedalSwitch)
```

```
    if countRelease > 0
      restart_synth_values()
    end
    if (countSynth == -1)
      if (sustain >= La+transSteps)
        % extract modulation parameters
        extract_modulation();
        iftamp = 10.^((iftamp+6)/20);
        Tau = RateSustain*100000*exp(-20/Fs*iftfreq);  % Decay of the partials
        Lambda = lambdaMin*exp(80/Fs*iftfreq);  % Decay for Release
        iftfreq = iftfreq*2*pi/Fs;
        AM = 10.^((AM+6)/20);
        FAM = FAM*2*pi/Fs;
        fprintf(1, ['PAM  - Number of Synthesized sinusoids: %d\n'], length(iftamp))
        countSynth = countSynth + 1;
      else
        [maxBins maxPw maxPh] = pick_peaks18_12(logAbsFFTInput,phaseFFTInput,...
          Fs,Thres,MAFS, round(fc/Fs*NFFT),maxPeaks);
        iftfreq = (maxBins-1)/NFFT*2*pi;
        iftamp = 10.^((maxPw+6)/20);
        iftphase = maxPh+iftfreq*H;
        ntrax = size(maxBins,1);
        if ntrax > 0
          FAM       = zeros(ntrax,maxPeaksMod);
          AM        = zeros(ntrax,maxPeaksMod);
          PAM       = zeros(ntrax,maxPeaksMod);
          FAM(:,1)  = abs(random('norm',0,2,ntrax,1))*2*pi/Fs;
          AM(:,1)   = abs(random('norm',0,0.5,ntrax,1));
          AM = AM .* (AM < 0.3);
          PAM       = random('unif',0,2*pi,ntrax,1);
          randDec = 30-exp(abs(random('norm',0,1, [ntrax 1])));
          Tau = RateSustain*100000*exp(-randDec/2/pi.*iftfreq);  % Decay of the
partials
          Lambda = lambdaMin*exp(80/2/pi*iftfreq);  % Decay for Release
        end
        fprintf(1, ['Rand - Number of Synthesized sinusoids: %d\n'], ntrax)
        countSynth = countSynth + 1;
      end
    else
      nextPageSamplesSynth = synthesize();
      % add fade-in to avoid clips
      if countSynth == 0
        nextPageSamplesSynth(1:NFade) = nextPageSamplesSynth(1:NFade).*...
          linspace(0,1,NFade);
      end
      %synthesize
      countSynth = countSynth + 1;      % counter to follow the phase
      nextPageSamples = nextPageSamplesSynth + nextPageSamples;

    end
  else
    if countSynth ~= -1
      % release time
      if countRelease == 0
        restart_track_values();
      end
      nextPageSamplesSynth = synth_release();
```

```
      countRelease = countRelease + 1;
      countSynth = countSynth + 1;
      nextPageSamples = nextPageSamplesSynth + nextPageSamples;
      if countRelease >= cRelease
        restart_synth_values()
      end
    end
  end
end


fprintf('Loop back complete with %d samples worth of glitches\n',...
  playrec('getSkippedSampleCount'));

%delete all pages now loop has finished
playrec('delPage');
playrec('reset');

end
```

function restart_synth_values

```
function restart_synth_values()
global FAM  AM  PAM
global iftfreq iftamp iftphase
global N2 c maxPeaksMod
global phaseFFTInput1 phaseFFTInput2
global countSynth countRelease
global chord sustain

FAM       = zeros(0,maxPeaksMod);
AM        = zeros(0,maxPeaksMod);
PAM       = zeros(0,maxPeaksMod);
iftfreq   = ones(0,0);
iftamp    = ones(0,0);
iftphase  = ones(0,0);
countSynth = -1;
countRelease = 0;
c         = 0;
sustain   = 0;
chord     = 0;
phaseFFTInput1 = zeros(N2,1);
phaseFFTInput2 = zeros(N2,1);
end
```

function restart_synth_values

```
function phase = princarg(phase_in)
% This function puts an arbitrary phase value into ]-pi,pi] [rad]
%
%------------------------------------------------------------------------
% This source code is provided without any warranties as published in
% DAFX book 2nd edition, copyright Wiley & Sons 2011, available at
% http://www.dafx.de. It may be used for educational purposes and not
% for commercial applications without further permission.
%------------------------------------------------------------------------

phase = mod(phase_in+pi,-2*pi) + pi;
```

function restart_track_values

```
function restart_track_values()
global Bins Pw Ph Ts Ms
global freqTracks ampTracks phaseTracks
global ntrax

Bins        = ones(0,0);
Pw          = ones(0,0);
Ph          = ones(0,0);
Ts          = ones(0,0);
Ms          = ones(0,0);
freqTracks  = ones(0,0);
ampTracks   = ones(0,0);
phaseTracks = ones(0,0);
ntrax       = 0;
end
```

function move_tracks

```
function move_tracks()
global freqTracks ampTracks phaseTracks

% Move Track
if size(freqTracks,1) > 0
  freqTracks(:,1) = NaN;
  ampTracks(:,1) = NaN;
  phaseTracks(:,1) = NaN;
  freqTracks = circshift(freqTracks,[0 -1]);
  ampTracks = circshift(ampTracks,[0 -1]);
  phaseTracks = circshift(phaseTracks,[0 -1]);
end
end
```

```
function extract_tracks(mXp,pXp)

global Fs
global NFFT
global ntrax
global c fc MAFS
global steps
global maxPeaks
global Thres
% 'frequencies' (i.e. bin indexes) of current track's most recent peaks
global Bins
% 'energies' (i.e. values) of current track's most recent peaks
global Pw
% 'phase'
global Ph
% 'times' (column indices) of current track's most recent peaks
% this can be less than the previous column because of permitted bridging
global Ts
% Current magnitude threshold limit.  This adapts
global Ms
% initialize the data array
global freqTracks
% Also follow our extracted magnitudes, for yuks
global ampTracks
% Debug parameters the same size
global phaseTracks

% Maximum col-to-col tolerable peak movement (in bins)
maxdbdt = 1;   % used to be 1.0

% How many steps before say a peak is finished
deadsteps = 8;

% Maximum mag-increase before new track (dB/step)
maxdBincr = 30.0;

% Asymptote of adaptive magnitude-increase threshold level (dB/step)
finaldBincr = 6; % was 6.0

% Adptv mag thresh stays this far above allowed positive slopes (must be > 1.0)
dbIncrSafeRatio = 4;

% Decay time for adptv mag incr threshold (steps)
dbIncrTsteps = 10;

% Parameters for absolute threshold
At = -35;
Al = 0;
Ar = 38;
b  = 0.0003;

%--- detect peaks & caluclate inerpolated values (peak position, phase,
%amplitude)
[maxBins maxPw maxPh] = ...
  pick_peaks18_12(mXp,pXp,Fs,Thres,MAFS, round(fc/Fs*NFFT),maxPeaks);
```

```
c = c+1;
% Match up to existing tracks
numcurrent = size(Bins, 1);
% Setup mask to indicate which tracks should be terminated this time
allowCont = ones(1,numcurrent);
 for i = 1:numcurrent
  %fprintf(1, 'col %d: extending current#%d (r=%d) f=%f\n', c, i, Rs(i), Bins(i));
  b = Bins(i);
  foundcont = 0;
  % Unfortunately, when maxBins is empty, this test crashes, so have to
  % do it as 2 nested tests.
  %if size(maxBins,2) > 0 & abs(f - maxBins(bestix)) < maxdfdt
  if size(maxBins,1) > 0
    dbs = abs(maxBins - b);
    [dbsmin bestix] = min(dbs);
    if size(maxBins,1) > 0 && dbsmin < maxdbdt
      % Found a continuation for this current peak
      % Check that it doesn't fail the magnitude increase check
      magdBStep = abs(maxPw(bestix)-Pw(i));
      if magdBStep > Ms(i)
        % Step was too big: Kill this one & leave the continuation unused
        % so that a new track is created with it
        allowCont(i) = 0;
      else
        Bins(i) = maxBins(bestix);
        Pw(i)   = maxPw(bestix);
        Ph(i)   = maxPh(bestix);
        Ts(i)   = c;
        % delete the taken peak of maxBins & maxPw
        mask = (1:size(maxBins,1)) ~= bestix;
        maxBins = maxBins(mask);
        maxPw = maxPw(mask);
        maxPh = maxPh(mask);
        % Store 'freq' in output array
        freqTracks(i, steps) = (Bins(i)-1)*Fs/NFFT;
        ampTracks(i, steps) = Pw(i);
        phaseTracks(i, steps) = Ph(i);
        foundcont = 1;
        % Update the mag threshold
        Ms(i) = max(dbIncrSafeRatio*magdBStep, ...
          Ms(i) - (Ms(i)-finaldBincr)*(1-exp(-1/dbIncrTsteps)));
      end
    end
  end

  if foundcont == 0
    % No acceptable continuation found - maybe kill this track
    if (((c - Ts(i)) > deadsteps)) %|| (col(round(Bins(i))) < lowEthresh*Es(i)))
      % Mark this track to be removed from current structures
      allowCont(i) = 0;
    end
  end
end % of loop over currently active tracks

% Remove the tracks that have ended from the current records
```

```
if min(allowCont) == 0
  tracksToKeep = find(allowCont);
  Bins = Bins(tracksToKeep);
  Pw = Pw(tracksToKeep);
  Ts = Ts(tracksToKeep);
  Ms = Ms(tracksToKeep);
  Ph = Ph(tracksToKeep);
  freqTracks = freqTracks(tracksToKeep,:);
  ampTracks = ampTracks(tracksToKeep,:);
  phaseTracks = phaseTracks(tracksToKeep,:);
  ntrax = length(tracksToKeep);
end

% Create new tracks for all remaining peaks
freq = (maxBins-1)*Fs/NFFT;
startupThresh = max(mXp)+At+Al+(Ar/(b-1))-(Ar/(b-1))*b.^(freq/20000);
for i = 1:size(maxBins,1);
  if maxPw(i) > startupThresh(i)
    freqTracks = [freqTracks; -ones(1, steps)];
    ampTracks = [ampTracks; -ones(1, steps)];
    phaseTracks = [phaseTracks; NaN*ones(1, steps)];
    ntrax = ntrax + 1;
    Bins = [Bins; maxBins(i)];
    Pw = [Pw; maxPw(i)];
    Ts = [Ts; c];
    Ms = [Ms; maxdBincr];
    Ph = [Ph; maxPh(i)];
    freqTracks(ntrax, steps) = (Bins(ntrax)-1)*Fs/NFFT;
    ampTracks(ntrax, steps) = Pw(ntrax);
    phaseTracks(ntrax, steps) = Ph(ntrax);
  end
end
end
```

```
function [iloc, ival, iphase] = pick_peaks_modulation19_12(mXp, pXp,...
  sfreq, Thres, MAFS, nPeaks)
%function [iloc, ival, iphase] = PickPeaks2(mXp, pXp, sfreq, slopeThreshold,
% ampThreshold, nPeaks)
%
%==> peaking the nPeaks highest peaks in the given mXp from the
%    greater to the lowest
% data output:
%       iloc:   frequency peaks (if isnan(iloc), no peak detected)
%       ival:   amplitude of the given peaks
%       iphase:
% data input:
%       mXp:    spectrum in dB: 20*log10(abs(fft(signal)) Positive part only
%       pXp:    positive phase of spectrum
%       sfreq:  sampling frequency
%       Thres:  Amplitude Threshold of signal to pick a peak
%       MAFS:   smallest separation that allows the peak detection
%       nPeaks: number of peaks to pick

% Initialze variables
N2 = length(mXp);
N = (N2-1)*2;                          % Number of FFT

tloc = find_peaks18_12(mXp, MAFS, Thres);
tval = mXp(tloc);
if isempty(tloc)
  iloc = zeros(1,nPeaks);
  ival = -Inf*ones(1,nPeaks);
  iphase = zeros(1,nPeaks);
  return
end
[tiloc, tival, tiphase] = interpolated_values(mXp, pXp, tloc, tval);
peaks = length(tiloc);

% Eliminate if there are more than nPeaks and sort by magnitude
if peaks>nPeaks
  ival = ones(nPeaks,1)*(-Inf);
  iloc = zeros(nPeaks,1);
  iphase = zeros(nPeaks,1);
  for p=1:nPeaks
    [ival(p), l] = max(tival);             % find current maximum
    iloc(p) = (tiloc(l)-1)*sfreq/N;
    iphase(p) = tiphase(l);
    tival(l) = [];
    tiloc(l) = [];
    tiphase(l) = [];
  end
else
  ival = ones(nPeaks,1)*(-Inf);
  iloc = zeros(nPeaks,1);
  iphase = zeros(nPeaks, 1);
  for p=1:peaks
    [ival(p), l] = max(tival);             % find current maximum
    iloc(p) = (tiloc(l)-1)*sfreq/N;
    iphase(p) = tiphase(l);
    tival(l) = [];
```

```
    tiloc(l) = [];
    tiphase(l) = [];
  end
end

    ival = ival';
    iloc = iloc';
    iphase = iphase';
return
```

```
function [iloc, ival, iphase] = pick_peaks18_12(mXp, pXp, sfreq, Thres, MAFS, minBin,
nPeaks)
% Creat el 18/12/2011
%function [iloc, ival, iphase] = PickPeaks(mXp, pXp, sfreq, nPeaks)
%
%==> peaking the nPeaks highest peaks in the given mXp from the
%    greater to the lowest
% data output:
%       iloc:   bin interpolated of peaks (if isnan(iloc), no peak detected)
%       ival:   interpolated amplitude of the given peaks in
%       iphase: interpolated phase of the peak
% data input:
%       mXp:       spectrum in dB (Positive part only)
%       pXp:       positive phase of spectrum
%       sfreq:     sampling frequency
%       Thres:     Amplitude threshold for peak detection [dB]
%       MAFS:      smallest separation that allows the peak detection
%       minBin:    bin high-pass cutoff detection
%       nPeaks:    number of peaks to pick

N2 = length(mXp);
NFFT = (N2-1)*2;
ploting = 0;
tloc = find_peaks18_12(mXp, MAFS, Thres);


% Eliminate peaks under minBin
tloc = tloc(find(tloc>=minBin));
tval = mXp(tloc);

% Ploting peaks detected
if ploting > 0
  At = -35;
  Al = 0;
  Ar = 38;
  b  = 0.0003;
  freq = ((1:N2)-1)*sfreq/NFFT;
  Tb = max(mXp)+At+Al+(Ar/(b-1))-(Ar/(b-1))*b.^(freq/20000);
  plot(freq, mXp, freq, Thres,'r',freq,Tb,'k');  % Graph the signal
  %plot(freq, mXp)
  line(freq,Thres)
  ylabel('dB');
  xlabel('Frequency [Hz]');
  title('Frame analyzed and Peaks detected')
  set(gca,'XScale', 'Log');
  axis([30 sfreq/2 Thres-10 (ceil(max(mXp)/10)*10)]);
  %axis([30 sfreq/2 -110 -20]);
end

if size(tloc,1) < 1
  iloc = [];
  ival = [];
  iphase = [];
  return
end
```

```
% Interpolating valures
[tiloc, tival, tiphase] = interpolated_values(mXp, pXp, tloc, tval);
peaks = length(tiloc);
mask = 1;
%-------------------------------------------------------------------------
% Eliminating masked sinusoids
if mask == 1
d = 6;              % masking threshold [dB]
S = 20;             % masking threshold at one critical band distance
i = 1;
while i<=peaks
  f0 = (tiloc(i)-1)*sfreq/NFFT;
  % Hearing Threshold in quiet
  %Tq = 3.64*(f0kHz)^(-0.8)-6.5*exp(-0.6*(f0kHz-3.3)^2)+0.001*(f0kHz)^4;
  %if ival(i) < Tq
  %  iloc(i) = [];
  %  ival(i) = [];
  %  i = i-1;
  %  peaks = peaks-1;
  %else
    % Masking curves
    BWCk = 100 + 0.1*f0+0.00001*f0*f0;
    j = 1;
    while j<=peaks
      f1 = (tiloc(j)-1)*sfreq/NFFT;      % frecuency of possible partial masked
      a1 = tival(j);                  % amplitude of possible partial masked
      maskThreshold = tival(i)-d-S/BWCk*abs(f1-f0);
      if a1 < maskThreshold
        if ploting > 0
          text((tiloc(j)-1)*sfreq/NFFT, tival(j), 'x')  % Number the peaks
        end
        tiloc(j) = [];
        tival(j) = [];
        tiphase(j) = [];
        j = j-1;
        peaks = peaks-1;
        if j<i
          i = i-1;
        end
      end
      j = j+1;
    end
  %end
  i = i+1;
end
end

% Eliminate if there are more than nPeaks and sort by magnitude
if peaks>nPeaks
  ival = ones(nPeaks,1)*NaN;
  iloc = zeros(nPeaks,1);
  iphase = zeros(nPeaks,1);
  for p=1:nPeaks
    [ival(p), l] = max(tival);            % find current maximum
    iloc(p) = tiloc(l);
    iphase(p) = tiphase(l);
    tival(l) = [];
```

```
    tiloc(l) = [];
    tiphase(l) = [];
  end
else
  ival = ones(peaks,1)*NaN;
  iloc = zeros(peaks,1);
  iphase = zeros(peaks, 1);
  for p=1:peaks
    [ival(p), l] = max(tival);            % find current maximum
    iloc(p) = tiloc(l);
    iphase(p) = tiphase(l);
    tival(l) = [];
    tiloc(l) = [];
    tiphase(l) = [];
  end
end


% Ploting final peaks detected
if ploting > 0
  text((iloc-1)*sfreq/NFFT, ival, 'o')  % Number the peaks
  text((tiloc-1)*sfreq/NFFT, tival, 'x')
end
return
```



```
function loc  = find_peaks18_12(mXp, MAFS, ThresD)
% function [loc val] = find_peaks18_12(mXp, ThresD, nPeaks, minBin)
%
%==> find peaks
% data output:
%       loc:    location of peak detected (if isnan(loc), no peak detected)
% data input:
%       mXp:    spectrum
%       MAFS:   smallest separation that allows for peak detection and
%                 does not introduce significant bias in the peak
%                 interpolation due to interference from a neighboring peak
%       ThresD: threshold peak detection
N2 = length(mXp);
MAFS2 = ceil(MAFS/2);
possiblePeaks = (mXp(MAFS2+1:N2-MAFS2) > ThresD);
for i = 0:MAFS2-1
  possiblePeaks = possiblePeaks .*...
    (mXp((MAFS2+1)-i:N2-(MAFS2+i))> mXp((MAFS2+1)-i-1:N2-(MAFS2+i+1))) .*...
    (mXp((MAFS2+1)+i:N2-(MAFS2-i))> mXp((MAFS2+1)+i+1:N2-(MAFS2-i-1)));
end
loc = find(possiblePeaks)+MAFS2;
val = mXp(loc);
```

```
function [iftloc, iftval, iftphase] = interpolated_values(r, phi, ftloc,
ftval)
% function [iftloc, iftphase, iftval] = InterpolatedValues(r, phi, ftloc,
ftval)
% ==> computation of the interpolated values of location and magnitude
%     (parabolic interpolation) and phase (linear interpolation)
% data output:
%       iftloc:     interpolated location (bin)
%       iftval:     interpolated magnitude
%       iftphase:   interpolated phase
% data input:
%       r:          magnitude of the FFT 20*log10(abs(fft(x)(1:N2)))
%       phi:        phase of the FFT (only positive part)
%       ftloc:      peak locations (bins)
%       ftval:      peak magnitudes

%peaks = length(ftloc);
%--- calculate interpolated peak position in bins (iftloc) --------------
--
leftftval = (r(ftloc-1));
rightftval = (r(ftloc+1));
ftval = (ftval);
iftloc = ftloc + .5*(leftftval - rightftval)./(leftftval-
2*ftval+rightftval);

%--- eliminate strange values of iftloc----------------------------------
--
iftlocKeep = find((iftloc <= (ftloc+1)).*(iftloc >= (ftloc-1)));  %
Tracks to keep
ftloc = ftloc(iftlocKeep);
iftloc = iftloc(iftlocKeep);
leftftval = leftftval(iftlocKeep);
rightftval = rightftval(iftlocKeep);
ftval = ftval(iftlocKeep);

%--- calculate interpolated phase ---------------------------------------
--
leftftphase = phi(floor(iftloc));
rightftphase = phi(ceil(iftloc));
delta = iftloc-floor(iftloc);
m = rightftphase-leftftphase;
iftphase = unwrap2pi(leftftphase+m.*delta);

%--- calculate interpolated amplitude -----------------------------------
--
iftval = ftval - .25*(leftftval-rightftval).*(iftloc-ftloc);
return

 function argunwrap = unwrap2pi(arg)
% function argunwrap = unwrap2pi(arg)
%==> unwrapping of the phase, in [-pi, pi]
% arg: phase to unwrap
arg = arg - floor(arg/2/pi)*2*pi;
argunwrap = arg - (arg>=pi)*2*pi;
```

```
function extract_modulation()
global freqTracks  ampTracks  phaseTracks
global iftfreq iftamp iftphase
global FAM  AM  PAM
global Fs H steps ntrax
global maxPeaksMod
global La
ploting = 0;
Na = 64;          % number of FFT for estimate modulation parameters
Na2 = Na/2+1;

% modulation, there should be, at least La values
% different from NaN and -1 in each partial. If not, no
% modulation will be added to the partial
indexes = zeros(ntrax,2);
for i=1:ntrax
  j = 0;
  while indexes(i,1)==0
    j = j+1;
    if(~isnan(freqTracks(i,j)) && freqTracks(i,j)~=(-1)) || (j == steps)
      indexes(i,1)=j;            % First no NaN number
    end
  end
  j = steps+1;
  while indexes(i,2)==0
    j = j-1;
    if(~isnan(freqTracks(i,j))) || (j == 1)
      indexes(i,2) = j;           % Last no NaN number
    end
  end
end
tracksToKeep  = (indexes(:,2)-indexes(:,1)+1)>=La;
tracksToKeep2 = (sum(~isnan(freqTracks),2)-indexes(:,1)+1)>=(0.5*La);
tracksToKeep  = logical(tracksToKeep.*tracksToKeep2);
freqTracks    = freqTracks(tracksToKeep,:);
ampTracks     = ampTracks(tracksToKeep,:);
phaseTracks   = phaseTracks(tracksToKeep,:);
indexes       = indexes(tracksToKeep,:);
ntrax         = size(freqTracks,1);
iftfreq       = zeros(ntrax,1);     % final frequency partial
synthetization
iftamp        = iftfreq;            % final amp partial synthetization
iftphase      = iftfreq;            % final phase partial synthetization


for i=1:ntrax
  %--- interpolating values that are not matched ---
  nan = find(isnan(freqTracks(i,:)));
  naninterp = nan(nan<indexes(i,2));                % indexes to
interpolate
  num = find(~(isnan(freqTracks(i,:))+(freqTracks(i,:)==-1))); % known
values
  if ~isempty(naninterp)
    freqTracks(i,naninterp) =
interp1(num,freqTracks(i,num),naninterp,'linear');
```

```
    ampTracks(i,naninterp) =
interp1(num,ampTracks(i,num),naninterp,'spline');
  end
  val = indexes(i,1):indexes(i,2);
  coef = polyfit(val,ampTracks(i,val),1);  % coefficients for linear
regresion
  PdBwithoutDecay = ampTracks(i,val)-(coef(1)*val+coef(2));
  % Extracting decay and amplitude of the Partial
  PwithoutDCDecay = 10.^(PdBwithoutDecay/20)-1;
  if(ploting == 1)
    figure(4)
    subplot(2,2,1)
    plot(ampTracks(i,val));
    hold on, plot((coef(1)*(val)+coef(2)),'r'), hold off;
    Ylabel('Amplitude (dB)')
    Xlabel('Frame Number')
    message = sprintf('Partial %d track analysis', i);
    title(message)

    subplot(2,2,2)
    plot(PwithoutDCDecay);
    Ylabel('Amplitude');
    Xlabel('Frame Number');
    message = sprintf('Amplitude Modulation without decay from %d', i);
    title(message)
  end
  % Modulation of the partial
  PWwithoutDCDecay = PwithoutDCDecay.*hamming(length(val))'/...
    sum(hamming(length(val)));
  fftPWwithoutDCDecay = fft(PWwithoutDCDecay,Na);
  mPp = 20*log10(abs(fftPWwithoutDCDecay(1:Na2)));
  pPp = unwrap(angle(fftPWwithoutDCDecay(1:Na2)));
  % Detecting peaks
  [FAM(i,:) AM(i,:) PAM(i,:)] = pick_peaks_modulation19_12(mPp', pPp',
...
    1/H*Fs, -45, 4, maxPeaksMod);
  % Adding the DC component
  if(ploting > 0)
    figure(4)
    subplot(2,2,4)
    plot(((1:Na2)-1)*1/H*Fs/Na, mPp);
    text(FAM(i,:), 20*log10(AM(i,:)), 'x')
    Ylabel('dB');
    Xlabel('Frequency [Hz]');
    title('FFT Track without decay and DC and the Peaks detected')
    axis([0 25 -80 10])

    fftP = fft(10.^((ampTracks(i,val))/20)/La,Na);
    mPDCDecayp = 20*log10(abs(fftP(1:Na2)));
    subplot(2,2,3)
    plot(((1:Na2)-1)*1/H*Fs/Na, mPDCDecayp);
    Ylabel('dB');
    Xlabel('Frequency [Hz]');
    title('FFT of Track analyzed')
    axis([0 25 -80 10])
```

```
  end
  for j = 1:size(FAM,2)
    PAM(i,j) = PAM(i,j)+2*pi*FAM(i,1)/Fs*(H)*(steps-indexes(i,1)+6);
  end
  iftfreq(i,1)  = sum(freqTracks(i,val))/length(val); % Frequency of the
partial
  if coef(1) < 0
    iftamp(i,1)   = (coef(1)*(steps+6)+coef(2));  % Amplitude on the next
frame
  else
    iftamp(i,1) = ampTracks(i,max(val));
  end
  iftphase(i,1) = phaseTracks(i,max(val))+2*pi*iftfreq(i,1)/Fs*...
    (steps-indexes(i,1)+6)*H;       % Phase of the partial
end
end
```

```
function [iloc, ival, iphase] = pick_peaks_modulation19_12(mXp, pXp,...
  sfreq, Thres, MAFS, nPeaks)
%function [iloc, ival, iphase] = PickPeaks2(mXp, pXp, sfreq,
slopeThreshold,
% ampThreshold, nPeaks)
%
%==> peaking the nPeaks highest peaks in the given mXp from the
%    greater to the lowest
% data output:
%       iloc:   frequency peaks (if isnan(iloc), no peak detected)
%       ival:   amplitude of the given peaks
%       iphase:
% data input:
%       mXp:    spectrum in dB: 20*log10(abs(fft(signal)) Positive part
only
%       pXp:    positive phase of spectrum
%       sfreq:  sampling frequency
%       Thres:  Amplitude Threshold of signal to pick a peak
%       MAFS:   smallest separation that allows the peak detection
%       nPeaks: number of peaks to pick

% Initialze variables
N2 = length(mXp);
N = (N2-1)*2;                          % Number of FFT

tloc = find_peaks18_12(mXp, MAFS, Thres);
tval = mXp(tloc);
if isempty(tloc)
  iloc = zeros(1,nPeaks);
  ival = -Inf*ones(1,nPeaks);
  iphase = zeros(1,nPeaks);
  return
end
[tiloc, tival, tiphase] = interpolated_values(mXp, pXp, tloc, tval);
peaks = length(tiloc);

% Eliminate if there are more than nPeaks and sort by magnitude
if peaks>nPeaks
  ival = ones(nPeaks,1)*(-Inf);
  iloc = zeros(nPeaks,1);
  iphase = zeros(nPeaks,1);
  for p=1:nPeaks
    [ival(p), l] = max(tival);             % find current maximum
    iloc(p) = (tiloc(l)-1)*sfreq/N;
    iphase(p) = tiphase(l);
    tival(l) = [];
    tiloc(l) = [];
    tiphase(l) = [];
  end
else
  ival = ones(nPeaks,1)*(-Inf);
  iloc = zeros(nPeaks,1);
  iphase = zeros(nPeaks, 1);
  for p=1:peaks
    [ival(p), l] = max(tival);             % find current maximum
    iloc(p) = (tiloc(l)-1)*sfreq/N;
```

```
    iphase(p) = tiphase(l);
    tival(l) = [];
    tiloc(l) = [];
    tiphase(l) = [];
  end
end

    ival = ival';
    iloc = iloc';
    iphase = iphase';
return
```



```
function output = synthesize()
%function output = synthesize()
%
%===> creates the synthesized sound from the estimated parameters

global FAM AM PAM
global iftfreq iftamp iftphase
global pageSize
global countSynth
global Tau
output = zeros(1,pageSize);

if ~isempty(iftphase)
  i = pageSize*countSynth:(pageSize*(countSynth+1)-1);
  for j = 1:size(iftfreq)
    decay = exp(-i./Tau(j));
    sinusoid = iftamp(j)*cos(iftfreq(j).*i+iftphase(j));
    mod = 1+AM(j,1)*cos(FAM(j,1)*i+PAM(j,1));
    output = output + mod.*decay.*sinusoid;
  end
end
```

```
function output = synth_release()
%function output = synthesize()
%
%===> creates the synthesized sound from the estimated parameters in
%     time domain with the release part

global FAM AM PAM
global iftfreq iftamp iftphase
global pageSize
global countSynth countRelease
global Tau Lambda
output = zeros(1,pageSize);

if ~isempty(iftphase)
  i = pageSize*countSynth:(pageSize*(countSynth+1)-1);
  l = pageSize*countRelease:(pageSize*(countRelease+1)-1);
  for j = 1:size(iftfreq)
    release = exp(-l*Lambda(j)-i./Tau(j));
    sinusoid = iftamp(j)*cos(iftfreq(j).*i+iftphase(j));
    mod = 1+AM(j,1)*cos(FAM(j,1)*i+PAM(j,1));
    output = output + mod.*release.*sinusoid;
  end
end
```