Chapter 9

Higher level spectral processing

Xavier Amatriain, Jordi Bonada, Alex Loscos, Xavier Serra

Index

9.1 INTRODUCTION 2

9.2 SPECTRAL MODELS 4

9.2.1 SINUSOIDAL MODEL 5
9.2.2 SINUSOIDAL PLUS RESIDUAL MODEL 5

9.3 TECHNIQUES 7

9.3.1 ANALYSIS 7
9.3.2 FEATURE ANALYSIS 24
9.3.3 SYNTHESIS 27
9.3.4 MAIN ANALYSIS-SYNTHESIS APPLICATION 32

9.4 FX AND TRANSFORMATIONS 38

9.4.1 FILTERING WITH ARBITRARY RESOLUTION 38
9.4.2 PARTIAL DEPENDENT FREQUENCY SCALING 39
9.4.3 PITCH TRANSPOSITION WITH TIMBRE PRESERVATION 40
9.4.4 VIBRATO AND TREMOLO 42
9.4.5 SPECTRAL SHAPE SHIFT 42
9.4.6 GENDER CHANGE 43
9.4.7 HARMONIZER 45
9.4.8 HOARSENESS 45
9.4.9 MORPHING 46

9.5 CONTENT DEPENDENT PROCESSING 47

9.5.1 REAL-TIME SINGING VOICE CONVERSION 47
9.5.2 TIME SCALING 51

9.6 CONCLUSIONS 56

9.7 REFERENCES 57

9.8 INDEX OF FIGURES 59

9 Higher level spectral processing

9.1 Introduction

In the context of this book, we are looking for representations of sound signals and signal processing systems that can give us ways to design sound transformations in a variety of music applications and contexts. It should have been clear throughout the book, that several points of view have to be considered, including a mathematical, thus objective perspective, and a cognitive, thus mainly subjective, standpoint. Both points of view are necessary to fully understand the concept of sound effects and to be able to use the described techniques in practical situations.

The mathematical and signal processing points of view are straightforward to present, which does not mean easy, since the language of the equations and of flow diagrams is suitable for them. However, the top-down implications are much harder to express due to the huge number of variables involved and to the inherent perceptual subjectivity of the music making process. This is clearly one of the main challenges of the book and the main reason for its existence.

The use of a spectral representation of a sound yields a perspective that is *sometimes* closer to the one used in a sound engineering approach. By understanding the basic concepts of frequency domain analysis, we are able to acquire the tools to use a large number of effects processors and to understand many types of sound transformations systems. Moreover, being the frequency domain analysis a somewhat similar process than the one performed by the human hearing system, it yields fairly intuitive intermediate representations.

The basic idea of spectral processing is that we can analyze a sound to obtain alternative frequency domain representations, which can then be transformed and inverted to produce new sounds (see Figure 9.1). Most of the approaches start by developing an analysis/synthesis system from which the input sound is reconstructed without any perceptual loss of sound quality. The techniques described in chapter 8 are clear examples of this approach. Then the main issue is what is the intermediate representation and what parameters are available for applying the desired transformations.

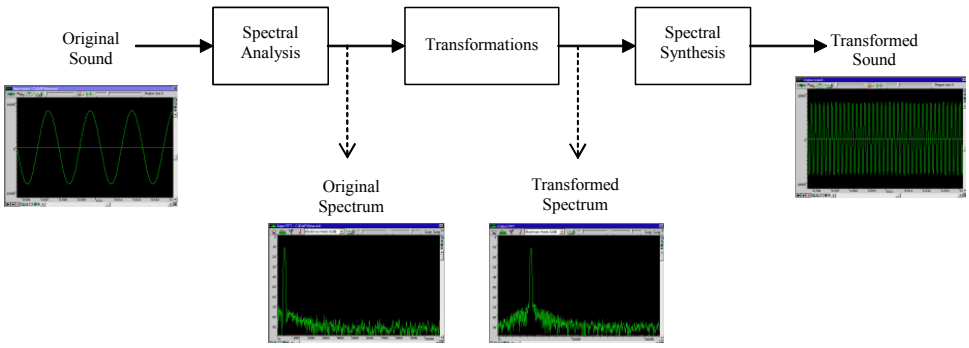


Fig 9.1. *Block diagram of a simple spectral processing framework*

Perceptual or musical concepts such as timbre or pitch are clearly related to the spectral characteristics of a sound. Even some common processes for sound effects are better explained using a frequency domain representation. We usually think on the frequency axis when we talk about equalizing, filtering, pitch shifting, harmonizing... In fact, some of them are specific to this signal processing approach and do not have an immediate counterpart on the time domain. On the other hand, most (but not all) of the sound effects presented in this book can be implemented in the frequency domain.

Another issue is whether or not this approach is the most efficient, or practical, for a given application. The process of transforming a time domain signal into a frequency domain representation is, by itself, not an immediate step. Some parameters are difficult to adjust and force us to take several compromises. Some settings, such as the size of the analysis window, have little or nothing to do with the high-level approach we intend to favor, and require the user to have a basic signal processing understanding.

In that sense, when we talk about higher level spectral processing we are thinking of an intermediate analysis step in which relevant features are extracted, or computed, from the spectrum. These relevant features should be much closer to a musical or high-level approach. We can then process the features themselves or even apply transformations that keep some of the features unchanged. For example, we can extract the fundamental frequency and the spectral shape from a sound and then modify the fundamental frequency without affecting the shape of the spectrum.

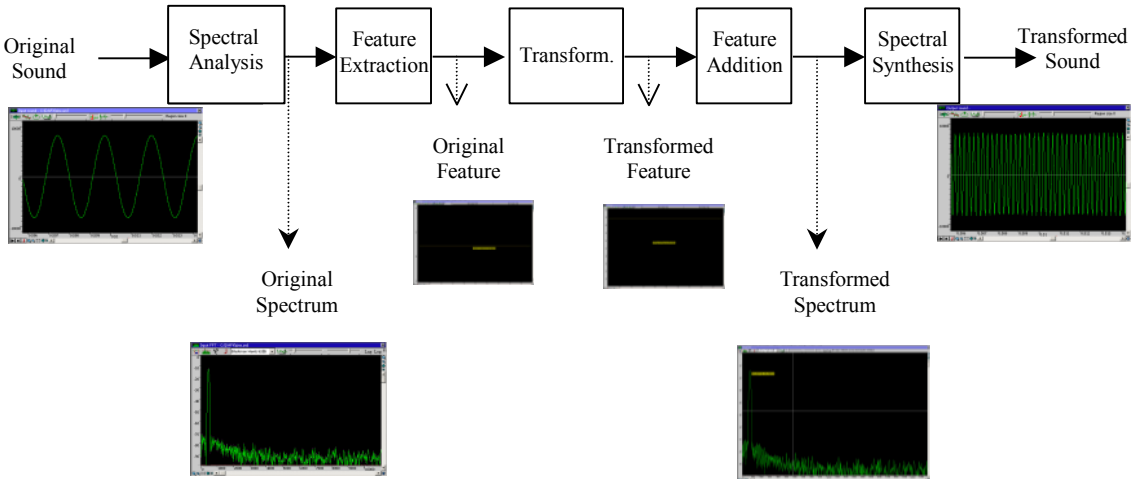


Fig 9.2. *Block diagram of a higher-level spectral processing framework*

Assuming the fact that there is no single representation and processing system optimal for everything, our approach will be to present a set of complementary spectral models that can be combined to be used for the largest possible set of sounds and musical applications.

In the next section we introduce two spectral models: *Sinusoidal* and *Sinusoidal plus Residual*. These models already represent a step up on the abstraction ladder and from either of them, we can identify and extract higher-level information of a sound, such as: harmonics, pitch, spectral shape, vibrato, or note boundaries, that is *Higher Level Features*. This analysis step brings the representation closer to our perceptual understanding of a sound. The complexity of the analysis will depend on the type of feature that we want to identify and the sound to analyze. The benefits of going to this higher level of analysis are enormous and open up a wide range of new musical applications.

Having set the basis of the *Sinusoidal plus Residual* model, we will then give some details of the techniques used both in its analysis and synthesis process, providing Matlab code to implement an analysis-synthesis framework. This Matlab implementation is based on the *Spectral Modeling Synthesis* framework. SMS [http://www.iua.upf.es/~sms] is a set of spectral based techniques and related implementations for the analysis/transformation/synthesis of an audio signal based on the scheme presented in Fig 9.2.

In section 9.4 we will provide a set of basic audio effects and transformations based on the implemented *Sinusoidal plus Residual* analysis/synthesis. Matlab code is provided for all of them.

We will finish with an explanation of content dependant processing implementations. In section 9.5.1 we introduce a real-time singing voice conversion application that has been developed for use in Karaoke applications, and in section 9.5.2 we define the basis of a nearly loss less Time Scaling algorithm. The complexity and extension of these implementations prevent us from providing the associated Matlab code, so we leave that task as a challenge for advanced readers.

9.2 Spectral Models

The most common approach for converting a time domain signal into its frequency domain representation is the *Short-Time Fourier Transform (STFT*). It is a general technique from which we can implement loss-less analysis/synthesis systems. Many sound transformation systems are based on direct implementations of the basic algorithm and several examples have been presented in chapter 8.

In this chapter, we will briefly mention the *Sinusoidal Model* and will concentrate, with a Matlab sample code, in the *Sinusoidal plus Residual Model*. Anyhow, the decision as to what spectral representation to use in a particular situation is not an easy one. The boundaries are not clear and there are always compromises to take into account, such as: (1) sound fidelity, (2) flexibility, (3) coding efficiency, and (4) computational requirements. Ideally, we want to maximize fidelity and flexibility while minimizing memory consumption and computational requirements. The best choice for maximum fidelity and minimum compute time is the STFT that, anyhow, yields a rather inflexible representation and inefficient coding scheme. Thus our interest in finding higher-level representations as the ones we present in this section.

9.2.1 Sinusoidal Model

Using the output of the *STFT*, the *Sinusoidal* model represents a step towards a more flexible representations while compromising both sound fidelity and computing time. It is based on modeling the time-varying spectral characteristics of a sound as sums of time-varying sinusoids. The input sound $s(t)$ is modeled by,

$$(1) \qquad s(t) = \sum_{r=1}^{R} A_r(t)\cos\left[\theta_r(t)\right] + e(t)$$

where $A_r(t)$ and $\theta_r(t)$ are the instantaneous amplitude and phase of the r^{th} sinusoid, respectively. [McAulay and Quatieri, 1986; Smith and Serra, 1987].

To obtain a sinusoidal representation from a sound, an analysis is performed in order to estimate the instantaneous amplitudes and phases of the sinusoids. This estimation is generally done by first computing the STFT of the sound, as described in chapter 8, then detecting the spectral peaks (and measuring the magnitude, frequency and phase of each one), and finally organizing them as time-varying sinusoidal tracks.

It is a quite general technique that can be used in a wide range of sounds and offers a gain in flexibility compared with the direct *STFT* implementation.

9.2.2 Sinusoidal plus Residual Model

The *Sinusoidal plus Residual* model can cover a wide "compromise space" and can in fact be seen as the generalization of both the *STFT* and the *Sinusoidal* models. Using this approach, we can decide what part of the spectral information is modeled as *sinusoids* and what is left as *STFT*. With a good analysis, the *Sinusoidal plus Residual* representation is very flexible while maintaining a good sound fidelity, and the representation is quite efficient. In this approach, the *Sinusoidal* representation is used to model only the stable partials of a sound. The residual, or its approximation, models what is left, which should ideally be a stochastic component. This model is less general than either the *STFT* or the S*inusoidal* representations but it results in an enormous gain in flexibility [Serra 1989,1996; Serra and Smith 1990].

The input sound $s(t)$ is modeled by,

$$(2) \qquad s(t) = \sum_{r=1}^{R} A_r(t)\cos\left[\theta_r(t)\right] + e(t)$$

where $A_r(t)$ and $\theta_r(t)$ are the instantaneous amplitude and phase of the r^{th} sinusoid, respectively, and $e(t)$ is the noise component at time t (in seconds).

The sinusoidal plus residual model assumes that the sinusoids are stable partials of the sound with a slowly changing amplitude and frequency. With this restriction, we are able to add major constraints to the detection of sinusoids in the spectrum and omit the detection of the phase of each peak. The instantaneous phase that appears in the equation is taken to be the integral of the instantaneous frequency $\omega_r(t)$, and therefore satisfies

$$\theta_r(t) = \int_0^t \omega_r(\tau) d\tau \qquad (3)$$

where $\omega(t)$ is the frequency in radians, and r is the sinusoid number. When the sinusoids are used to model only the stable partials of the sound, we refer to this part of the sound as the deterministic component.

Within this model we can either leave the residual signal, $e(t)$, to be the difference between the original sound and the sinusoidal component, resulting into an identity system, or we can assume that $e(t)$ is a stochastic signal. In this case, the residual can be described as filtered white noise,

$$e(t) = \int_0^t h(t,\tau) u(\tau) d\tau \qquad (4)$$

where $u(t)$ is white noise and $h(t,\tau)$ is the response of a time varying filter to an impulse at time t. That is, the residual is modeled by the time-domain convolution of white noise with a time-varying frequency-shaping filter.

The implementation of the analysis for the Sinusoidal plus Residual Model is more complex than the one for the Sinusoidal Model. *Figure 9.3* shows a simplified block-diagram of this analysis.

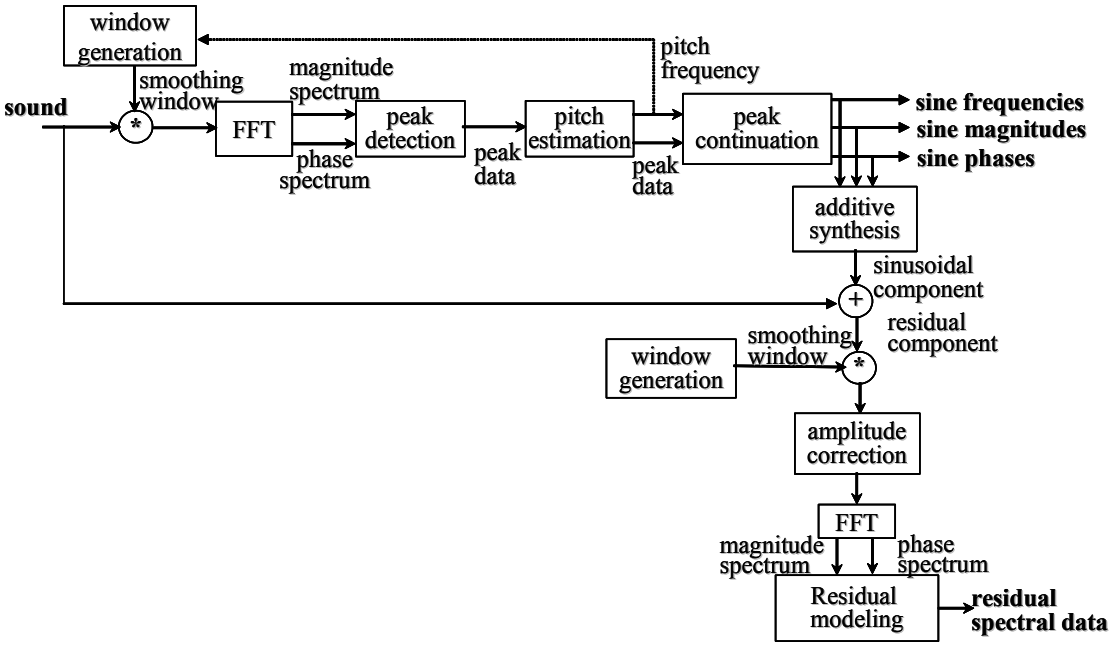


Fig 9.3. *Block diagram of the Sinusoidal plus residual analysis.*

The first few steps are the same than in a sinusoidal-only analysis. The major differences start on the peak continuation process since in order to have a good partial-residual decomposition we have to refine the peak-continuation process in such a way as to be able to identify the stable partials of the sound. Several strategies can be used to accomplish this. The simplest case is when the sound is monophonic and pseudo-harmonic. By using the fundamental frequency information in the peak continuation algorithm, we can identify the harmonic partials.

The residual component is obtained by first generating the sinusoidal component with additive synthesis, and then subtracting it from the original waveform. This is possible because the instantaneous phases of the original sound are matched and therefore the

shape of the time domain waveform preserved. A spectral analysis of this time domain residual is done by first windowing it, window which is independent of the one used to find sinusoids, and thus we are free to choose a different time-frequency compromise. An amplitude correction step can improve the time smearing produced in the sinusoidal subtraction. Then the FFT is computed and the resulting spectrum can be modeled using several existing techniques. The spectral phases might be discarded if the residual can be approximated as a stochastic signal.

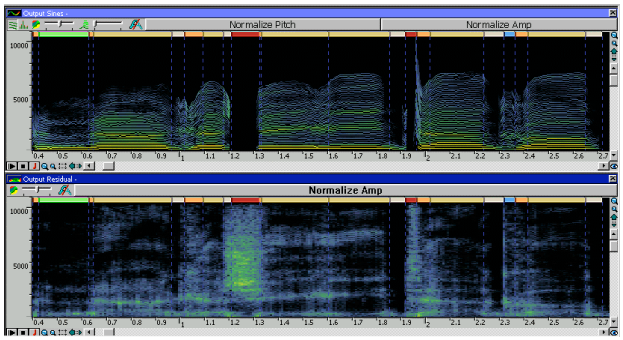


Fig 9.4. *a. Sinusoidal component. b. Residual spectrum*

The original sinusoidal plus residual model has led to other different spectral models that still share some of its basis [Ding and Qian, 1997; Fitz, Haken and Christensen, 2000; Verma, 2000].

9.3 Techniques

It is beyond the scope of this chapter to discuss the details of the whole analysis-synthesis process that results into a Sinusoidal plus Residual representation of the sound, but lets describe the major steps.

9.3.1 Analysis

The analysis step of the Sinusoidal plus Residual model was already presented in the previous section and it is illustrated in Fig 9.3. Next we will introduce the most important techniques and the basic considerations that need be taken into account when analyzing a sound.

9.3.1.1 Previous considerations: STFT settings

In the current section, we will see that the STFT process is far from being unsupervised, and its settings are indeed critical in order to get a good representation of the sound. The main parameters involved in this step are window size, window type, frame size and hop size.

As it has already been mentioned in previous chapters, the first step involved in the process of converting a time domain signal into its frequency domain representation,

is the windowing of the sound. This operation involves selecting a number of samples from the sound signal and multiplying their value by a windowing function. [Harris, 1978]

The number of samples taken in every processing step is defined by the window size. It is a crucial parameter, especially if we take into account that the number of spectral samples that the DFT will yield at its output, corresponds to half the number of samples of its input spread over half of the original sampling rate. We will not go into the details of the DFT mathematics that lead to this property, but it is very important to note that the longer the window, the more frequency resolution we will have. On the other hand, it is almost immediate to see the drawback of taking very long windows: the loss of temporal resolution. This phenomenon is known as the time vs. frequency resolution tradeoff (see Fig 9.5). A more specific limitation of the window size has to do with choosing windows with odd sample-length in order to guarantee even symmetry about the origin.

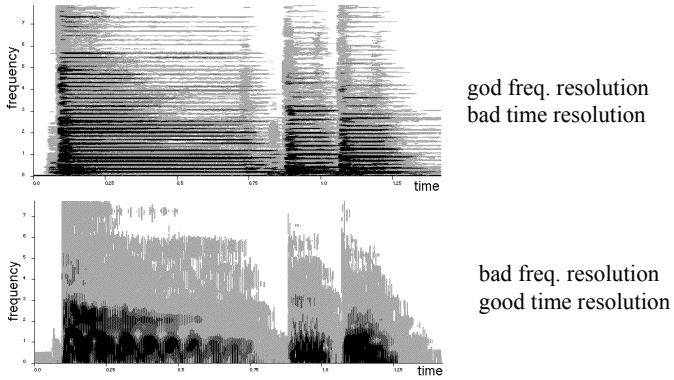


Fig 9.5. *Time vs. frequency resolution tradeoff*

The kind of window used has also a very strong effect on the qualities of the spectral representation we will obtain. At this point we should remember that a time domain multiplication (as the one done between the signal and the windowing function), becomes a frequency domain convolution between the Fourier Transforms of each of the signals (see Fig 9.6). One may be tempted to forget about deciding on these matters and apply no window at all, just taking *n* samples from the signal and feeding them to the chosen FFT algorithm. Even in that case, though, a rectangular window is being used, so the spectrum of the signal is being convolved with the transform of a rectangular pulse, a *sinc-like* function.

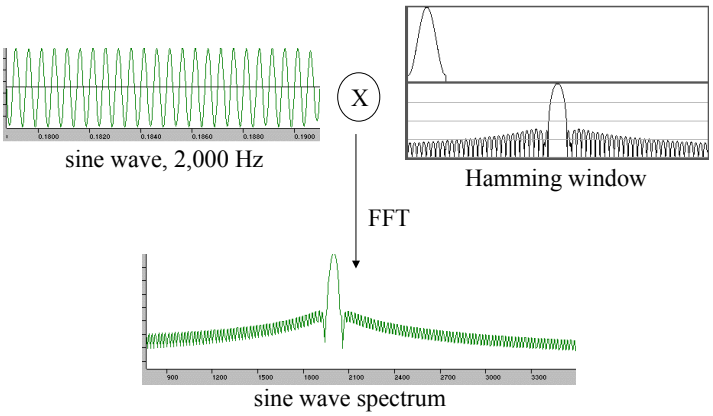
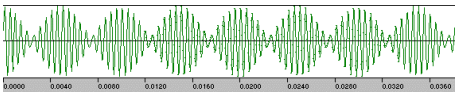


Fig 9.6. *Effect of applying a window in the time domain*

Two features of the transform of the window are specially relevant to whether a particular function is useful or not: the width of the main lobe, and the main to highest side lobe relation. The main lobe bandwidth is expressed in bins (spectral samples) and, in conjunction with the window size, defines the ability to distinguish two sinusoidal peaks. The following formula expresses the relation that the window size (M), the main lobe bandwidth (B_s) and the sampling rate (f_s) should meet in order to distinguish two sinus of frequency f_k and f_{k+1}:

$$M \geq B_s \frac{f_s}{|f_{k+1} - f_k|} \tag{5}$$



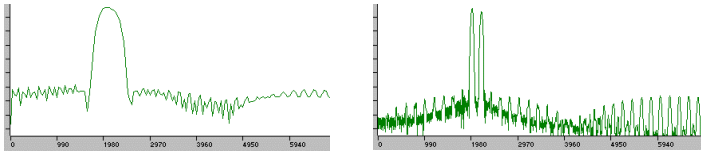


Fig 9.7. *Effect of the window size in distinguishing between two sinusoids*

The amplitude relation between the main and the highest side lobe explains the amount of distortion a peak will receive from surrounding partials. It would be ideal to have a window with an extremely narrow main lobe and a very high main to secondary lobe relation. However, the inherent tradeoff between these two parameters forces a compromise to be taken.

Common windows that can be used in the analysis step are: Rectangular, Triangular, Kaiser-Bessel, Hamming, Hanning and Blackman-Harris. In the Matlab code supplied in this chapter, we have chosen a Blackman-Harris 92 dB window for the sake of simplicity. This window has a rather wide main lobe (9 bins) but an extremely high

main-to-secondary lobe relation, 92 dB. This difference is so close to the dynamic range of a 16-bit representation that, in that case, we need only take into account the influence of the main lobe.

The following Matlab function implements the generation of a Blackman-Harris table:

```
function [bh92SINE2SINE,bh92SINE2SINEsize] = bh92SINE2SINEgeneration;
% function [bh92SINE2SINE,bh92SINE2SINEsize] =
bh92SINE2SINEgeneration;
%
% ==> generation of the Blackman-Harris window
% output data:
%     bh92SINE2SINEsize: size of the window
%     bh92SINE2SINE: (sampled) window

bh92SINE2SINEsize = 4096;
bh92SINE2SINE = zeros(bh92SINE2SINEsize,1);
bh92N = 512;
bh92const = [.35875, .48829, .14128, .01168];
bh92Theta = -4*2*pi/bh92N;
bh92ThetaIncr = 8*2*pi/bh92N/bh92SINE2SINEsize;
for i=1:bh92SINE2SINEsize
   for m=0:3
      bh92SINE2SINE(i)=bh92SINE2SINE(i)-bh92const(m+1)/2*...
         (sine2sine(bh92Theta-m*2*pi/bh92N,bh92N)+...
         sine2sine(bh92Theta+m*2*pi/bh92N,bh92N));
   end;
   bh92Theta = bh92Theta + bh92ThetaIncr;
end;
bh92SINE2SINE = bh92SINE2SINE/bh92SINE2SINE(bh92SINE2SINEsize/2+1);
```

The value of the *sine2sine* function (not included in the basic Matlab package) is computed as follows:

```
function x = sine2sine( x , N )
% sine2sine function !!!

x = sin((N/2)*x) / sin(x/2);
```

One may think that a possible way of overcoming the time/frequency tradeoff is to add zeros to the windowed signals in order to have a longer FFT and so increase the frequency resolution. This process is known as *zero padding* and it represents an interpolation in the frequency domain. Thus, when we zero-pad a signal before the DFT process, we are not adding any information to its frequency representation (we will still not distinguish to sinusoids if eq.5 is not satisfied), but we are indeed increasing the frequency resolution by adding intermediate interpolated bins. This process can help in the peak detection process as later explained.

A final step is the circular shift already described in 8.2.1. This *buffer centering* guarantees the preservation of zero-phase conditions in the analysis process.

Once the spectrum of a *frame* has been computed, the window must move to the next position in the waveform in order to take the next set of samples. The distance between the centers of two consecutive windows is known as *hop size*. If the hop size is smaller than the window size, we will be including some overlap, that is, some samples will be used more than once in the analysis process. In general, the more

overlap, the smoother the transitions of the spectrum will be across time, but that is a computational expensive process. The window type and the hop must be chosen in such a way that the resulting envelope adds approximately to a constant, following the next equation:

$$A_w(m) \equiv \sum_{n=-\infty}^{\infty} w(m - nH) \approx \text{constant} \quad (6)$$

A measure of the deviation of A_w from a constant is the difference between the maximum and minimum values for the envelope as a percentage of the maximum value. This measure is referred to as the *amplitude deviation* of the overlap factor, variables should be chosen so to keep this factor around or below one per cent.

$$d_w = 100 \times \frac{\max_m \left[A_w(m)\right] - \min_m \left[A_w(m)\right]}{\max_m \left[A_w(m)\right]} \quad (7)$$

We have seen that the STFT process that is bound to provide a suitable frequency domain representation of the input signal, is a far from trivial process and is dependant on some low-level parameters closely related to the signal processing domain. A little theoretical knowledge is required but only practice will surely lead to the desired results.

9.3.1.2 Peak Detection

The sinusoidal model assumes that each spectrum of the STFT representation can be explained by a series of sinusoids. Given enough frequency resolution, thus, enough points in the spectrum, a sinusoid can be identified by its shape. Theoretically, a sinusoid that is stable both in amplitude and in frequency, a partial, has a well-defined frequency representation: the transform of the analysis window used to compute the Fourier transform. It should be possible to take advantage of this characteristic to distinguish partials from other frequency components. However, in practice this is rarely the case, since most natural sounds are not perfectly periodic and do not have nicely spaced and clearly defined peaks in the frequency domain. There are interactions between the different components, and the shapes of the spectral peaks cannot be detected without tolerating some mismatch. Only some instrumental sounds (e.g., the steady-state part of an oboe sound) are periodic enough and sufficiently free from prominent noise components that the frequency representation of a stable sinusoid can be recognized easily in a single spectrum. A practical solution is to detect as many peaks as possible, with some small constraints, and delay the decision of what is a "well behaved" partial, to the next step in the analysis: the peak continuation algorithm.

A "peak" is defined as a local maximum in the magnitude spectrum, and the only practical constraints to be made in the peak search are to have a frequency range and a magnitude threshold.

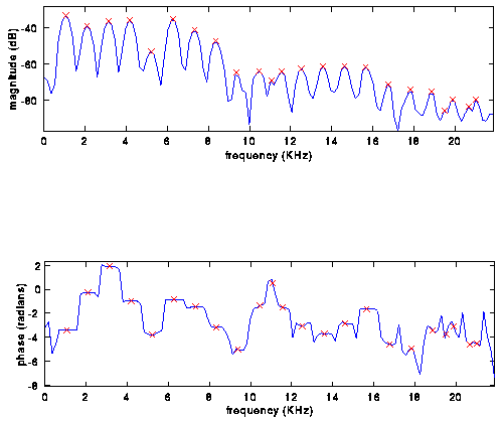


Fig 9.8. *Peak detection. a. Peaks in the magnitude spectrum. b. Peaks in the phase spectrum.*

Due to the sampled nature of the spectrum returned by the FFT, each peak is accurate only to within half a sample. A spectral sample represents a frequency interval of f_s / N Hz, where f_s is the sampling rate and N is the FFT size. Zero-padding in the time domain increases the number of spectral samples per Hz and thus increases the accuracy of the simple peak detection (see previous section). However, to obtain frequency accuracy on the level of 0.1% of the distance from the top of an ideal peak to its first zero crossing (in the case of a Rectangular window), the zero-padding factor required is 1000.

A more efficient spectral interpolation scheme is to zero-pad only enough so that quadratic (or other simple) spectral interpolation, using only samples immediately surrounding the maximum-magnitude sample, suffices to refine the estimate to .1% accuracy. That is the approach we have chosen and is illustrated in the following picture.

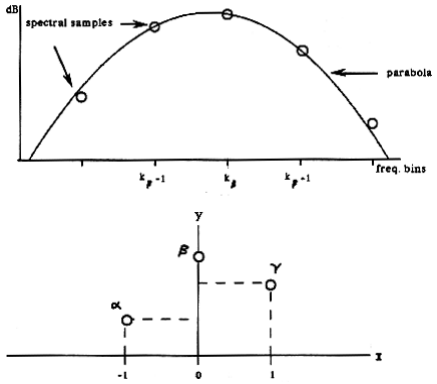


Fig 9.9. *Parabolic interpolation in the peak detection process*

The frequency and magnitude of a peak is obtained from the magnitude spectrum expressed in dB. Then the phase value of the peak is measured by reading the value of the unwrapped phase spectrum at the position resulting from the frequency of the peak.

Although we cannot rely on the exact shape of the peak to decide whether it is a partial or not, it is sometimes useful to have a measure of how close its shape is to the ideal sinusoidal peak. With this idea in mind, different techniques have been used in order to improve the estimation of the spectral peaks parameters. [Depalle and Hélie, 1997]

The following Matlab Code implements the peak detection algorithm, the function *pickpeak* finds the local maximums of the spectrum:

```
function  [loc, val] = PickPeaks(spectrum, nPeaks, minspace)
%function  [loc, val] = pickpeaks(spectrum, nPeaks, minspace)
%
%==> peaking the nPicks highest peaks in the given spectrum
%    from the greater to the lowest
% data:
%       loc: bin number of peaks  (if loc(i)==0, no peak detected)
%       val: amplitude of the given spectrum
%       spectrum: spectrum (abs(fft(signal))
%       nPicks: number of peaks to pick
%       minspace: minimum of space between two peaks

[r, c]  = size(spectrum);
rmin = min(spectrum) - 1;

% ---find a peak, zero out the data around the peak, and repeat
val =  ones(nPeaks,c)*NaN;
loc =  zeros(nPeaks,c);

for k=1:c      %--- find all local peaks
   difference = diff([rmin; spectrum(:,k); rmin]);            %
derivate
   iloc = find(difference(1:r)>= 0 & difference(2:r+1) <= 0); % peak
locations
   ival = spectrum(iloc,k);          % peak values

   for p=1:nPeaks
      [val(p,k),l] = max(ival);      % find current maximum
      loc(p,k) = iloc(l);            % save value and location
      ind = find(abs(iloc(l)-iloc) > minspace); % find peaks which
are far away
      if (isempty(ind))
         break                       % no more local peaks to pick
      end
      ival = ival(ind);                       % shrink peak value and
location array
      iloc = iloc(ind);
   end
end
```

and the function *interpolatedValues* computes interpolated values for each peak,

```
function [iftloc, iftphase, iftval] = interpolatedValues ...
   (r, phi, N, zp, ftloc, ftval)
%function [iftloc, iftphase, iftval] = interpolatedValues ...
%   (r, phi, N, zp, ftloc, ftval)
%
%==> calculatus of the interpolated values of location (bin),
%    phase and magnitude by cubic interpolation
% data:
%      iftloc:    interpolated location (bin)
%      iftval:    interpolated magnitude
%      iftphase:  interpolated phase
%      ftloc:     peak locations (bin)
%      ftval:     peak magnitudes
%      r:         modulus of the FFT
%      phi:       phase of the FFT
```

```
%      N:         size of the FFT
%      zp:        zero-padding multiplicative coefficient

%--- calculate interpolated peak position in bins (iftloc) ----------
-
leftftval = r( (ftloc-1).*((ftloc-1)>0) + ((ftloc-1)<=0).*1 );
rightftval= r( (ftloc+1).*((ftloc+1)<N/2) + ((ftloc+1)>=N/2).*(N/2)
);
leftftval = 20*log10(leftftval);
rightftval= 20*log10(rightftval);
ftval     = 20*log10(ftval);
iftloc    = ftloc + .5*(leftftval - rightftval) ./ ...
   (leftftval - 2*ftval + rightftval);

%--- interpolated ftloc ---------------------------------------------
-
iftloc    = (iftloc>=1).*iftloc + (iftloc<1).*1;
iftloc    = (iftloc>N/2+1).*(zp/2+1) + (iftloc<=N/2+1).*iftloc;

%--- calculate interpolated phase (iphase) --------------------------
-
leftftphase = phi(floor(iftloc));
rightftphase= phi(floor(iftloc)+1);
intpfactor  = iftloc-ftloc;
intpfactor  = (intpfactor>0).*intpfactor +
(intpfactor<0).*(1+intpfactor);
diffphase   = unwrap2pi(rightftphase-leftftphase);
iftphase    = leftftphase+intpfactor.*diffphase;

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

This function (as well as others that will be introduced later in this chapter) make use of the `argunwrap` function, which codes is the following:

```
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;
```

9.3.1.3 Pitch Estimation

Although the term *Pitch* should be ideally used to refer only to perceptual issues, the term *Fundamental Frequency* is not suitable to describe the output of techniques that will be herein explained. For that reason we will use both terms without making any distinction to refer to the output of these algorithms that aim at providing an estimation of this psycho acoustical sensation that is often (but not always) explained by the value of the fundamental frequency of a given harmonic series.

Pitch estimation is an optional step used when we know that the input sound is monophonic and pseudo-harmonic. Given this restriction and the set of spectral peaks of a frame, obtained as in the *Sinusoidal Analysis*, with magnitude and frequency values for each one, there are many possible pitch estimation strategies, none of them perfect [Hess, 1983; Maher and Beauchamp, 1994; Cano 1998]. The most obvious approach is to define the pitch as the common divisor of the harmonic series that best explains the spectral peaks found in a given frame. For example, in the Two-Way Mismatch procedure proposed by Maher and Beauchamp the estimated F_0 is chosen as to minimize discrepancies between measured peak frequencies and the harmonic frequencies generated by trial values of F_0. For each trial F_0, mismatches between the harmonics generated and the measured peak frequencies are averaged over a fixed subset of the available peaks. This is a basic idea on top of which we can add features and tune all the parameters for a given family of sounds.

Many trade-offs are involved in the implementation of a fundamental frequency detection system and every application will require a clear design strategy. For example, the issue of real-time performance is a requirement with strong design implications. We can add context specific optimizations when knowledge on the signal is available. Knowing, for instance, the frequency range of the F_0 of a particular sound helps both the accuracy and the computational cost. Then, there are sounds with specific characteristics, like in a clarinet where the even partials are softer than the odd ones. From this information, we can define a set of rules that will improve the performance of the used estimator.

In the framework of the sinusoidal plus residual analysis system, there are strong dependencies between the fundamental frequency detection step and many other analysis steps. For example, choosing an appropriate window for the Fourier analysis will facilitate detecting the fundamental and, at the same time, getting a good fundamental frequency will assist other analysis steps, including the selection of an appropriate window. Thus, it could be designed as a recursive process.

The following Matlab code implements an algorithm for pitch detection (Note that first, different computations are accomplished in order to decide if the region being analyzed is harmonic or not):

```
function [pitchvalue, pitcherror, isHarm] = pitchDetection (r, N, SR,
nPeaks, iftloc, iftval)
% function [pitchvalue, pitcherror, isHarm] = pitchDetection (r, N,
SR, nPeaks, iftloc, iftval)
%
%==> pitch detection function, using the Two-Way Mismatch algorithm
(see TWM.m)
% data:
%      r:       FFT magnitude
%      N:       size of the FFT
%      SR:      sampling rate
%      nPeaks:  number of peaks tracked
%      iftloc, iftval: location (bin) and magnitude of the peak

%--- harmonicity evaluation of the signal
highenergy = sum(r(round(5000/SR*N):N/2)); % 5000 Hz to SR/2 Hz
lowenergy  = sum(r(round(50/SR*N):round(2000/SR*N))); % 50 Hz to 2000
Hz
isHarm = max(0,(highenergy/lowenergy < 0.6));
```

```
if (isHarm==1)    %--- 2-way mismatch pitch estimation when harmonic
   npitchpeaks = min(50,nPeaks);
   [pitchvalue,pitcherror] =
TWM(iftloc(1:npitchpeaks),iftval(1:npitchpeaks),N,SR);
else
   pitchvalue = 0;
   pitcherror = 0;
end;

%--- in case of two much pitch error, signal supposed to be inhamonic
isHarm = min (isHarm,(pitcherror<=1.5));
```

And the Two-way mismatch procedure is implemented as follows:

```
function [pitch, pitcherror] = TWM (iloc, ival, N, SR)
%function [pitch, pitcherror] = TWM (iloc, ival, N, SR)
%
%==> Two-way mismatch error pitch detection using Bauchamp & Maher
algorithm
% data:  iloc:  location (bin) of the peaks
%        ival:  magnitudes of the peaks
%        N:     number of peaks
%        SR:    sampling rate


ifreq = (iloc-1)/N*SR;  % frequency in Hertz

%--- avoid zero frequency peak
[zvalue,zindex] = min(ifreq);
if (zvalue==0)
   ifreq(zindex) = 1;
   ival(zindex) = -100;
end

ival2 = ival;
[MaxMag,MaxLoc1] = max(ival2);
ival2(MaxLoc1)   = -100;
[MaxMag2,MaxLoc2]= max(ival2);
ival2(MaxLoc2)   = -100;
[MaxMag3,MaxLoc3]= max(ival2);

%--- pitch candidates
nCand = 10; % number of candidates
pitchc = zeros(1,3*nCand);
pitchc(1:nCand) = (ifreq(MaxLoc1)*ones(1,nCand))./((nCand+1-
[1:nCand]));
pitchc(nCand+1:nCand*2) = (ifreq(MaxLoc2)*ones(1,nCand))./ ((nCand+1-
[1:nCand]));
pitchc(nCand*2+1:nCand*3) =
(ifreq(MaxLoc3)*ones(1,nCand))./((nCand+1-[1:nCand]));
%pitchc=100:300;
harmonic = pitchc;

%--- predicted to measured mismatch error
ErrorPM = zeros(fliplr(size(harmonic)));
MaxNPM = min(10,length(iloc));
for i=1:MaxNPM
   difmatrixPM = harmonic' * ones(size(ifreq))';
```

```
   difmatrixPM = abs(difmatrixPM -
ones(fliplr(size(harmonic)))*ifreq');
   [FreqDistance,peakloc] = min(difmatrixPM,[],2);
   Ponddif     = FreqDistance .* (harmonic'.^(-0.5));
   PeakMag     = ival(peakloc);
   MagFactor   = max(0, MaxMag - PeakMag + 20);
   MagFactor   = max(0, 1.0 - MagFactor/75.0);
   ErrorPM     = ErrorPM + (Ponddif + MagFactor .* (1.4*Ponddif-
0.5));
   harmonic    = harmonic + pitchc;
end

%--- measured to predicted mismatch error
ErrorMP = zeros (fliplr(size(harmonic)));
MaxNMP  = min(10,length(ifreq));

for i=1:length(pitchc)
   nharm        = round(ifreq(1:MaxNMP)/pitchc(i));
   nharm        = (nharm>=1).*nharm + (nharm<1);
   FreqDistance = abs(ifreq(1:MaxNMP) - nharm*pitchc(i));
   Ponddif      = FreqDistance.* (ifreq(1:MaxNMP).^(-0.5));
   PeakMag      = ival(1:MaxNMP);
   MagFactor    = max(0,MaxMag - PeakMag + 20);
   MagFactor    = max(0,1.0 - MagFactor/75.0);
   ErrorMP(i)   = sum(MagFactor.*(Ponddif + MagFactor .*(1.4*Ponddif-
0.5)));
end

%--- total error
Error = (ErrorPM/MaxNPM) + (0.3*ErrorMP/MaxNMP);
[pitcherror, pitchindex] = min(Error);

pitch = pitchc(pitchindex);
```

9.3.1.4 Peak Continuation

The peak detection process returns the estimated magnitude, frequency, and phase of the prominent peaks in a given frame sorted by frequency. Once the spectral peaks of a frame have been detected, and possibly a fundamental frequency identified, a peak continuation algorithm can organize the peaks into time-varying trajectories.

The output of the Sinusoidal Analysis is a set of spectral peak values (frequency, magnitude and phase) organized into frequency trajectories, where each trajectory models a time-varying sinusoid. As it will be shown later, from this information we can synthesize a sound using additive synthesis. The less restrictive the peak detection step is, the more faithful the reconstruction of the original sound will be after synthesis.

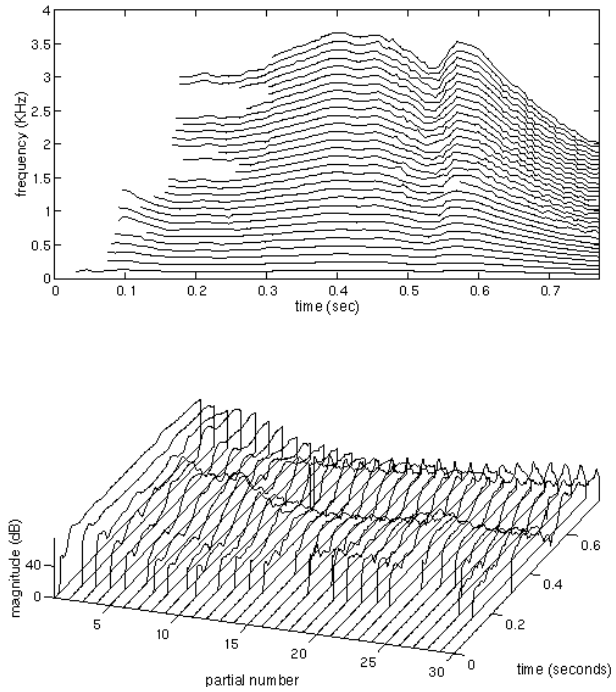


Fig 9.10. *Frequency trajectories resulting from the sinusoidal analysis of a vocal sound.*

The sinusoidal model assumes that each of these peaks is part of frequency trajectory and the peak continuation algorithm is responsible for assigning each peak to a given "track". There are many possibilities for such a process. The original one used by McAulay and Quatiery in their sinusoidal representation [McAulay and Quatiery, 1986] is based on finding, for each peak, the closest one in frequency in the following frame.

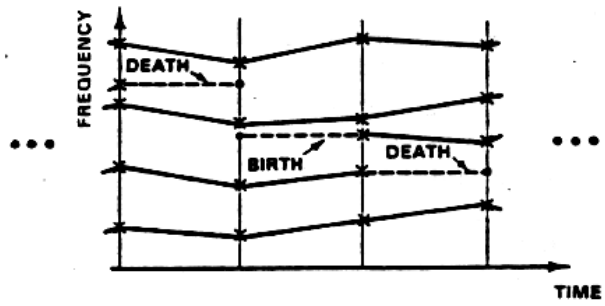


Fig 9.11. *Traditional peak continuation algorithm[McAulay and Quatiery, 1986]*

The schemes used in the traditional sinusoidal model (as the one just presented in this section), incorporate all the spectral peaks into trajectories, thus obtaining a sinusoidal representation for the whole sound. These schemes are not optimal when we want the trajectories to follow just the stable partials, leaving the rest to be modeled as part of the residual component. For example, when the partials change in frequency

significantly from one frame to the next, these algorithms easily switch from the partial that they were tracking to another one that at that point is closer.

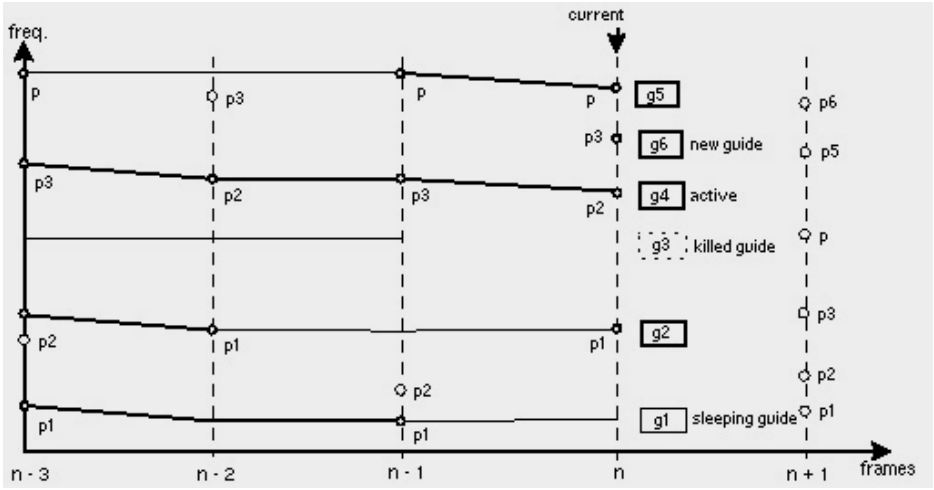


Fig 9.12. *Peak Continuation process. g represent the guides and p the spectral peaks.*

Here we will describe a basic framework under which we can define rules for specifying the behavior of the partials of musical sounds and thus implement systems for identifying partials out of spectral peaks. The behavior of a partial, and therefore the way to track it, varies depending on the signal. Whether we have speech, a harmonic instrumental tone, a gong sound, a sound of an animal, or any other, the time-varying behavior of the component partials will be different. Thus, the algorithm requires some knowledge about the characteristics of the sound that is being analyzed.

The basic idea of the algorithm is that a set of "guides" advances in time through the spectral peaks of each frame, looking for the appropriate peaks to be used (according to the specified constraints) and forming trajectories out of them. Thus, a guide is an abstract entity employed to create sinusoidal trajectories, being the trajectories the actual result of the peak continuation process. The instantaneous state of the guides, including their frequency and magnitude, are continuously updated as the guides are turned on, continued, and finally turned off. For the case of harmonic sounds, these guides are initialized according to the harmonic series of the fundamental frequency found, and for inharmonic sounds, each guide is created dynamically.

The guides use the peak values and their context, such as surrounding peaks and fundamental frequency, to advance in time and form trajectories. For example, by using the detected fundamental frequency and the "memory" of an incoming trajectory to a given frame, we control the adaptation of the guides to the instantaneous changes in the sound. For a very harmonic sound, since all the harmonics evolve together, the fundamental should be the main control. Nevertheless, when the sound is not very harmonic and we cannot rely on the fundamental frequency as a strong reference for all the harmonics, the information of the incoming trajectory should have a bigger weight.

Each peak is assigned to the guide that is closest to it and that is within a given frequency and amplitude deviation. If a guide does not find a match, it is assumed that the corresponding trajectory must "turn off". In inharmonic sounds, if a guide has not found a continuation peak for a given amount of time the guide is killed. New guides,

and therefore new trajectories, are created from the peaks of the current frame that are not incorporated into trajectories by the existing guides. If there are killed or unused guides, a new guide can be started. Searching through the "unclaimed" peaks of the frame for the one with the highest magnitude creates a guide. Once the trajectories have been continued for a few frames, the short ones can be deleted and we can fill the "gaps" encountered in long trajectories. A real-time implementation would not be able to use the rules that use the information of "future" frames.

The creation of trajectories from the spectral peaks is compatible with very different strategies and algorithms. A promising approach is to use hidden Markov models [Depalle, Garcia and Rodet, 1993]. This type of approach might be very valuable for tracking partials in polyphonic sounds and complex inharmonic tones.

In our Matlab implementation, we have chosen to provide a simple tracking algorithm that uses a simplified version of the techniques previously introduced for the case of harmonic and inharmonic sounds:

```
function [iloc, ival, iphase, previousiloc, previousival,
distminindex] = ...
      peakTrackSimple (nSines, nPeaks, N, SR, pitchvalue, iftloc,
iftval, ...
      iftphase, isHarm, previousiloc, previousival);
%function [iloc, ival, iphase, previousiloc, previousival,
distminindex] = ...
%      peakTrackSimple (nSines, nPeaks, N, SR, pitchvalue, iftloc,
iftval, ...
%      iftphase, isHarm, previousiloc, previousival);
%
%==> simplest partial tracking
% data:
%      iloc, ival, iphase:   location (bin) magnitude & phase of
peaks (current frame)
%      previousiloc, previousival, previousiphase:    idem for
previous frame
%      iftloc, iftval, iftphase:   idem of all of the peaks in the FT
%      distminindex:   indexes of the minimum distance between iloc
and iftloc
%      nPeaks:         number of peaks detected
%      nSines:         number of peaks tracked
%      N:              size of the FFT
%      SR:             sampling rate
%      pitchvalue:     estimated pitch value
%      isHarm:         indicator of harmonicity


tmpharm = pitchvalue;   %--- temporary harmonic
iloc = zeros(nSines,1);
MindB = -100;
ival = zeros(nSines,1) + MindB;
iphase = zeros(nSines,1);
distminindex = zeros(nSines,1);
Delta = 0.01;


for i=1:nSines             %--- for each sinus detected
  if (isHarm==1)       %--- for a harmonic sound
      [closestpeakmag, closestpeakindex] = min(abs((iftloc-1)/N*SR-
tmpharm));
```

```
      tmpharm = tmpharm + pitchvalue;
  else                 %--- for an inharmonic sound
      [closestpeakmag, closestpeakindex] = min(abs(iftloc-
previousiloc(i)));
   end
   iloc(i)   = iftloc(closestpeakindex);  %--- bin of the closest
   ival(i)   = iftval(closestpeakindex);
   iphase(i) = iftphase(closestpeakindex);
   dist      = abs(previousiloc-iloc(i));
   [distminval, distminindex(i)] = min(dist);
end
```

And the following code implements a visual representation of the sinusoidal tracks:

```
function PlotTracking(SineFreq, pitch)
%function PlotTracking(SineFreq, pitch)
%
%==> plot the partial tracking
% data:
%      SineFreq: frequencies of the tracks
%      pitch:    frequency of the pitch

[nSines, nFrames] = size(SineFreq);

for n=1:nSines
   f=1;
   while (f<=nFrames)
      while (f<=nFrames & SineFreq(n,f)==0)
         f = f+1;
      end
      iStart = min(f,nFrames);
      while (f<=nFrames & SineFreq(n,f)>0)
         f = f+1;
      end
      iEnd = min(max(1,f-1),nFrames);
      if (iEnd > iStart)
        line((iStart:iEnd), SineFreq(n,iStart:iEnd));
      end
   end
end

h = line((1:nFrames), pitch(1:nFrames));
set(h,'linewidth', 2, 'Color', 'black');
```

9.3.1.5 Residual Analysis

Once we have identified the stable partials of a sound, we are ready to subtract them from the original signal and obtain the residual component. This subtraction can be done either in the time domain or in the frequency domain. A time domain approach requires to first synthesize a time signal from the sinusoidal trajectories, while if we stay in the frequency domain, we can perform the subtraction directly in the already computed magnitude spectrum. For the time domain subtraction, the phases of the original sound have to be preserved in the synthesized signal, thus we have to use a type of additive synthesis with which we can control the instantaneous phase. A type of synthesis that is computationally quite expensive. In the other hand, the sinusoidal

subtraction in the spectral domain is simpler but not much more. Our sinusoidal information from the analysis is very much under sampled, since for every sinusoid we only have the value at the tip of the peaks, and thus we have to generate all the frequency samples that belong to the sinusoidal peak to be subtracted.

Once we have, either the residual spectrum or the residual time signal, it is useful to study it in order to check how well the partials of the sound were subtracted and therefore analyzed. If partials remain in the residual, the possibilities for transformations will be reduced, mainly because it will not be possible to approximate the residual as a stochastic signal, thus reducing its flexibility. In this case, we should re-analyze the sound until we get a good residual, free of deterministic components. Ideally, the resulting residual should be as close as possible to a stochastic signal.

From the residual signal, we can continue our modeling strategy. To model the stochastic part of sounds, such as the attacks of most percussion instrument, the bow noise in string instruments, or the breath noise in wind instruments, we need a good time resolution and we can give up some frequency resolution. The deterministic component cannot maintain the sharpness of the attacks, because, even if a high frame-rate is used we are forced to use a long enough window, and this size determines most of the time resolution. When the deterministic subtraction is done in the time domain, the time resolution in the stochastic analysis can be improved by redefining the analysis window. The frequency domain approach implies that the subtraction is done in the spectra computed for the deterministic analysis, thus the STFT parameters cannot be changed [Serra, 1989].

Since it is the deterministic signal that is subtracted from the original sound, measured from long windows, the resulting residual signal might have the sharp attacks smeared. To improve the stochastic analysis, we can "fix" this residual so that the sharpness of the attacks of the original sound is preserved. The resulting residual is compared with the original waveform and its amplitude re-scaled whenever the residual has a greater energy than the original waveform. Then the stochastic analysis is performed on this scaled residual. Thus, the smaller the window the better time resolution we will get in the residual. We can also compare the synthesized deterministic signal with the original sound and whenever this signal has a greater energy than the original waveform it means that a smearing of the deterministic component has been produced. This can be fixed a bit by scaling the amplitudes of the deterministic analysis in the corresponding frame by the difference between original sound and deterministic signal.

Sinusoidal subtraction

The first step of the Residual Analysis is the synthesis of the sinusoidal tracks obtained as the output of the Peak Continuation algorithm. For a time domain subtraction the synthesized signal will reproduce the instantaneous phase and amplitude of the partials of the original sound. One frame of the sinusoidal part of the sound, $d(m)$, is generated by

$$d(m) = \sum_{r=1}^{R} \hat{A}_r \cos\left[m\hat{\omega}_r + \hat{\varphi}_r\right], \qquad m = 0,1,2,\ldots,S-1 \tag{8}$$

where R is the number of trajectories present in the current frame and S is the length of the frame. To avoid "clicks" at the frame boundaries, the parameters $(\hat{A}_r, \hat{\omega}_r, \hat{\varphi}_r)$ are smoothly interpolated from frame to frame.

The instantaneous amplitude $\hat{A}(m)$ is easily obtained by linear interpolation from frame to frame. Frequency and phase values are tied together (frequency is the phase derivative), and both control the instantaneous phase $\hat{\theta}(m)$, defined as

$$\hat{\theta}(m) = m\hat{\omega} + \hat{\varphi} \tag{9}$$

Different approaches are possible for computing the instantaneous phase [MacAulay and Quatieri, 1986], thus being able to synthesize one frame of a sound from

$$d^l(m) = \sum_{r=1}^{R^l} \hat{A}_r^l(m) \cos\left[\hat{\theta}_r^l(m)\right] \tag{10}$$

which goes smoothly from the previous to the current frame with each sinusoid accounting for both the rapid phase changes (frequency) and the slowly varying phase changes. *(describe other alternatives for obtaining instantaneous phase)*

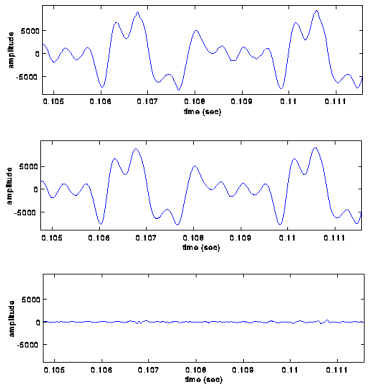


Fig 9.13. *Time domain substraction*

Stochastic Approximation

One of the underlying assumptions of the *Sinusoidal plus Residual* model is that the residual is a stochastic signal. Such an assumption implies that the residual is fully described by its amplitude and its general frequency characteristics. It is unnecessary to keep either the instantaneous phase or the exact spectral shape information. Based on this, a frame of the stochastic residual can be completely characterized by a filter, i.e., this filter encodes the amplitude and general frequency characteristics of the residual. The representation of the residual for the overall sound will be a sequence of these filters, i.e., a time-varying filter.

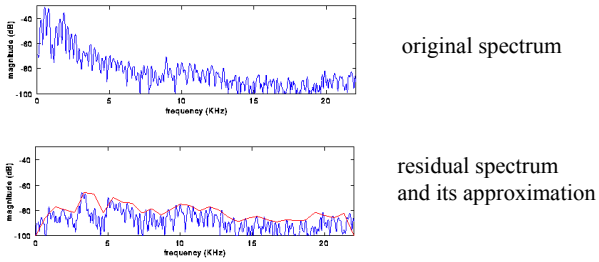


Fig 9.14. *(a)Original spectrum, (b) Residual spectrum and aprosimation*

The filter design problem is generally solved by performing some sort of curve fitting in the magnitude spectrum of the current frame [Strawn, 1980; Sedgewick, 1988]. Standard techniques are: spline interpolation [Cox, 1971], the method of least squares [Sedgewick, 1988], or straight-line approximations [Phillips, 1968].

One way to carry out the line-segment approximation is to step through the magnitude spectrum and find local maxima in each of several defined sections, thus giving equally spaced points in the spectrum that are connected by straight lines to create the spectral envelope. The number of points gives the accuracy of the fit, and that can be set depending on the sound complexity. Other options are to have unequally spaced points, for example, logarithmically spaced, or spaced according to other perceptual criteria.

Another practical alternative is to use a type of least squares approximation called linear predictive coding, LPC [Makhoul, 1975; Markel and Gray, 1976]. LPC is a popular technique used in speech research for fitting an nth-order polynomial to a magnitude spectrum. For our purposes, the line-segment approach is more flexible than LPC, and although LPC results in less analysis points, the flexibility is considered more important.

For a comprehensive collection of different approximation techniques of the residual component see [Goodwin, 1997]

9.3.2 Feature analysis

The accomplishment of a meaningful parameterization for sound transformation applications is a difficult task. We want a parameterization that offers an intuitive control over the sound transformation process, with which we can access most of perceptual attributes of a sound. The analysis techniques described so far result in a simple parameterization, appropriate for describing the lower physical characteristics of the sound. In the Sinusoidal plus Residual model, these parameters are the instantaneous frequency, amplitude and phase of each partial and the instantaneous spectral characteristics of the residual signal.

There are other useful instantaneous attributes that give a higher-level abstraction of the sound characteristics. For example we can describe fundamental frequency, amplitude and spectral shape of sinusoidal component, amplitude and spectral shape of residual component, and overall amplitude. These attributes are calculated at each analysis frame from the output of the basic Sinusoidal plus Residual analysis. Afterwards, some of them can be extracted.

From a digital effects designer point of view, the extraction of such attributes allow us to implement transformations that modify only one of those features without affecting the rest. A clear example is illustrated in Fig 9.2 where the fundamental frequency is extracted, multiplied by a scaling factor, and then incorporated back to the original spectral data.

Many other features like the degree of harmonicity, noisiness, spectral tilt, or spectral centroid, can also be computed from the spectral representation of a sound. Some of them are just information attributes that describe the characteristics of the frame and have mainly found applications in sound classification tasks. Apart from the instantaneous, or frame, values, it is also useful to have parameters that characterize the time evolution of the sound. The time changes can be described by the derivatives of each one of the instantaneous attributes.

Another important step towards a musically useful parameterization is the segmentation of a sound into regions that are homogeneous in terms of its sound attributes. Then we can identify and extract region attributes that will give higher-level control over the sound.

From the basic sinusoidal plus residual representation it is possible to extract some of the attributes mentioned above. The critical issue is how to extract them while minimizing interferences, thus obtaining significant high level attributes free of correlations [Serra and Bonada, 98]. The general process will be to first extract instantaneous attributes and their derivatives, then to segment the sound based on that information, and finally to extract region attributes.

As already indicated, the basic instantaneous attributes are: amplitude of sinusoidal and residual component, overall amplitude, fundamental frequency, spectral shape of sinusoidal and residual component, harmonic distortion, noisiness, spectral centroid, and spectral tilt. These attributes are obtained at each frame using the information that results from the basic Sinusoidal plus Residual analysis and not taking into account the data from previous or future frames. The amplitude of the sinusoidal component is the sum of the amplitudes of all harmonics of one frame expressed in dB,

$$AS = 20 \log_{10}\left(\sum_{i=1}^{I} a_i\right) \tag{11}$$

where a_i is the linear amplitude of the ith harmonic and I is the total number of harmonics found in the frame.

The amplitude of the residual component is the sum of the absolute values of the residual of one frame expressed in dB. This amplitude can also be computed by adding the frequency samples of the corresponding magnitude spectrum,

$$\begin{aligned} AR &= 20 \log_{10}\left(\sum_{n=0}^{M-1} |x_R(n)|\right) \\ &= 20 \log_{10}\left(\sum_{k=0}^{N-1} |X_R(k)|\right) \end{aligned} \tag{12}$$

where $x_R(n)$ is the residual sound, M is the size of the frame, $X_R(k)$ is the spectrum of the residual sound, and N is the size of the magnitude spectrum.

The total amplitude of the sound at one frame is the sum of its absolute values expressed in dB. It can also be computed by summing the amplitudes of the sinusoidal and residual components,

$$A = 20\log_{10}\left(\sum_{n=0}^{M-1}|x(n)|\right) = 20\log_{10}\left(\sum_{k=0}^{N-1}|X(k)|\right) \quad (13)$$
$$= 20\log_{10}\left(\sum_{i=1}^{I}a_i + \sum_{k=0}^{N-1}|X_R(k)|\right)$$

where $x(n)$ is the original sound and $X(k)$ is its spectrum.

The fundamental frequency is the frequency that best explains the harmonics of one frame. Many different algorithms can be used to compute the fundamental (see 9.3.1.3, for example) but a reasonable approximation, once we have the sinusoidal component, can be the weighted average of all the normalized harmonic frequencies,

$$F_0 = \sum_{i=1}^{I}\frac{f_i}{i}\times\frac{a_i}{\sum_{i=1}^{I}a_i} \quad (14)$$

where f_i is the frequency of the ith harmonic.

The spectral shape of the sinusoidal component is the envelope described by the amplitudes and frequencies of the harmonics, or its approximation,

$$Sshape = \{(f_1,a_1)(f_2,a_2)..(f_I,a_I)\} \quad (15)$$

The spectral shape of the residual component is an approximation of the magnitude spectrum of the residual sound at one frame. A simple function is computed as the line segment approximation of the spectrum,

$$Rshape = \{e_1, e_2, \ldots, e_q, \ldots, e_{Ncoef}\} \quad (16)$$

Other spectral approximation techniques can be considered depending on the type of residual and the application. [Goodwin,1996]

The frame-to-frame variation of each attribute is a useful measure of its time evolution, thus an indication of changes in the sound. It is computed in the same way for each attribute,

$$\Delta = \frac{Val(l) - Val(l-1)}{H/SR} \quad (17)$$

where $Val(l)$ is the attribute value for the current frame, $Val(l-1)$ is the attribute value for the previous one, H is the hop-size and SR the sampling rate.

9.3.2.1 Segmentation

Sound segmentation has proven important in automatic speech recognition and music transcription algorithms. For our purposes it is very valuable as a way to apply region dependent transformations. For example, a time stretching algorithm would be able to transform the steady state regions, leaving the rest unmodified.

A musically meaningful segmentation process divides a melody into notes and silences and then each note into an attack, a steady state and a release regionsThe techniques originally developed for speech [Vidal and Marzal, 1990], such as those based on Pattern-Recognition or Knowledge-Based methodologies, start to be used in music segmentation applications [Rossignol and others 1998]. Most of the approaches apply classification methods that start from sounds features, such as the ones described in this paper, and are able to group sequences of frames into predefined categories. No reliable and general-purpose technique has been found. Our experience is that they require narrowing the problem to a specific type of musical signal or including a user intervention stage to help direct the segmentation process.

9.3.2.2 Region attributes

Once a given sound has been segmented into regions we can compute the attributes that describe each one. Most of the interesting attributes are the mean and variance of each of the frame attributes for the whole region. For example, we can compute the spectral shape or the mean and variance for the amplitude of sinusoidal and residual components, the fundamental frequency, or the spectral tilt.

Global attributes that can characterize attacks and releases make use of the average variation of each of the instantaneous attributes, such as average fundamental frequency variation, average amplitude variation, or average spectral shape change. In the steady state regions it is important to extract the average value of each of the instantaneous attributes and measure other global attributes such as time-varying rate and depth of vibrato. Vibrato is a specific attribute present in many steady state regions of sustained instrumental sounds that requires a special treatment [Herrera, 1998].

Some region attributes can be extracted from the frame attributes in the same way that these were extracted from the Sinusoidal plus Residual data. The result of the extraction of the frame and region attributes is a hierarchical multi-level data structure where each level represents a different sound abstraction.

9.3.3 Synthesis

From the output of the analysis techniques presented we can synthesize a new sound. The similarity with respect to the original sound will depend on how well the input sound fits the implicit model of the analysis technique and the settings of the different variables that the given technique has. In the context of the chapter we are interested in transforming the analysis output in order to produce a specified effect in the synthesized sound.

All these transformations can be done in the frequency domain. Afterwards, the output sound can be synthesized using the techniques presented in this section. The

sinusoidal component will be generated using some type of additive synthesis approach and the residual, if present, will be synthesized using some type of subtractive synthesis approach.

Thus, the transformation and synthesis of a sound is done in the frequency domain; generating sinusoids, noise, or arbitrary spectral components, and adding them all to a spectral frame. Then, we compute a single IFFT for each frame, which can yield efficient implementations.

Fig 9.15 shows a block diagram of the final part of the synthesis process. Previous to that we have to transform and add all the *High Level Features*, if they have been extracted, and obtain the lower level data (sine and residual) for the frame to be synthesized. Since the stored data might have a different frame rate, or a variable one, we also have to generate the appropriate frame by interpolating the stored ones. These techniques are presented in the following sections.

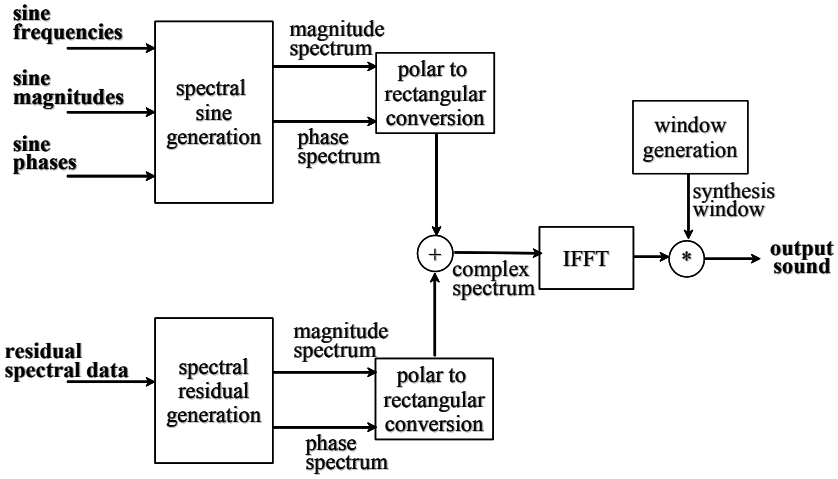


Fig 9.15. *Diagram of the spectral synthesis.*

9.3.3.1 Sinusoidal synthesis

The sinusoidal component is generated with additive synthesis, similar to the sinusoidal synthesis that was part of the analysis, with the difference that now the phase trajectories might be discarded.

Additive synthesis is based in the control of the instantaneous frequency and amplitude of a bank of oscillators as shown in the following figure:

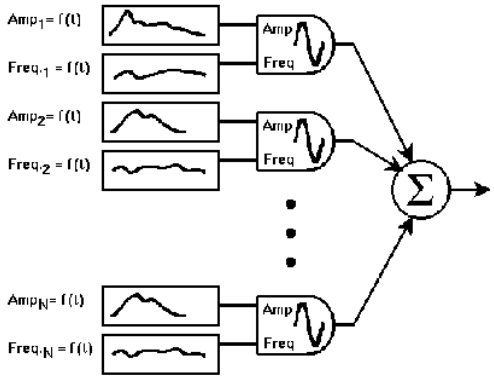


Fig 9.16. *Additive Synthesis Block Diagram*

The instantaneous amplitude $\hat{A}(m)$ of an oscillator is obtained by linear interpolation,

$$\hat{A}(m) = \hat{A}^{l-1} + \frac{(\hat{A}^{l} - \hat{A}^{l-1})}{S} m \qquad (18)$$

where $m = 0,1,\ldots,S-1$ is the time sample in the l^{th} synthesis frame.

The instantaneous phase is taken to be the integral of the instantaneous frequency, where the instantaneous radian frequency $\hat{\omega}(m)$ is obtained by linear interpolation,

$$\hat{\omega}(m) = \hat{\omega}^{l-1} + \frac{\left(\hat{\omega}^{l} - \hat{\omega}^{l-1}\right)}{S} m \qquad (19)$$

and the instantaneous phase for the rth sinusoid is

$$\hat{\theta}_r(m) = \hat{\theta}_r(l-1) + \hat{\omega}_r(m) \qquad (20)$$

Finally, the synthesis equation becomes

$$d^{l}(m) = \sum_{r=1}^{R^{l}} \hat{A}_r^{l}(m) \cos\left[\hat{\theta}_r^{l}(m)\right] \qquad (21)$$

where $\hat{A}(m)$ and $\hat{\theta}(m)$ are the calculated instantaneous amplitude and phase.

A very efficient implementation of additive synthesis, when the instantaneous phase is not preserved, is based on the inverse-FFT [Rodet and Depalle, 1992; Goodwin and Rodet, 1994]. While this approach looses some of the flexibility of the traditional oscillator bank implementation, especially the instantaneous control of frequency and magnitude, the gain in speed is significant. This gain is based on the fact that a sinusoid in the frequency domain is a sinc-type function, the transform of the window used, and on these functions not all the samples carry the same weight. To generate a sinusoid in the spectral domain it is sufficient to calculate the samples of the main lobe of the window transform, with the appropriate magnitude, frequency and phase values. We can then synthesize as many sinusoids as we want by adding these main lobes in the FFT buffer and performing an IFFT to obtain the resulting time-domain signal. By an overlap-add process we then get the time-varying characteristics of the sound.

In the following Matlab code we implement a sinusoidal synthesis algorithm based on this latter approach:

```
function padsynthft = sinefillspectrum(iloc, ival, iphase, nSines,
...
   w1Length, zp, bh92SINC, bh92SINCsize)
%function padsynthft = sinefillspectrum(iloc, ival, iphase, nSines,
...
%   w1Length, zp, bh92SINC, bh92SINCsize)
%
%==> compute the spectrum of all the sines in the frequential domain,
%    in order to remove it from the signal
% data:
%      padsynth:
%      iloc, ival, iphase: location (bin), magnitude value (dB) and
phase of a peak
```

```
%      nSines:         number of sines (=length of ival and iloc)
%      w1Length:       size of the analysis window
%      zp:             zero-padding multiplicative coefficient
%      bh92SINC:       Blackman-Harris window
%      bh92SINCsize:   Blackman-Harris window size


peakmag         = 10.^(ival/20);            % magnitude (in [0;1])
halflobe        = 8*zp/2-1;                 % bin number of the half
lobe
firstbin        = floor(iloc)-halflobe;     % first bin for filling
positive frequencies
firstbin2       = floor(w1Length*zp-iloc+2)-halflobe; % idem for
negative frequencies
binremainder    = iloc-floor(iloc);
sinphase        = sin(iphase);
cosphase        = cos(iphase);
findex          = 1-binremainder;
bh92SINCindexes = zeros(8*zp,1);
sinepadsynthft  = zeros(w1Length*zp+halflobe+halflobe+1,1);
padsynthft = zeros(w1Length*zp,1);

%--- computation of the complex value
for i=1:nSines   %--- for each sine
   if (iloc(i)~=0) %--- JUST WORK WITH NON ZEROS VALUES OF iloc !!! -
> tracked sines
      beginindex = floor(0.5 + findex(i)*512/zp)+1;
      bh92SINCindexes = [beginindex:512/zp:beginindex+512/zp*(8*zp-
1)]';
      if (bh92SINCindexes(8*zp) > bh92SINCsize)
         bh92SINCindexes(8*zp) = bh92SINCsize;
      end
      magsin = bh92SINC(bh92SINCindexes).*sinphase(i)*peakmag(i);
      magcos = bh92SINC(bh92SINCindexes).*cosphase(i)*peakmag(i);
      %--- fill positive frequency
      sinepadsynthft(firstbin(i)+halflobe:firstbin(i)+halflobe+8*zp-
1) = ...

sinepadsynthft(firstbin(i)+halflobe:firstbin(i)+halflobe+8*zp-1) +
(magcos+j*magsin);
      %--- fill negative frequency
      if (firstbin2(i)+halflobe <= w1Length*zp)

sinepadsynthft(firstbin2(i)+halflobe:firstbin2(i)+halflobe+8*zp-1) =
...

sinepadsynthft(firstbin2(i)+halflobe:firstbin2(i)+halflobe+8*zp-1) +
(magcos-j*magsin);
      end
   end
end

%--- fill padsynthft
padsynthft = padsynthft +
sinepadsynthft(halflobe+1:halflobe+1+w1Length*zp-1);
padsynthft(1:halflobe) = padsynthft(1:halflobe) + ...
   sinepadsynthft(w1Length*zp+1:w1Length*zp+halflobe);
padsynthft(w1Length*zp-halflobe+1:w1Length*zp) = ...
   padsynthft(w1Length*zp-halflobe+1:w1Length*zp) +
sinepadsynthft(1:halflobe);
```

The synthesis frame rate is completely independent of the analysis one. In the implementation using the IFFT we want to have a frame rate high enough so to preserve the temporal characteristics of the sound. As in all short-time based processes we have the problem of having to make a compromise between time and frequency resolution. The window transform should have the fewest possible significant bins since this will be the number of points to generate per sinusoid. A good window choice is the Blackman-Harris 92dB because, as already explained in 9.3.1.1, its main lobe includes most of the energy. However the problem is that such a window does not overlap perfectly to a constant in the time domain without having to use very high overlap factors, thus very high frame rates. A solution to this problem [Rodet and Depalle, 1992] is to undo the effect of the window by dividing by it in the time domain and applying a triangular window before performing the overlap-add process. This will give a good time-frequency compromise.

Here we provide the Matlab code for generating the triangular window:

```
function w = triang(n)
%TRIANG Triangular window.
if rem(n,2)
  % It's an odd length sequence
  w = 2*(1:(n+1)/2)/(n+1);
  w = [w w((n-1)/2:-1:1)]';
else
  % It's even
  w = (2*(1:(n+1)/2)-1)/n;
  w = [w w(n/2:-1:1)]';
end
```

9.3.3.2 Residual synthesis

The synthesis of the residual component of the sound is also done in the frequency domain. When the analyzed residual has not been approximated, i.e. it is represented as a magnitude and phase spectrum for each frame, a STFT, each residual spectrum is added to the spectrum of the sinusoidal component at each frame. But when a magnitude spectral envelope has approximated the residual, an appropriate complex spectrum has to be generated.

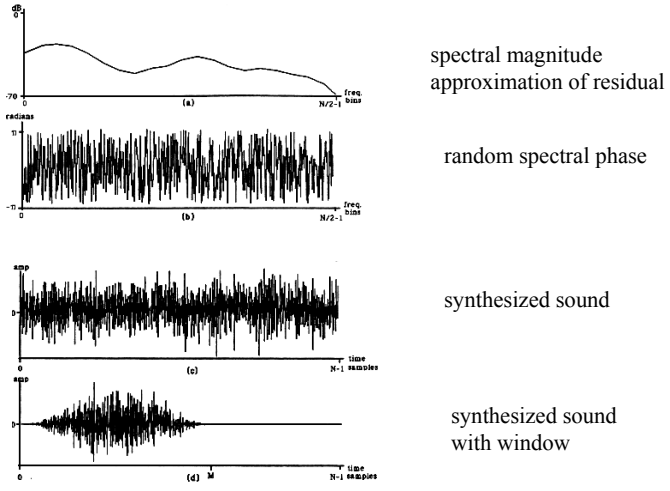


Fig 9.17. *Residual synthesis from approximated spectrum*

The synthesis of a stochastic signal from the residual approximation can be understood as the generation of noise that has the frequency and amplitude characteristics described by the spectral magnitude envelopes. The intuitive operation is to filter white noise with these frequency envelopes, that is, performing a time-varying filtering of white noise, which is generally implemented by the time-domain convolution of white noise with the impulse response corresponding to the spectral envelope of a frame. We do it in the frequency domain by creating a magnitude spectrum from the approximated one, or its transformation, and generating a random phase spectrum with new values at each frame in order to avoid periodicity.

9.3.3.3 Integration of Sinusoidal and Residual synthesis

Once the two spectral components are generated, to add the spectrum of the residual component to that of the sinusoids, we need to worry about windows. In the process of generating the noise spectrum there has not been any window applied, since the data was added directly into the spectrum without any smoothing consideration, but in the sinusoidal synthesis we have used a Blackman-Harris 92dB, which is undone in the time domain after the IFFT. Therefore we should apply the same window in the noise spectrum before adding it to the sinusoidal spectrum. Convolving the transform of the Blackman-Harris 92dB by the noise spectrum accomplishes this, and there is only need to use the main lobe of the window since it includes most of its energy,. This is implemented quite efficiently because it only involves a few bins and the window is symmetric. Then we can use a single IFFT for the combined spectrum. Finally in the time domain we undo the effect of the Blackman-Harris 92dB and impose the triangular window. By an overlap-add process we combine successive frames to get the time-varying characteristics of the sound.

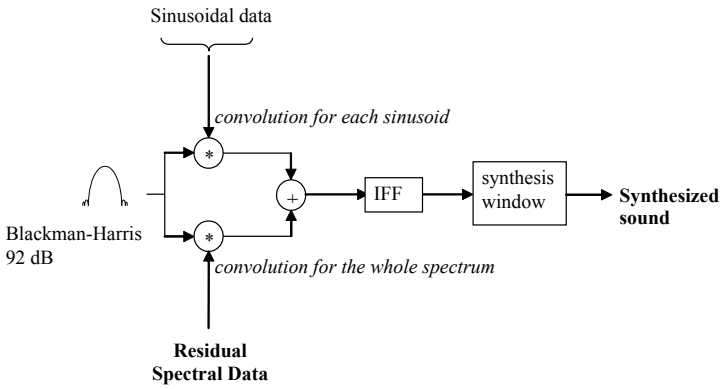


Fig 9.18. *Integrating sinusoidal plus residual synthesis*

Several other approaches have been used for synthesizing the output of a sinusoidal plus residual analysis. These techniques, though, include modifications to the model as a whole (see [Fitz, K.; L. Haken, and P. Christensen. 2000], for example).

9.3.4 Main analysis-synthesis application

In the following Matlab code, we make use of all the previous functions in order to implement a complete analysis-synthesis process. We will use this framework to implement sound effects in the next sections.

Note that no residual approximation is used in this implementation.

```
%=============================================================
% SMS-Matlab like emulation
%
%=============================================================

clear all
close all

%==== USER DATA =====
DAFx_in  = wavread('love.wav'); % wave file
SR       = 44100;  % sampling rate
w1Length = 2048;   % analysis window size
n1       = 256;    % analysis window hop size
nPeaks   = 100;    % number of peaks detected
nSines   = 50;     % number of sinuosoids to track (and synthetise)
minSpacePeaks = 2; % minimum space (bins) between two picked peaks
zp       = 2;      % zero-padding coefficient
rgain    = 1.;     % gain for the residual component
MaxFreq  = 11000;  % maximum frequency, in Hertz, for plottings
MinMag   = -100;   % minimum magnitude, in dB, for plottings

%--- figure data
%fig1 = 'yes';     % if uncommented, will plot the Blackman-Harris
window
%fig2 = 'yes';  % if uncommented, will plot the peaks detection and
tracking in one frame
%fig3 = 'yes';  % if uncommented, will plot the peak trackings real
time
%fig4 = 'yes';  % if uncommented, will plot the original and the
transformed FT in one frame
%fig5 = 'yes';  % if uncommented, will plot the peak trackings only
at the end of the process
%fig6 = 'yes';  % if uncommented, will plot the original signal, its
sine and residual part,
               % and the transformed signal


%=== Definition of the Windows ===

%--- definition of the analysis window
fConst = 2*pi/(w1Length+1-1);
w1=[1:w1Length]';
w1=.35875 - .48829*cos(fConst*w1) + .14128*cos(fConst*2*w1) -
.01168*cos(fConst*3*w1);
w1=w1/sum(w1)*2;
N = w1Length*zp;        % new size of the window
%--- synthesis window
w2=w1;
n2=n1;
%--- triangular window
wt2=triang(n2*2+1); % triangular window
%--- main lobe table of bh92
[bh92SINC, bh92SINCsize] = bh92SINCgeneration;
%--- data for the loops
```

```
frametime = n1/SR;
pin  = 0;
pout = 0;
TuneLength = length(DAFx_in);
pend = TuneLength - w1Length;

%=== Definition of the data arrays ===

DAFx_in      = [zeros(w1Length/2-n1-1,1); DAFx_in];
DAFx_outsine = zeros(TuneLength,1);
DAFx_outres  = zeros(TuneLength,1);
%--- arrays for the partial tracking
iloc            = zeros(nSines,1);
ival            = zeros(nSines,1);
iphase          = zeros(nSines,1);
previousiloc    = zeros(nSines,1);
previousival    = zeros(nSines,1);
maxSines = 400; % maximum voices for harmonizer
syniloc                  = zeros(maxSines,1);
synival                  = zeros(maxSines,1);
previoussyniloc = zeros(maxSines,1);
previousiphase  = zeros(maxSines,1);
currentiphase   = zeros(maxSines,1);
%--- arrays for the sinus' frequencies and amplitudes
SineFreq = zeros(nSines,ceil(TuneLength/n2));
SineAmp  = zeros(nSines,ceil(TuneLength/n2));
pitch = zeros(1,1+ceil(pend/n1));
pitcherr = zeros(1,1+ceil(pend/n1));

%--- creating figures ---
if(exist('fig1'))
   h = figure(1); set(h,'position', [10, 45, 200, 200]);
end
if(exist('fig2'))
   h = figure(2); set(h,'position', [10, 320, 450, 350]);
  axisFig2 = [0  MaxFreq MinMag  0]; zoom on;
end
if(exist('fig3'))
   h = figure(3); set(h,'position', [220, 45, 550, 200]);
   axisFig3 = [1  1+ceil(pend/n1) 0  MaxFreq]; zoom on;
end
if(exist('fig4'))
   h = figure(4); set(h,'position', [470, 320, 450, 350]);
   axisFig4 = [0  MaxFreq MinMag  0]; zoom on;
end
if(exist('fig5'))
   h = figure(5); set(h,'position', [220, 45, 550, 200]);
   axisFig5 = [1  1+ceil(pend/n1) 0  MaxFreq]; zoom on;
end

%--- plot the Blackman-Harris window
if(exist('fig1'))
  figure(1)
  plot(20*log10(abs(fftshift(fft(bh92SINC)/bh92SINCsize))))
  title('Blackman-Harris window'); xlabel('Samples');
ylabel('Amplitude')
end


tic
```

```
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
disp('analazing frame ...');
while pin<pend

   %--- windowing
   grain = DAFx_in(pin+1:pin+w1Length).*w1(1:w1Length);

   %--- zero padding
   padgrain = zeros(N,1);
   padgrain(1:w1Length/2)     = grain(w1Length/2+1:w1Length);
   padgrain(N-w1Length/2+1:N) = grain(1:w1Length/2);

   %--- fft computation
   f   = fft(padgrain);
   r   = abs(f);
   phi = angle(f);
   ft  = r.*exp(j*phi);

   %===== Analysis =====

   %--- peak detection (and their plottings)
   [ftloc, ftval] = PickPeaks(r(1:N/2), nPeaks, minSpacePeaks);

   %--- calculate interpolated values (peak position, phase,
amplitude)
   [iftloc, iftphase, iftval] = ...
      interpolatedValues (r, phi, N, zp, ftloc, ftval);

   %--- pitch detection
   [pitchvalue, pitcherror, isHarm] = ...
      pitchDetection (r, N, SR, nPeaks, iftloc, iftval);
   pitch(1+pin/n1) = pitchvalue * isHarm;
   pitcherr(1+pin/n1) = pitcherror;

  %--- peaks tracking
   if (pin==0)    %--- for the first frame
      nNewPeaks = nSines;
   else                 %--- creating new born tracks
      for i=1:nSines
         if (previousiloc(i)==0)
            [previousiloc(i), previousival(i)] = CreateNewTrack ...
               (iftloc, iftval, previousiloc, previousival, nSines,
MinMag);
            nNewPeaks = nNewPeaks - 1;
        end
     end

    %--- simple Peak tracker
     [iloc, ival, iphase, previousiloc, previousival, distminindex] =
...
       peakTrackSimple (nSines, nPeaks, N, SR, pitchvalue, iftloc,
iftval, ...
       iftphase, isHarm, previousiloc, previousival);
   end

   %--- savings
   previousival = ival;
  previousiloc = iloc;
   SineFreq(:,1+pin/n1) = max((iloc-1)/N*SR,0.);  % frequency of the
partials
```

```
   SineAmp(:,1+pin/n1)  = max(ival, MinMag);         % amplitudes of
the partials

   syniloc(1:nSines) = max(1,iloc);
   synival(1:nSines) = ival;

   if(exist('fig3'))                              % plot: the trackings
of partials
      figure(3); clf; hold on
      PlotTracking(SineFreq(:,1:1+pin/n1), pitch(1:1+pin/n1));
      xlabel('Frame number'); ylabel('Frequency (Hz)');
axis(axisFig3);
    title('Peak tracking'); drawnow
  end

   %--- residual computation
   resfft = ft;
   if(isHarm==1)
      resfft = resfft - sineFillSpectrum(iloc, ival, iphase,
nSines,...
         w1Length, zp, bh92SINC, bh92SINCsize);
   end

   %--- figures
  if(exist('fig2'))
     figure(2); clf; hold on  % plot: the FFT of the windowed signal
(Hz, dB)
    plot((1:N/2)/N*SR, 20*log10(r(1:N/2)));
      for l=1:nPeaks           % plot: the peaks detected
         plot([ftloc(l)-1 ftloc(l)-1]/N*SR, [20*log10(ftval(l)),
MinMag-1],'r:x');
    end
     for l=1:nSines            % plot: the sines tracked and the
residual part
       plot([iloc(l)-1, iloc(l)-1]/N*SR, [ival(l), MinMag-1],'k')
     end
      plot((1:N/2)/N*SR, 20*log10(abs(resfft(1:N/2))),'g');
      if(isHarm)                % plot: the true pitch of each
harmonic
         for l=1:nSines
            plot([pitchvalue*l, pitchvalue*l],[1, MinMag-1],'y:')
         end
      end
      xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');
axis(axisFig2);
    title('Peak detection and tracking for one frame'); drawnow
   end


nSynSines = nSines;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%===== Transformations =====

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%===== Synthesis =====

   %--- phase computation
```

```
   if (pin > 0)
      for i=1:nSynSines
            if (syniloc(i)~=0)
           ifreq = (previoussyniloc(distminindex(i))+ syniloc(i))/2;
% average bin
           freq = (ifreq-1)/N*SR; % freq in hz (if loc=1 --> freq=0)
           currentiphase(i) =
unwrap2pi(previousiphase(distminindex(i))+...
              2*pi*freq*frametime);
       end
      end
   end

   previoussynival = synival;
   previoussyniloc = syniloc;
   previousiphase  = currentiphase;

   %--- compute sine spectrum
  padsynthft = sinefillspectrum(syniloc, synival, currentiphase,
nSynSines, ...
         w1Length, zp, bh92SINC, bh92SINCsize);
   if (isHarm==0)
      padsynthft = zeros(size(padsynthft));
   end

  %--- residual computation
  respadgrain = real(ifft(resfft));
   resgrain      = [respadgrain(N-w1Length/2+1:N);
respadgrain(1:w1Length/2)] ./ w2(1:w1Length);
   ressynthgrain = wt2(1:n2*2) .* resgrain(w1Length/2-
n2:w1Length/2+n2-1);
  DAFx_outres(pout+1:pout+n2*2) = DAFx_outres(pout+1:pout+n2*2) +
ressynthgrain;

   %--- sinusoidal computation
   sinpadgrain   = real(ifft(padsynthft));
   singrain      = [sinpadgrain(N-w1Length/2+1:N);
sinpadgrain(1:w1Length/2)] ./ w2(1:w1Length);
   sinsynthgrain = wt2(1:n2*2) .* singrain(w1Length/2-
n2:w1Length/2+n2-1);
   DAFx_outsine(pout+1:pout+n2*2) = DAFx_outsine(pout+1:pout+n2*2) +
sinsynthgrain;

   %--- figure with original signal and transformed signal FFT
   synthr = abs(fft(respadgrain + sinpadgrain));
   if(exist('fig4'))
     figure(4); clf; hold on
      plot((1:N/2)/N*SR, 20*log10(r(1:N/2)),'b:'); axis(axisFig4);
      plot((1:N/2)/N*SR, 20*log10(synthr(1:N/2)),'r');
      figure(4); xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');
axis(axisFig4);
    title('FFT of the original (blue) and the transformed (red)
signals'); drawnow
   end

   %--- increment loop indexes
   pin  = pin + n1;
   pout = pout + n2;

   disp(pin/n1);
```

```
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU

toc

%===== write output sounds =====
DAFx_in = DAFx_in(w1Length/2-n1:length(DAFx_in));  % remove the zeros
added for the process
DAFx_outresynth = DAFx_outsine(1:TuneLength) +
rgain*DAFx_outres(1:TuneLength);
mm = max(abs(DAFx_outresynth));
wavwrite(DAFx_outresynth/mm, SR, 'DAFx_out.wav');
wavwrite(DAFx_outsine/mm, SR, 'DAFx_outsine.wav');
wavwrite(DAFx_outres/mm, SR ,'DAFx_outres.wav');


if(exist('fig3')==0 & exist('fig5')) % plot: trackings of partials
only at the end of the process
   figure(5); clf; hold on
   PlotTracking(SineFreq(:,1:1+pend/n1), pitch(1:1+pend/n1));
   xlabel('Frame number'); ylabel('Frequency (Hz)'); axis(axisFig5);

   title('Peak tracking'); drawnow
end

if(exist('fig6')) % plot the input signal, its sinus and its residual
part,
                  % and the transformed signal
   figure(6)
  subplot(4,1,1); plot(DAFx_in); xlabel('input signal');
  subplot(4,1,2); plot(DAFx_outsine);  xlabel('sinus part');
  subplot(4,1,3); plot(DAFx_outres);xlabel('residual part');
  subplot(4,1,4); plot(DAFx_outresynth);  xlabel('resynthetized
signal');
end
```

9.4 FX and Transformations

In this section we introduce a set of effects and transformations based on the analysis-synthesis framework introduced throughout this chapter. All of them are accompanied by their corresponding Matlab code. In order to use them, you just have to add the code of the effect to use under the "`===== Transformation =====`" line in the main analysis-synthesis application code of the previous section.

9.4.1 Filtering with arbitrary resolution

Filters are probably the paradigm of a "classical" effect. Many different implementations are provided in the general DSP literature and in the previous chapters of this book. Here we introduce a different approach that differs in many aspects from the classical one.

For our "filter" implementation, we take advantage of the sinusoidal plus residual model in order to modify the amplitude of any arbitrary partial present in the sinusoidal component.

For example, we can implement a band-pass filter defined by (x,y) points where x is the frequency value in Hertzs and y is the amplitude factor to apply. In the example code given below, we define a band pass filter which supreses frequencies not included in the range [2100 3000]

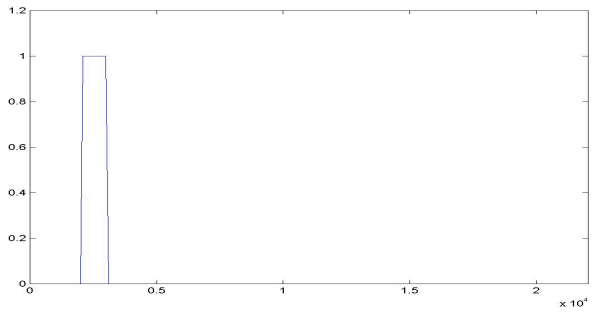


Fig 9.19. *Band-pass filter with arbitrary resolution*

```
%===== Filtering with arbitrary resoultion =====
Filter=[ 0  2099 2100 3000 3001 22050 ;
         0  0    1    1    0    0 ];
[syniloc,ind] = sort(iloc);
FilterEnvelope = interp1(Filter(1,:)',Filter(2,:)',syniloc/N*SR);
synival = ival(ind)+(20*log10(max(FilterEnvelope,10^-9)));
synival(ind)=synival;
syniloc(ind)=syniloc;
```

As shown, our filter does not need to be characterized by a traditional transfer function, and a more complex filter can be defined by summing delta-functions. For example: the following code filters out the even partials of the input sound. If applied to a sound with a broadband spectrum, like a vocal sound, it will convert it to a clarinet-like sound.

```
%=== voice to clarinet ===
syniloc = iloc;
synival = ival;
if (isHarm == 1)
      pitchvalue
      for i=1:nSines
            harmNum = round(((iloc(i)-1)/w1Length*SR/2)/pitchvalue);
            if (mod(harmNum,2)==0)  % case of an even harmonic number
                  synival(i) = -100;
            end
      end
end
```

9.4.2 Partial dependent frequency scaling

In a similar way, we can apply a frequency scaling to the sinusoidal components of our modeled sound. In that way, we can transpose all the partials in the spectrum or reproduce pseudo-inharmonicities like the higher partials frequency streching characteristical in a piano sound.

In this first example we introduce a frequency shift factor to all the partials of our sound. Note, though, that if a constant is added to every partial of a harmonic spectrum, the resulting sound will be inharmonic.

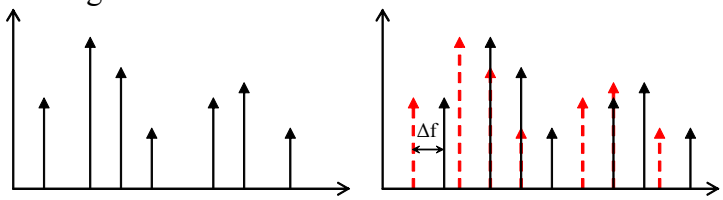


Fig 9.20. *Frequency shift of the partials*

```
%==== Frequency Shift =====
fstretch = 300;  % frequency shift in Hz
syniloc = iloc + round(fstretch/SR*N);
syniloc = syniloc.*(syniloc<=N/2);
```

Another effect we can implement following this same idea is to add a stretching factor to the frequency of every partial. The relative shift of every partial will depend on its original partial index, following the formula:

$$f_i = f_i \cdot fstretch^{(i-1)} \tag{22}$$

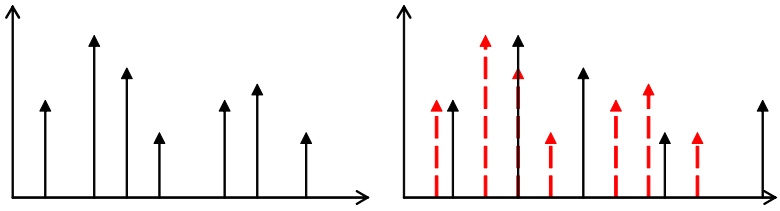


Fig 9.21. *Frequency stretch*

```
%===== Frequency Stretch  =====
fstretch = 1.1;
[syniloc,ind] = sort(iloc);
syniloc = syniloc.*((fstretch).^[0:nSines-1]');
syniloc = syniloc.*(syniloc<=N/2);
```

In the same way, we can scale all the partials multiplying them by a given scaling factor. Note that this effect will act as a pitch shifter without timbre preservation (read next section for more details on how to avoid this sometimes undesirable effect).

```
%==== Frequency Scale =====
fscale = 1.6;  % frequency scaling factor
syniloc = iloc * fscale;
syniloc = syniloc.*(syniloc<=N/2);
```

9.4.3 Pitch Transposition with Timbre Preservation

As introduced in chapter 8, pitch transposition is the scaling of all the partials of a sound by the same multiplying factor. Here we introduce the concept of timbre preservation by leaving the spectral shape unmodified. For that reason we scale the frequency of each partial applying the original spectral shape.

```
%===== Pitch transposition with timbre preservation =====
if (isHarm == 1)
   pt = 2.;                 % pitch transposition factor
   [spectralShape, shapePos] = CalculateSpectralShape(iloc, ival,
MinMag, N);
[syniloc, synival] = PitchTransposition(iloc, ival, spectralShape,
shapePos, pt, N, MinMag, nSines);
   %--- comb filtering the residual
   CombCoef = 1;
   if (isHarm==1)
      resfft = combFilter(resfft, N, SR/(pitchvalue*pt), CombCoef);
   end
end
```

where *PitchTransposition* is the following function:

```
%===== Pitch Transposition =====
function [syniloc, synival] = PitchTransposition(iloc, ival,
spectralShape, shapePos, pt, N)
syniloc = iloc(i)*pt;
syniloc = syniloc.*(syniloc<=N/2);
%--- linear interpolation of the spectral shape for the synival
computation
if shapePos > 1
   synival =
interp1(spectralShape(1,:)',spectralShape(2,:)',syniloc);
else
   synival = ival;
end
```

9.4.3.1 Pitch Discretization to Temperate Scale

An interesting effect can be accomplished by forcing the pitch to take the nearest frequency value of the temperate scale. It is indeed a very particular case of pitch transposition where the pitch is cuantified to one of the 12 semitones in which an octave is divided. This effect is widely used on vocal sounds for *dance* music and is many times referred to with the misleading name of *vocoder effect*.

```
%===== Pitch discretization to temperate scale =====
if (pitchvalue ~= 0)
  nst = round(12*log(pitchvalue/55)/log(2));
  discpitch = 55*((2^(1/12))^nst);           % discretized pitch
  pt = discpitch/pitchvalue ;            % pitch transposition factor
  [spectralShape, shapePos] = CalculateSpectralShape(iloc, ival,
MinMag, N);
  [syniloc, synival] = PitchTransposition(iloc, ival, spectralShape,
shapePos, pt, N);
  %--- comb filtering the residual
  CombCoef = 1;
  if (isHarm==1)
    resfft = combFilter(resfft, N, SR/(pitchvalue*pt), CombCoef);
```

```
  end
end;
```

9.4.4 Vibrato and Tremolo

Vibrato and tremolo are common effects used with different kinds of acoustical instruments, including the human voice. Both are low frequency modulations: vibrato is applied to the frequency and tremolo to the amplitude of the partials. Note, though, that in this particular implementation, both effects share the same modulation frequency.

```
%===== vibrato and tremolo =====
if (isHarm == 1)
  vtf = 5;       % vibrato-tremolo frequency in Hz
  va  = 10;     % vibrato depth in percentil
  td  = 3;       % tremolo depth in dB
  synival = ival + td*sin(2*pi*vtf*pin/SR);   % tremolo
  pt = 1 + va/200*sin(2*pi*vtf*pin/SR);    % pitch transposition
factor
  [spectralShape, shapePos] = CalculateSpectralShape(iloc, ival,
MinMag, N);
  [syniloc, synival] = PitchTransposition(iloc, ival, spectralShape,
shapePos, pt, N);
    %--- comb filtering the residual
    CombCoef = 1;
    resfft = combFilter(resfft, N, SR/(pitchvalue*pt), CombCoef);
end
```

9.4.5 Spectral Shape Shift

Many interesting effects can be accomplished by shifting the spectral shape or spectral envelope of the sinusoidal component of a sound. This shift is performed in such a way that no new partials are generated, just the amplitude envelope of the spectrum is modified (see Fig 9.22). In the following code we implement a shift of the spectral envelope by just modifying the amplitude of the partials according to the values of the shifted version of the spectral shape.

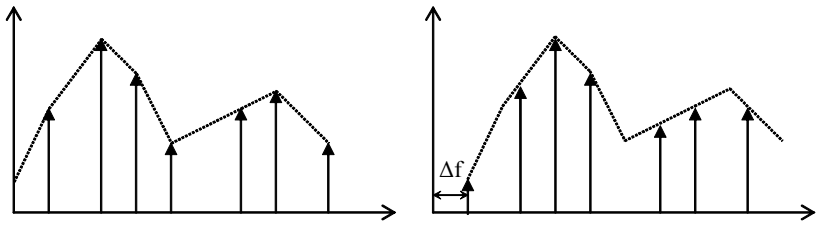


Fig 9.22. *Spectral shape shift of value Δf*

```
%===== Spectral Shape Shift (positive or negative) =====
  sss  = -200;               % spectral shape shift value in Hz
  %--- spectral shape computation
```

```
  [spectralShape, shapePos] = CalculateSpectralShape(iloc, ival,
MinMag, N);
  %--- spectral shape shift
  syniloc = zeros(nSines,1);
  if shapePos > 1
    [shiftedSpectralShape,shapePos] = SpectralShapeShift(sss, iloc,
ival, spectralShape, shapePos, N, SR);
  end
  syniloc = iloc;
  %--- linear interpolation of the spectral shape for the synival
computation
  if shapePos > 1
    synival = interp1(shiftedSpectralShape(1,1:shapePos+1)',
shiftedSpectralShape(2,1:shapePos+1)', syniloc, 'linear');
  else
    synival = ival;
  end
```

where the function *SpectralShapeShift* is implemented as follows:

```
%===== Spectral Shape Shift =====
function [shiftedSpectralShape,shapePos] = SpectralShapeShift(sss,
iloc, ival, spectralShape, shapePos, N, SR)
shiftedSpectralShape = spectralShape;
sssn = round (sss*N/SR);   % spectral shape shift in number of bins
if sssn > 0
   shiftedSpectralShape(1,2:shapePos) =
min(N/2,spectralShape(1,2:shapePos) + sssn);
   for i=shapePos:-1:1
      if shiftedSpectralShape(1,i) < N/2
         shapePos = i;
         break;
      end;
   end;
else
   shiftedSpectralShape(1,2:shapePos) =
max(1,spectralShape(1,2:shapePos) + sssn);
   for i=1:shapePos
      if shiftedSpectralShape(1,i) > 1
         shiftedSpectralShape(1,2:2+shapePos+1-i) =
shiftedSpectralShape(1,i:shapePos+1);
         shapePos = shapePos-(i-2);
         break;
      end;
   end;
end;
```

9.4.6 Gender Change

Using the results of 9.4.3 and 9.4.5 we can change the gender of a given vocal sound. Note how by combining different “basic” effects we are able to step higher in the level of abstraction and get closer to what a naive user could ask for in a sound transformation environment, such as: imagine having a gender control on a vocal processor...

In this implementation, we apply two transformations in order to convert a male voice into a female one. The first one is a pitch transposition an octave higher. The other

one is a shift in the spectral shape. The theoretical explanation to this effect is that women change their formant (resonant filters) frequencies depending on the pitch. That is, when a female singer rises up the pitch, the formants move along with the fundamental.

To convert a female into a male voice we also apply a pitch transposition and a shift in the spectral shape. This shifting has to be applied in a way the formants of the female voice remain stable along different pitches.

```
%===== gender change: man to woman =====
if (isHarm == 1)
  pitchmin=100;
  pitchmax=500;
  sssmax = 50;
  if (pitchvalue<pitchmin)
      sss = 0;
  elseif (pitchvalue>pitchmax)
    sss = sssmax;
  else
    sss = (pitchvalue-pitchmin)/((pitchmax-pitchmin)/sssmax);
  end
  %--- spectral shape computation
  [spectralShape, shapePos] = CalculateSpectralShape(iloc, ival,
MinMag, N);
  %--- spectral shape shift
  [shiftedSpectralShape] = SpectralShapeShift(sss, iloc, ival,
spectralShape, N, SR);
  %--- pitch transposition
  pt = 2.;                 % pitch transposition factor
  [syniloc, synival] = PitchTransposition(iloc, ival,
shiftedSpectralShape, shapePos, pt, N);
  if (shapePos > 1)
    synival =
interp1(shiftedSpectralShape(1,shapePos)',shiftedSpectralShape(2,shap
ePos)',syniloc, 'linear');
  else
    synival = ival;
  end
  %--- comb filtering the residual
  CombCoef = 1;
  if (isHarm==1)
    resfft = combFilter(resfft, N, SR/(pitchvalue*pt), CombCoef);
  end
end


%===== gender change: woman to man =====
if (isHarm == 1)
  pitchmin=100;
  pitchmax=500;
  sssmax = 50;
  if (pitchvalue<pitchmin)
    sss = 0;
  elseif (pitchvalue>pitchmax)
    sss = sssmax;
  else
    sss = (pitchvalue-pitchmin)/((pitchmax-pitchmin)/sssmax);
  end
  %--- spectral shape shift
```

```
  [synival] = SpectralShapeShift(-sss, iloc, ival, N, SR, nSines,
MinMag)
  %--- pitch transposition
  pt = 0.5;
  [syniloc, synival] = PitchTransposition(iloc, ival, pt, N, MinMag,
nSines);
  %--- comb filtering the residual
  CombCoef = 1;
  if (isHarm==1)
    resfft = combFilter(resfft, N, SR/(pitchvalue*pt), CombCoef);
  end
end
```

9.4.7 Harmonizer

In order to create the effect of a harmonizing vocal chorus, we can add pitch-shifted versions of the original voice (with the same timbre) and force them to be in tune with the original melody.

```
% harmonizer
nVoices = 2;
nSynSines = nSines*(1+nVoices);
[spectralShape, shapePos] = CalculateSpectralShape(syniloc(1:nSines),
synival(1:nSines), MinMag, N);
synival(1:nSines) = synival(1:nSines) - 100;
pt = [1.3 1.5]; % pitch transposition factor
ac = [-1 -2]; % amplitude change factor in dB

for i=1:nVoices
   [tmpsyniloc, tmpsynival] = ...
      PitchTransposition(syniloc(1:nSines), synival(1:nSines),
spectralShape, shapePos, pt(i), N);
   tmpsynival = tmpsynival + ac(i);
   syniloc(nSines*i+1:nSines*(i+1)) = tmpsyniloc;
   synival(nSines*i+1:nSines*(i+1)) = tmpsynival;
   if (pin > 0)
      distminindex(nSines*i+1:nSines*(i+1)) =
distminindex(1:nSines)+nSines*i;
   end
end
```

9.4.8 Hoarseness

Although hoarseness is sometimes thought of as a symptom of some kind of vocal disorder [Childers, 1994], this effect has been widely used by singers in order to resemble the voice of famous performers (Louis Armstrong or Tom Waits, for example). In this elemental approximation, we accomplish a similar effect by just applying a gain to the residual component of our analysis.

```
rgain = 2; % gain factor applied to the residual
  ...
```

```
wavwrite((DAFx_outsine+rgain*DAFx_outres)/mm,SR,'DAFx_outsineres.wav'
);
```

9.4.9 Morphing

Morphing is a transformation with which, out of two or more elements, we can generate new ones with hybrid properties.

With different names, and using different signal processing techniques, the idea of audio morphing is well known in the Computer Music community (Serra, 1994; Tellman, Haken, Holloway, 1995; Osaka, 1995; Slaney, Covell, Lassiter. 1996; Settel, Lippe, 1996). In most of these techniques, the morph is based on the interpolation of sound parameterizations resulting from analysis/synthesis techniques, such as the Short-time Fourier Transform (STFT), Linear Predictive Coding (LPC) or Sinusoidal Models (see Cross Synthesis and Spectral Interpolation in 8.3.6 and 8.3.7 respectively).

In the following Matlab code we introduce a morphing algorithm based on the interpolation of the frequency, phase, and amplitude of the sinusoidal component of two sounds. The factor 'alpha' controls the amount of the first sound we will have in the resulting morph. Different controlling factors could be introduced for more flexibility. Note that if the sounds have different durations, the sound resulting from the morphing will have the duration of the shortest one.

Next we include the code lines that have to be inserted in the Transformation part. However, by doing only this you won't get any morph. The morph transformation requires two or more inputs we have not included in order to keep the code short and understandable. Therefore the code will have to include the following modifications:

1. You will have to read two input sounds:
```
DAFx_in1  = wavread('source1.wav');
DAFx_in2  = wavread('source2.wav');
```
2. You will have to analyze both sounds. This means every analysis code line will have to be duplicated using the variable names: iloc1, iloc2, ival1, ival2, iphase1, iphase2, syniloc1, syniloc2, synival1, synival2, syniphase1, syniphase2, distminindex1, distminindex2, previousiloc1, previousiloc2, previousival1, previousival2, pitch1, pitch2, and pitcherror1, pitcherror2.

```
%=== Morphing ===
%--- sorting the frequencies in bins; iloc1, iloc2
[syniloc1, ind1] = sort(iloc1);
synival1   = ival1(ind1);
syniphase1 = iphase1(ind1);
distminindex1 = distminindex1(ind1);
[syniloc2, ind2] = sort(iloc2);
synival2   = ival2(ind2);
syniphase2 = iphase2(ind2);
distminindex2 = distminindex2(ind2);
%--- interpolation -----
alpha = 0.5;      % interpolation factor
syniloc     = alpha*syniloc1 + (1-alpha)*syniloc2;
synival     = alpha*synival1 + (1-alpha)*synival2;
```

```
%--- pitch computation
isHarmsyn = isHarm1*isHarm2;
if (isHarmsyn ==1)
  npitchpeaks = min(50,nPeaks);
  [pitchvalue,pitcherror] =
TWM(syniloc(1:npitchpeaks),synival(1:npitchpeaks),N,SR);
else
  pitchvalue = 0;
  pitcherror = 0;
end
if (pin==0) %--- for the first frame
  nNewPeaks = nSines;
else
  %--- creation of new born tracks
  for i=1:nSines
    if (previoussyniloc(i)==0)
      [previoussyniloc(i), previoussynival(i)] = CreateNewTrack ...
       (syniloc, synival, previoussyniloc, previoussynival, nSines,
MinMag);
      nNewPeaks = nNewPeaks - 1;
    end
  end
  %--- peak tracking of the peaks of the synthetized signal
  [syniloc, synival, syniphase, previoussyniloc, previoussynival,
distminindex] = ...
  peakTrackSimple (nSines, nPeaks, N, SR, pitchvalue, syniloc,
synival, ...
    syniphase, isHarmsyn, previoussyniloc, previoussynival);
end
```

9.5 Content dependent processing

The hierarchical data structure that includes a complete description of a given sound offers many possibilities for sound transformations. Modifying several attributes at the same time and at different abstraction levels achieve, as it has already been pointed out in the previous section, most musically or end-user meaningful transformations.

Higher-level transformations can refer to aspects like sound character, articulation or expressive phrasing. These ideas lead to the development of front ends such as graphical interfaces or knowledge-based systems [Arcos, 1997 and 1998] that are able to deal with the complexity of this sound representation.

In this section we introduce two applications that have been developed with these ideas in mind: a singing voice conversion and a time scaling module.

9.5.1 Real-Time Singing Voice Conversion

Here we present a very particular case of audio morphing. What we want is to be able to morph, in real-time, two singing voice signals in order to control the resulting synthetic voice by mixing the characteristics of the two sources. Whenever this control is performed by modifying a reference voice signal and matching its

individuality parameters to another, we can refer to it as voice conversion [Masanobu, 1992].

In such a context, a karaoke-type application was developed in which the user can sing like his/her favorite singers [Cano, Loscos, Bonada, Boer, Serra, 2000]. The result is an automatic impersonating system that allows the user to morph his/her voice attributes (such as pitch, timbre, vibrato and articulations) with the ones from a prerecorded singer, which from now on we will refer to as *target*.

In this particular implementation, the target's performance of the complete song to be morphed is recorded and analyzed beforehand. In order to incorporate the corresponding characteristics of the target's voice to the user's voice, the system first recognizes what the user is singing (phonemes and notes), looks for the same sounds in the target performance (i.e. synchronizing the sounds), interpolates the selected voice attributes, and synthesizes the output morphed voice. All this is accomplished in real-time.

Fig 9.23 shows the general block diagram of the voice impersonator system. The system relies on two main techniques that define and constrict the architecture: the SMS framework (see 9.2.2) and a Hidden Markov Model based Speech Recognizer (SR). The SMS implementation is responsible of providing a suitable parameterization of the singing voice in order to perform the morph in a flexible and musical-meaningful way. On the other hand, the SR is responsible of matching the singing voice of the user with the target.

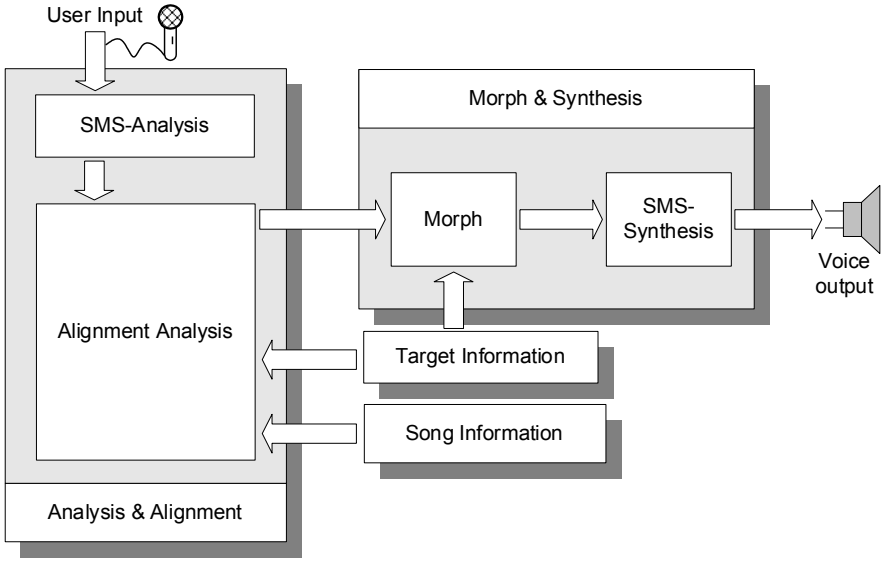


Fig 9.23. *System block diagram*

Let's have an overview of the whole process. Before we can morph a particular song, we have to supply information about the song to be morphed and the song recording itself (Target Information and Song Information). The system requires the phonetic transcription of the lyrics, the melody as MIDI data, and the actual recording to be used as the target audio data. Thus, a good impersonator of the singer that originally sang the song has to be recorded. This recording has to be analyzed with SMS, segmented into morphing units (phonemes), and each unit labeled with the

appropriate note and phonetic information of the song. This preparation stage is done semi-automatically, using a non-real time application developed for this task.

Once we have all the required inputs set we can start processing the user's voice. The first module of the running system includes the real-time analysis and the recognition/alignment steps. Each analysis frame, with the appropriate parameterization, is associated with the phoneme of a specific moment of the song and thus with a target frame. Once a user frame is matched with a target frame, we morph them interpolating data from both frames and we synthesize the output sound. Only voiced phonemes are morphed and the user has control over which and by how much each parameter is interpolated. The frames belonging to unvoiced phonemes are left untouched, thus always having the user's unvoiced consonants in the output.

Several modifications are done to the basic SMS procedures to adapt them to the requirements of the impersonator system. The major changes include the real-time implementation of the whole analysis/synthesis process with a processing latency of less than 30 milliseconds and the tuning of all parameters to the particular case of the singing voice. These modifications include the extraction of higher-level parameters meaningful in the case of the singing voice and that will be later used in the morphing process [Cano, Loscos, 1999].

To solve the matching problem the system includes an Automatic Speech Recognizer (ASR) based on phoneme-base discrete HMM's. This ASR has been adapted to handle musical information and work with very low delay [Loscos, Cano, Bonada, 1999] since we cannot wait for a phoneme to be finished before we recognize, moreover, we have to assign a phoneme to each frame. This would be a rather impossible/impractical situation if it was not for the fact that the lyrics of the song are known beforehand. This reduces a big portion of the search problem: all the possible paths are restricted to just one string of phonemes, with several possible pronunciations. The problem is cut down to the question of locating the phoneme in the lyrics and placing the start and end points.

Besides knowing the lyrics, music information is also available. The user is singing along with the music, and hopefully according to a tempo and melody already specified in the score. Thus, we also know the time at which a phoneme is supposed to be sung, its approximate duration, its associated pitch, etc. All this information is used to improve the performance of the recognizer and also to allow resynchronization, for example in the case that the singer skips a part of the song.

Depending on the phoneme the user is singing, a unit from the target is selected. Each frame from the user is morphed with a different frame from the target, advancing sequentially in time. Then the user has the choice to interpolate the different parameters extracted at the analysis stage, such as amplitude, fundamental frequency, spectral shape, residual signal, etc. In general, the amplitude will not be interpolated, thus always using the amplitude from the user and the unvoiced phonemes will not be morphed either, thus always using the consonants from the user. This will give the user the feeling of being in control. This recognition and matching process is illustrated in Fig 9.24.

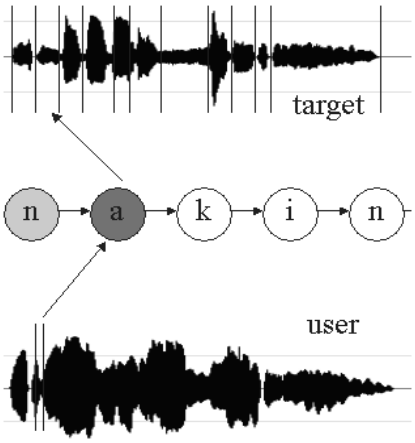


Fig 9.24. *Recognition and matching of morphable units*

Whenever the spectral shape is interpolated, and the morph factor is set around 50%, the resulting spectral shape is smoothed and looses much of its timbre characteristics. This problem can be solved if formants are included in the spectral shape model and they are taken into account in the interpolation step.

In most cases, the durations of the user and target phonemes to be morphed will be different. If a given user's phoneme is shorter than the one from the target, the system will simply skip the remaining part of the target phoneme and go directly to the articulation portion. In the case when the user sings a longer phoneme than the one present in the target data, the system enters in the loop mode. Each voiced phoneme of the target has a loop point frame, marked in the preprocessing, non-real time stage. The system uses this frame to loop-synthesis in case the user sings beyond that point in the phoneme. Once we reach this frame in the target, the rest of the frames of the user will be interpolated with that same frame until the user ends the phoneme. This process is illustrated in Fig 9.25.

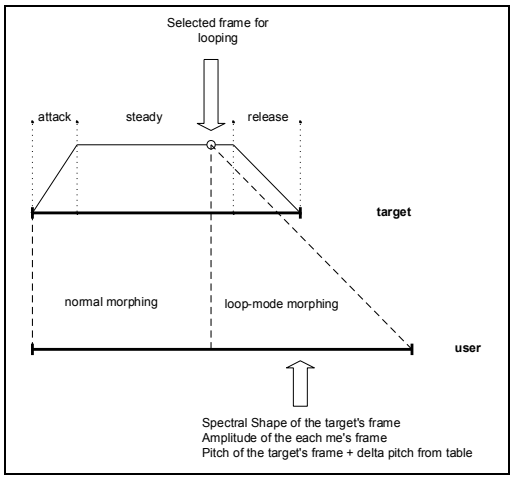


Fig 9.25. *Loop synthesis diagram*

The frame used as a loop frame requires a good spectral shape and, if possible, a pitch very close to the note that corresponds to that phoneme. Since we keep a constant spectral shape, we have to do something to make the synthesis sound natural. The way

we do it is by using some "natural" templates obtained from the analysis of a longer phoneme that are then used to generate more target frames to morph with the loop frame. For example, one feature that adds naturalness is pitch variations of a steady state note sung by the same target. These delta pitches are kept in a look up table whose first access is random and consecutive values are read afterwards. Two tables are kept, one with variations of steady pitch and another one with vibrato to generate target frames.

Once all the chosen parameters have been interpolated in a given frame, they are added back to the basic SMS frame of the user's synthesis frame. The synthesis is done with the standard synthesis procedures of SMS.

9.5.2 Time scaling

Time-scaling an audio signal means changing the length of the sound without affecting other perceptual features, such as pitch or timbre. Many different techniques, both in time and frequency domain, have been proposed to implement this effect. Some frequency domain techniques yield high-quality results and can work with large scaling factors. However, they are bound to present some artifacts, like phasiness, loss of attack sharpness and loss of stereo image. In this section we will present a frequency domain technique for near loss-less time-scale modification of a general musical stereo mix [Bonada 2000].

9.5.2.1 The basic sytem

The general block diagram of the system is represented in *Fig 9.26.* First, the input sound is windowed and goes through the FFT that yields the analysis frame (AF_n), that is the spectrum bins and the amplitude and phase envelopes. Then the time-scale module generates the synthesis frame (SF_m) that is input to the inverse FFT (IFFT). Finally, the Windowing&Overlap-Add block divides the sound segment by the analysis window and multiplies it by the overlap-add window, to reconstruct the output sound. The basics of the FFT/IFFT approach are detailed in chapter 8.

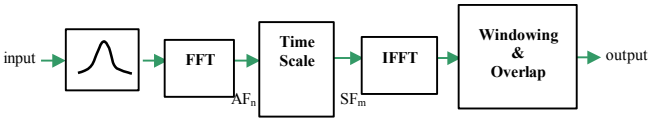


Fig 9.26. *General diagram*

It is important to remark that the frame rate used in both the analysis and synthesis modules is the same, as opposed to the most broadly used change of frame rate in synthesis to achieve the time-scale. The window size and type must also be the same in both processes.

In the next figure, we can see what happens in the cases of a time-scale stretching factor (TS > 1), and a time compression factor (TS < 1). The horizontal axis corresponds to the time of the center of the frame in the input audio signal. Therefore, when TS>1, the time increments relative to the input audio signal will be shorter in

the synthesis than in the analysis frames, but the actual frame rate will be exactly the same. Each synthesis frame points to the nearest analysis frame looking to the right.

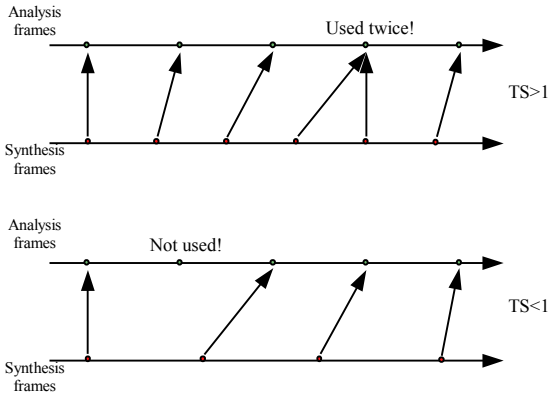


Fig 9.27. *Analysis and synthesis frames*

As it is shown in Fig 9.27, in some cases, an analysis frame is used twice (or more) while on other cases some frames are never used! This technique will not add any artifacts, provided the frame size we use is small enough and the sound does not present abrupt changes in that particular region. In the case of a percussive attack, though, a frame repetition or omission can be noticed regardless the analysis frame size. Therefore, some knowledge of the sound segment features is needed to decide where this technique can or cannot be applied.

In Fig 9.28, you can see a detailed block diagram of the time-scale module. The analysis frames (AF_n), containing the spectrum amplitude and phase envelopes, are fed to the time-scaling module. This module performs a peak detection (see 9.3.1.2) and a peak continuation algorithm (see 9.3.1.4) on the current and previous (Z^{-1}) amplitude envelopes. Then, only the peaks that belong to a sinusoidal track are used as inputs to the spectrum phase generation module. Note that the time-scale module only changes the phase, leaving the spectral amplitude envelope as it is.

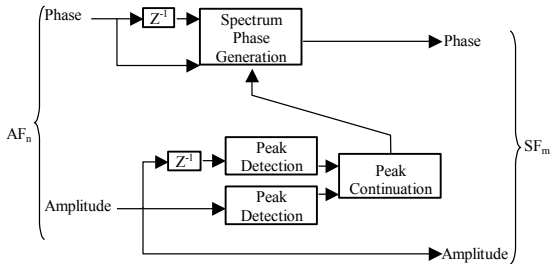


Fig 9.28. *The time-scales module*

The phase of each peak is computed supposing that the frequency varies linearly between two consecutives frames and that there is some phase deviation ($\Delta\varphi$) (see Fig 9.29). The usage of the same frame rate in analysis and synthesis allows us to suppose that the phase variation between two consecutives frames is also the same.

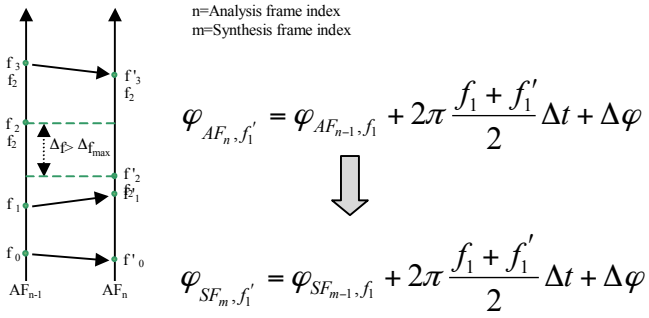


Fig 9.29. *Peak continuation and phase generation*

9.5.2.2 Common problems and solutions in time-scaling

The previous chapter already introduced an spectral technique for time scaling based on the phase vocoder approach. This kind of implementation presents very well known artifacts. In this section we will describe each of these problems and the solution that the implementation we are proposing can provide.

Phasiness

We will first refer to the phase vocoder implementation. In the original frame, the phase envelope was flat around the peak because of the circular convolution of the analysis window with the sinusoid. But after the time-scale is applied, the phase has lost its original behavior. This artifact is introduced due to the fact that the phase of each bin advances at different speed (see 8.2.5). This loss of peak's phase coherence is known as *phasiness*. To avoid this problem we can apply the original relative behavior of the phase around the peak. As pointed in [Laroche and Dolson, 1997], each peak subdivides the spectrum into a different region, with a phase related to the peak's phase. The phase around each peak is obtained applying the delta phase function of the original spectrum phase envelope (see Fig 9.30).

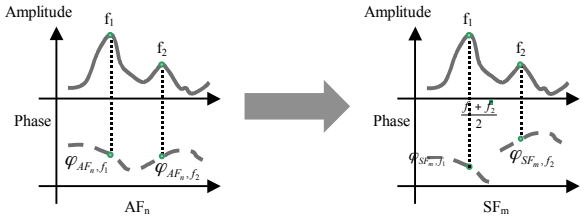


Fig 9.30. *Original delta phase function around each peak.*

Loss of attack transients

A typical artifact of the phase vocoder approach is the smoothing of the attack transients. A possible approach is to modify the sinusoidal plus residual model in order to model these transients [Verma and Meng, 1998]. Another possible solution is not to apply the time-scale on this kind of regions of the input signal so the original

timing is respected (see Fig 9.31). Consequently, and in order to preserve the overall scaling factor, a greater amount of scaling should be applied to surrounding regions.

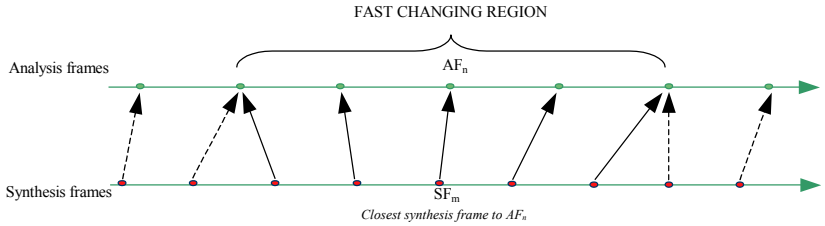


Fig 9.31. *Attack transient region*

In order to apply the previous technique, it is necessary to detect attack transients of the sound in an unsupervised manner. The computation of relative changes of energy along several frequency bands can be used for that purpose. A low frequency band could, for example, detect sharp bass notes, while a high frequency band was set to detect hits of a crash cymbal.

The spectrum of the input signal is given by

$$X(sR_a,k) = \left|X(sR_a,k)\right| \cdot e^{j\varphi(sR_a,k)} \tag{23}$$

where the FFT has been sampled every R_a samples in time, and s is the time index of the short-term transform. If we define a set of frequency bands $B_i(k)$, then the energy of the i^{th} band can be computed as follows:

$$E(s,i) = \sum_{l=0}^{N-1} B_i(k) \cdot X^2(sR_a,k) \tag{24}$$

and the relative change of energy $C(s,i)$ at frame s as,

$$C(s,i) = \frac{-2E(s-2,i) - E(s-1,i) + E(s+1,i) + 2E(s+2,i)}{E(s,i)} \tag{25}$$

The maximums of $C(s,i)$ over some threshold should then indicate the attack transients of the input signal at the desired band.

Frequency versus time resolution

As explained in 9.3.1.1, it is desirable to have long windows in order to achieve a high frequency resolution, but also to have short windows so to achieve a better temporal resolution. If the audio signal presents an important low frequency component, the use of a long window is a must, because the low frequency peaks will be too close to be distinguishable if we use a short window. On the other hand if we apply a very long window, the time-scaling process will add reverb and will smooth the sound.

The solution proposed is to use parallel windowing, that is, several analysis channels. Each channel is the result of an FFT with a specific window size, window type and zero padding. Obviously, the window should be longer for low frequencies than for high frequencies. The peak detection process is applied to each of the channels while the peak continuation takes care of the desired channel frequency cuts, so it can

connect peaks of different channels. Then the time-scale module fills the spectrum of all the channels (*amplitude and phase envelopes*) and applies a set of parallel filters $H_n(f)$ that must add up to a constant (*all pass filter*).

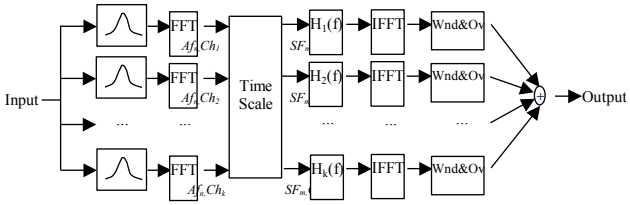


Fig 9.32. *Multiple parallel windowing*

If the cutoff frequency of a channel was close to a spectral peak, this would be broken apart into two different channels and we would be introducing some kind of artifacts. For that reason, and in order to guarantee that phase and amplitude envelopes around the peak behave the way we expect, we need to provide our system of time-varying frequency cuts. Each frequency cut is computed as the middle point between the two closest peaks to the original frequency cut (see Fig 9.33).

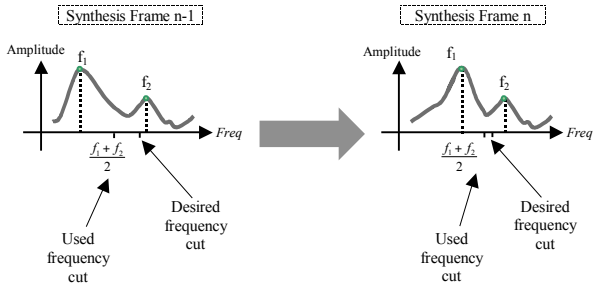


Fig 9.33. *Variable phase frequency cut*

Loss of stereo image

In the case of stereo signals, if we process independently each one of the channels, most of the stereo image is bound to be lost. This artifact is mainly due to the fact that the time-scale process changes the phase relation between the two channels. Therefore, if we want to keep the stereo image, it is necessary to preserve the phase and amplitude relation between left and right channels.

The fact that the system does not change the amplitude envelope of the spectrum guarantees that the amplitude relation between channels will be preserved, provided we always use frames with identical time tags for both channels. For that purpose, we need to synchronize the attack transients between the two channels.

In the next figure you can see the simplified block diagram of the stereo time-scale system. Notice that the number of FFT and IFFT operations is multiplied by two and, as a consequence, the same happens to the processing time.

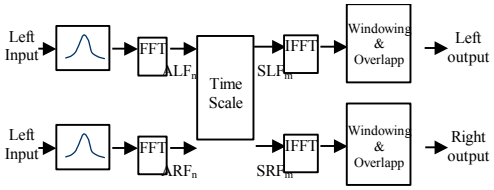


Fig 9.34. *Stereo time-scale*

9.5.2.3 Time varying time-scale

The system here presented can del with time varying scaling factors with no loss of quality tradeoff. The only significant change is that the time increments of the synthesis frames in the input signal are not constant.

The application of time-varying tempo variations opens up many new and interesting perspectives. The system could be easily adapted and used for alignment and synchronization of two sound sources. The amount of time-scale can be used in a wise way to inspire emotions. For example, to increase the climax or the suspense of a musical piece, by slowing or increasing the tempo during certain fragments. Another interesting application could be to control the time-scale factor the same way as the orchestra conductor does and play in real-time a previously recorded background with a live performance.

9.6 Conclusions

Throughout this chapter, we have shown how the use of higher-level spectral models can lead to new and interesting sound effects and transformations. We have also seen that it is not easy nor immediate to get a good spectral representation of a sound, so the usage of this kind of approach needs to be carefully considered bearing in mind the application and the type of sounds we want to process.

For example, most of the techniques here presented work well only on monophonic sounds and some rely on the pseudo-harmonicity of the input signal.

Nevertheless, the use of spectral models for musical processing has not been around too long and it has already proven useful for many applications, as the ones presented in this chapter. Under many circumstances, higher-level spectral models, such as the Sinusoidal plus Residual, offer much more flexibility and processing capabilities than more immediate representations of the sound signal.

In general, higher-level sound representations will offer more flexibility at the cost of a more complex and time-consuming analysis process. It is important to remember that the model of the sound we choose will surely have great effect on the kind of transformations we will be able to achieve and on the complexity and efficiency of our implementation. Hopefully, the reading of this chapter, and the book as a whole, will guide the reader on taking the right decision in order to get the desired results.

9.7 References

Arcos, J. L.; R. López de Mántaras; X. Serra. 1998. "Saxex: a Case-Based Reasoning System for Generating Expressive Musical Performances", *Journal of New Music Research*, Vol. 27, N. 3, Sept. 1998.

Arcos, J. LL., R. Lopez de Mántaras, X. Serra. 1997. "Saxex: a Case-Based Reasoning System for Generating Expressive Musical Performances". *Proceedings of the ICMC 1997.*

Bonada, J. 2000. "Automatic technique in frequency domain for near-lossless time-scale modification of audio." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: Computer Music Association.

Cano, P. 1998. "Fundamental Frequency Estimation in the SMS Analysis*"*. *Proceedings of the Digital Audio Effects Workshop (DAFX98)*, 1998.

Cano, P.; A. Loscos; J. Bonada; M. de Boer; X. Serra; 2000. "Voice Morphing System for Impersonating in Karaoke Applications." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: Computer Music Association.

Cano, P.; A. Loscos; 1999. "Singing Voice Morphing System based on SMS". *UPC,* 1999.

Childers, D.G. 1994. "Measuring and Modeling Vocal Source-Tract Interaction". *IEEE Transactions on Biomedical Engineering 1994.*

Cox, M. G. 1971. "An algorithm for approximating convex functions by means of first-degree splines." *Computer Journal*, vol. 14, pp. 272--275.

Depalle, Ph.; Hélie, T. 1997. "Estraction of Spectral Peak Parameters using a Short-Time Fourier Transform Modeling and no Sidelobe Windows" *Proceedings of the 1997 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. Monhonk.

Depalle, Ph.; G. Garcia; X. Rodet. 1993. "Analysis of Sound for Additive Synthesis: Tracking of Partials Using Hidden Markov Models." *Proceedings of the 1993 International Computer Music Conference*. San Francisco: Computer Music Association.

Ding Y.; X. Qian. 1997. "Sinusoidal and Residual Decomposition and Residual Modeling of Musical Tones Using the QUASAR Signal Model". Proceedings of the 1997 *International Computer Music Conference*. San Francisco: Computer Music Association.

Fitz, K.; L. Haken, and P. Christensen. 2000. "A New Algorithm for Bandwidth Association in Bandwidth-Enhanced Additive Sound Modeling". *Proceedings of the 2000 International Computer Music Conference*. San Francisco: Computer Music Association.

Goodwin, M. 1997. *Adaptative Signal Models: Theory, Algorithms and Audio Applications*. Ph.D. Dissertation, University of California. Berkeley.

Goodwin, M. 1996. “Residual Modeling in Music Analysis-Synthesis”. *Proceedings of the 1996 IEEE-ICASSP*. Atlanta.

Harris, F. J. 1978. “On the use of windows for harmonic analysis with the discrete Fourier transform,” *Proceedings IEEE*, vol. 66, pp. 51-83.

Herrera. P. 1998. “Vibrato Extraction and Parameterization in the Spectral Modeling Synthesis Framework”. *Proceedings of the Digital Audio Effects Workshop (DAFX98)*, 1998.

Hess, W. 1983. *Pitch Determination of Speech Signals*. New York: Springer-Verlag.

Laroche, J. ; Dolson, M. 1997. "About this phasiness business". *Proceedings of International Computer Music Conference, 1997*.

Loscos, A.; Cano, P.; Bonada, J. 1999. “Singing Voice Alignment to text” *Proceedings of the ICMC 1999*

Maher, R. C. and J. W. Beauchamp. 1994. “Fundamental Frequency Estimation of Musical Signals using a two-way Mismatch Procedure.” *Journal of the Acoustical Society of America* 95(4):2254--2263.

Masanobu, Abe. 1992. "A Study on Speaker Individuality Control", Doctorate Thesis

McAulay, R. J. and T. F. Quatieri. 1986. “Speech Analysis/Synthesis based on a Sinusoidal Representation.” *IEEE Transactions on Acoustics, Speech and Signal Processing* 34(4):744--754.

Osaka, N. 1995. “Timbre Interpolation of sounds using a sinusoidal model”, *Proceedings of the ICMC 1995*.

Rodet, X. and Ph. Depalle. 1992. “Spectral Envelopes and Inverse FFT Synthesis”, *93rd Convention of the Audio Engineering Society*. San Francisco, October 1992.

Rossignol, S. and others. 1998. “Feature Extraction and Temporal Segmentation of Acoustic Signals”. *Proceedings of the 1998 International Computer Music Conference*. San Francisco: Computer Music Association

Sedgewick, R. 1988. *Algorithms*. Reading, Massachusetts: Addison-Wesley.

Serra, X. 1989. *A System for Sound Analysis/Transformation/Synthesis based on a Deterministic plus Stochastic Decomposition*. Ph.D. Dissertation, Stanford University.

Serra, X. and J. Smith. 1990. “Spectral Modeling Synthesis: A Sound Analysis/Synthesis System based on a Deterministic plus Stochastic Decomposition.’’ *Computer Music Journal* 14(4):12--24.

Serra, X. 1994. “Sound Hybridization Techniques based on a Deterministic plus Stochastic Decomposition Model.” *Proceedings of the 1994 International Computer Music Conference*. San Francisco: Computer Music Association.

Serra, X. 1996. “Musical Sound Modeling with Sinusoids plus Noise”, in G. D. Poli, A. Picialli, S. T. Pope, and C. Roads, editors, *Musical Signal Processing*. Swets & Zeitlinger Publishers.

Serra, X.; J. Bonada. 1998. "Sound Transformations Based on the SMS High Level Attributes", *Proceedings of the 98 Digital Audio Effects Workshop*.

Settel, Z., C. Lippe. 1996. "Real-Time Audio Morphing", *7th International Symposium on Electronic Art*, 1996.

Slaney, M., M. Covell, B. Lassiter. 1996. "Automatic audio morphing", *Proc. IEEE Int. Conf. Acosut. Speech Signal Process*. 2, 1001-1004 (1996).

Smith, J.O. and X. Serra. 1987. "PARSHL: An Analysis/Synthesis Program for Non-Harmonic Sounds based on a Sinusoidal Representation." *Proceedings of the 1987 International Computer Music Conference*. San Francisco: Computer Music Association.

Strawn, J. 1980. "Approximation and Syntactic Analysis of Amplitude and Frequency Functions for Digital Sound Synthesis." *Computer Music Journal* 4(3):3--24.

Tellman, E., L. Haken, B. Holloway. 1995. "Timbre Morphing of Sounds with Unequal Number of Features", *J. Audio Eng. Soc.*, 43:9 1995.

Tellman, E.; L. Haken; B. Holloway 1995. "Timbre morphing of sounds with unequal numbers of features". *Journal of the Audio Engineering Society*, 43(9), 678-89.

Verma, T. S. ; T. H. Y. Meng. 2000. "Extending Spectral Modeling Synthesis With Transient Modeling Synthesis", *Computer Music Journal* 24:2, pp.47-59.

Verma, T. S.; T. H. Y. Meng. 1998. "Time Scale Modification Using a Sines+Transients+Noise Signal Model", *Proceedings of the Digital Audio Effects Workshop (DAFX98)*, Barcelona, November 1998.

Vidal, E. and A. Marzal 1990. "A Review and New Approaches for Automatic Segmentation of Speech Signals". L. Torres and others (eds.), *Signal Processing V: Theories and Applications*, Elsevier Science Publishers, 1990.

9.8 Index of figures

Fig 9.1. Block diagram of a simple spectral processing framework 2

Fig 9.2. Block diagram of a higher-level spectral processing framework.....3

Fig 9.3. Block diagram of the Sinusoidal plus residual analysis. 6

Fig 9.4. a. Sinusoidal component. b. Residual spectrum.....7

Fig 9.5. Time vs. frequency resolution tradeoff 8

Fig 9.6. Effect of applying a window in the time domain 9

Fig 9.7. Effect of the window size in distinguishing between two sinusoids.....9

Fig 9.8. Peak detection. a. Peaks in the magnitude spectrum. b. Peaks in the phase spectrum. 12

Fig 9.9. Parabolic interpolation in the peak detection process 12

Fig 9.10. Frequency trajectories resulting from the sinusoidal analysis of a vocal sound. 18

Fig 9.11. Traditional peak continuation algorithm[McAulay and Quatiery, 1986].....18

Fig 9.12. Peak Continuation process. g represent the guides and p the spectral peaks. 19

Fig 9.13. Time domain substraction 23
Fig 9.14. (a)Original spectrum, (b) Residual spectrum and aprosimation..... 24
Fig 9.15. Diagram of the spectral synthesis..... 28
Fig 9.16. Additive Synthesis Block Diagram 29
Fig 9.17. Residual synthesis approximation 31
Fig 9.18. Integrating sinusoidal plus residual synthesis 32
Fig 9.19. Band-pass filter with arbitrary resolution..... 39
Fig 9.20. Frequency shift of the partials 40
Fig 9.21. Frequency stretch 40
Fig 9.22. Spectral shape shift of value Δf..... 42
Fig 9.23. System block diagram 48
Fig 9.24. Recognition and matching of morphable units..... 50
Fig 9.25. Loop synthesis diagram..... 50
Fig 9.26. General diagram 51
Fig 9.27. Analysis and synthesis frames..... 52
Fig 9.28. The time-scales module..... 52
Fig 9.29. Peak continuation and phase generation 53
Fig 9.30. Original delta phase function around each peak. 53
Fig 9.31. Attack transient region 54
Fig 9.32. Multiple parallel windowing 55
Fig 9.33. Variable phase frequency cut 55
Fig 9.34. Stereo time-scale 56