10

Spectral processing

J. Bonada, X. Serra, X. Amatriain and A. Loscos

10.1 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 for a variety of musical applications and contexts. It should have been clear throughout the book that several points of view have to be considered, including a mathematical, and thus objective, perspective, and a perceptual, and 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-transformation systems. Moreover, as frequency-domain analysis is a somewhat similar process to 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 10.1). 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 7 are clear examples of this approach. Then the main issues are what is the intermediate representation and what parameters are available for applying the desired transformations.

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

DAFX: Digital Audio Effects, Second Edition. Edited by Udo Zölzer.
 Published 2011 by John Wiley & Sons, Ltd. ISBN: 978-0-470-66599-2

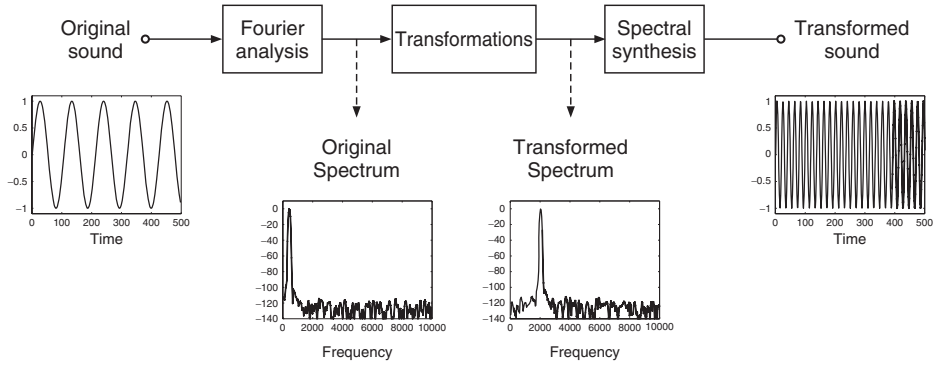


Figure 10.1 Block diagram of a simple spectral-processing framework.

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 make 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, as shown in Figure 10.2, 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.

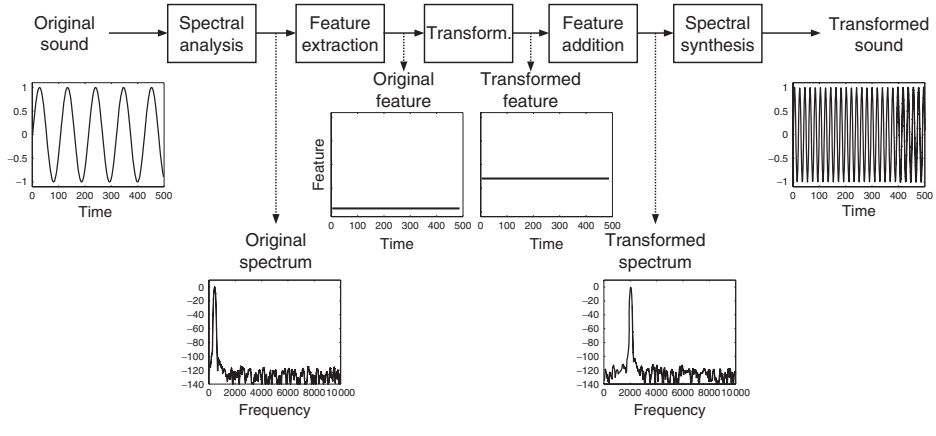


Figure 10.2 Block diagram of a higher-level spectral-processing framework.

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 cover the largest possible set of sounds and musical applications.

Having set the basis of the various spectral models, we will then give the details of the implementation techniques used both for their analysis and synthesis processes, providing MATLAB® code to implement a complete analysis-synthesis framework.

In the final section we will present a set of basic audio effects and their implementation based on the analysis-synthesis framework just introduced. **MATLAB** code is provided for all of them.

10.2 Spectral models

The most common approach for converting a time-domain signal into its frequency-domain representation is the short-time fourier transform (STFT), which can be expressed by the following equation

$$X_l(k) = \sum_{n=0}^{N-1} w(n)x(n + lH)\, e^{-j\omega_k n}, \tag{10.1}$$

where $X_l(k)$ is the complex spectrum of a given time frame, $x(n)$ is the input sound, $e^{-j\omega_k n}$ is a complex sinusoid with frequency ω_k expressed in radians, $w(n)$ is the analysis window, l is the frame number, k is the frequency index, n is the time index, H is the time index hop size, and N is the FFT size.

The STFT results in 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 already been presented in previous chapters.

In this chapter, we extend the STFT framework by presenting higher-level modeling of the spectral data obtained with it. There are many spectral models based on the STFT that have been developed for sound and music signals and that fulfill different compromises and applications. The decision as to which one to use in a particular situation is not an easy one. The boundaries are not clear and there are 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 computation time is the direct implementation of the STFT that, anyhow, yields a rather inflexible representation and inefficient coding scheme.

Here we introduce two different spectral models: sinusoidal and sinusoidal plus residual. These models represent an abstraction level higher than the STFT and from them, but with different compromises, we can identify and extract higher-level information on a musical sound, such as: harmonics, fundamental frequency, spectral shape, vibrato, or note boundaries. These models bring the spectral representation closer to our perceptual understanding of a sound. The complexity of the analysis will depend on the sound to be analyzed and the transformation desired. The benefits of going to this higher level of analysis are enormous and open up a wide range of new musical applications.

10.2.1 Sinusoidal model

Using the STFT representation, the sinusoidal model is a step towards a more flexible representation 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 sound $s(t)$ is modeled by

$$s(t) = \sum_{r=1}^{R} A_r(t)\cos[\theta_r(t)], \tag{10.2}$$

where $A_r(t)$ and $\theta_r(t)$ are the instantaneous amplitude and phase of the rth sinusoid, respectively, and R is the number of sinusoids [MQ86, SS87].

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 7, then detecting the spectral peaks (and measuring the magnitude, frequency and phase of each one), and finally organizing them as time-varying sinusoidal tracks. We can then reconstruct the original sound using additive synthesis.

The sinusoidal model yields a quite general analysis/synthesis technique that can be used in a wide range of sounds and offers a gain in flexibility compared with the direct STFT implementation.

10.2.2 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 sinusoidal representations, but it results in an enormous gain in flexibility [Ser89, SS90, Ser96]. One of its main drawbacks is that it is not suitable for transient signals, thus several extensions have been proposed to tackle these (e.g., [VM98, VM00]). The sound $s(t)$ is modeled in the continuous domain by

$$s(t) = \sum_{r=1}^{R} A_r(t) \cos[\theta_r(t)] + e(t), \tag{10.3}$$

where $A_r(t)$ and $\theta_r(t)$ are the instantaneous amplitude and phase of the rth sinusoid, R is the number of sinusoids and $e(t)$ is the residual component. 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 we might omit the detection of the phase of each peak. For many sounds the instantaneous phase that appears in the equation can be taken to be the integral of the instantaneous frequency $\omega_r(t)$, and therefore satisfies

$$\theta_r(t) = \int_0^t \omega_r(\tau)\, \mathrm{d}\tau, \tag{10.4}$$

where $\omega_r(t)$ 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 in 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

$$e(t) = \int_0^t h(t, \tau)\, u(\tau)\, \mathrm{d}\tau, \tag{10.5}$$

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 identification of the sinusoids is done by adding restrictions to a standard sinusoidal analysis approach. Then the residual is obtained by subtracting the sinusoids from the original sound. The residual can also be considered a stochastic signal and thus represent it with a noise-filter model. We can then reconstruct the original sound using additive synthesis for the sinusoidal component and subtractive synthesis for the residual component.

The sinusoidal plus residual model yields a more general analysis/synthesis framework than the one obtained with a sinusoidal model and it has some clear advantages for specific applications and types of sounds. It has also led to other different spectral models that still share some of its basic principles [DQ97, FHC00, VM00].

10.3 Techniques

In this section we present and discuss a set of implementations of the different spectral models, building one implementation on top of another and making sure that each implementation is self-contained, thus being useful for some applications. Each implementation is an analysis/synthesis system that has as input a monophonic sound, $x(n)$, plus a set of parameters, and outputs a synthesized sound, $y(n)$. If the parameters are set correctly and the input sound used is adequate for the model, the output sound should be quite close, from a perceptual point of view, to the input sound. We start with an implementation of the STFT, which should yield a mathematical input-ouput identity for any sound. On top of that we build an analysis/synthesis system based on detecting the spectral peaks, and we extend it to implement a simple sinusoidal model. The following system implements a sinusoidal model with a harmonic constraint, thus only working for pseudo-harmonic sounds. We then include the residual analysis, thus implementing a harmonic plus residual model. Finally we assume that the residual is stochastic and implement a harmonic plus stochastic residual model.

10.3.1 Short-time fourier transform

Figure 10.3 shows the general block diagram of an analysis/sysnthesis system based on the STFT.

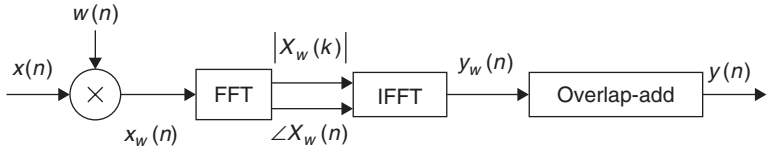


Figure 10.3 Block diagram of an analysis/synthesis system based on the STFT.

In order to use the STFT as a basis for the other spectral models we have to pay special attention to a number of issues related to the windowing process, specifically zero-phase windowing, the size and type of the analysis window, and the overlap-add process that is performed in the inverse transform process.

Below is the **MATLAB** code that implements a complete analysis/synthesis system based on the STFT.

M-file 10.1 (`stft.m`)

```
function y = stft(x, w, N, H)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the short-time Fourier transform
% x: input sound, w: analysis window (odd size), N: FFT size, H: hop size
```

```
% y: output sound
M = length(w);                                  % analysis window size
N2 = N/2+1;                                     % size of positive spectrum
soundlength = length(x);                        % length of input sound array
hM = (M-1)/2;                                   % half analysis window size
pin = 1+hM;       % initialize sound pointer in middle of analysis window
pend = soundlength-hM;                          % last sample to start a frame
fftbuffer = zeros(N,1);                         % initialize buffer for FFT
yw = zeros(M,1);                                % initialize output sound frame
y = zeros(soundlength,1);                       % initialize output array
w = w/sum(w);                                   % normalize analysis window
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);                % window the input sound
  fftbuffer(:) = 0;                             % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);         % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                           % compute FFT
  mX = 20*log10(abs(X(1:N2)));   % magnitude spectrum of positive frequencies
  pX = unwrap(angle(X(1:N2)));   % unwrapped phase spect. of positive freq.
  %-----synthesis-----%
  Y = zeros(N,1);                               % initialize output spectrum
  Y(1:N2) = 10.^(mX/20).*exp(i.*pX);            % generate positive freq.
  Y(N2+1:N) = 10.^(mX(N2-1:-1:2)/20).*exp(-i.*pX(N2-1:-1:2));
                                                % generate neg.freq.
  fftbuffer = real(ifft(Y));                    % inverse FFT
  yw(1:(M-1)/2) = fftbuffer(N-(M-1)/2+1:N);  % undo zero-phase window
  yw((M+1)/2:M) = fftbuffer(1:(M+1)/2);
  y(pin-hM:pin+hM) = y(pin-hM:pin+hM) + H*yw(1:M);     % overlap-add
  pin  = pin+H;                                 % advance sound pointer
end
```

In this code we step through the input sound x, performing the FFT and the inverse-FFT on each frame. This operation involves selecting a number of samples from the sound signal and multiplying their value by a windowing function. 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 plus one 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 trade-off (see Figure 10.4). A more specific limitation of the window size has to do with choosing windows with odd sample lengths in order to guarantee even symmetry about the origin.

The type of window used also has 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 Figure 10.5). One may be tempted to forget about deciding on these matters and apply no window at all, just taking M 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.

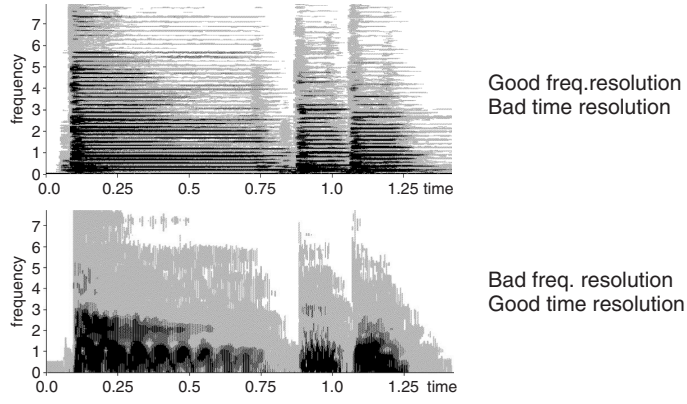


Figure 10.4 Time vs. frequency resolution trade-off.

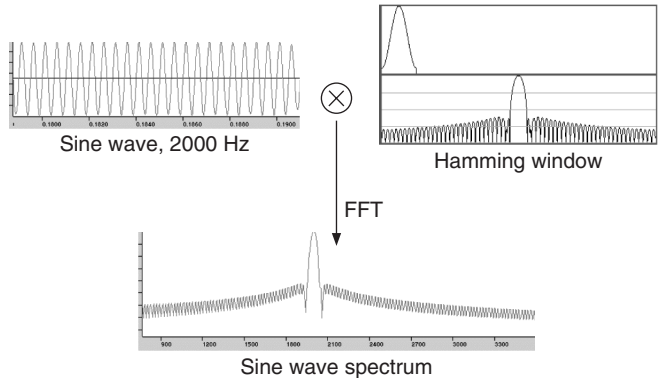


Figure 10.5 Effect of applying a window in the time domain.

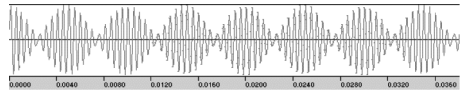
Two features of the transform of the window are especially relevant to whether a particular function is useful or not: the width of the main lobe, and the main to highest side lobe relation. The main lobe bandwidth is expressed in bins (spectral samples) and, in conjunction with the window size, defines the ability to distinguish two sinusoidal peaks (see Figure 10.6). 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 sinusoids of frequencies f_k and f_{k+1}:

$$M \geq B_s \frac{f_s}{|f_{k+1} - f_k|}. \tag{10.6}$$

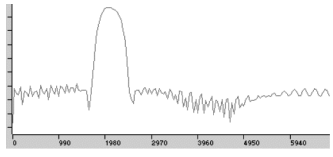
The amplitude relation between the main and the highest side lobe explains the amount of distortion a peak will receive from surrounding partials. It would be ideal to have a window with an extremely narrow main lobe and a very high main to secondary lobe relation. However, the inherent trade-off between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

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, the user provides the analysis window, $w(n)$, to use as the input parameter.

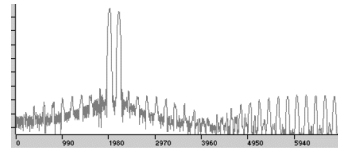
One may think that a possible way of overcoming the time/frequency trade-off is to add zeros at the extremes of the windowed signals to increase the frequency resolution, that is, to have the



Two sinusoids of 2.000 Hz and 2.200 Hz



Spectrum with a small window



Spectrum with a larger window

Figure 10.6 Effect of the window size in distinguishing between two sinusoids.

FFT size N larger than the window size M. This process is known as zero-padding and it represents an interpolation in the frequency domain. When we zero-pad a signal before the DFT process, we are not adding any information to its frequency representation, given that we are not adding any new signal samples. We will still not distinguish the sinusoids if Equation (10.6) is not satisfied, but we are indeed increasing the frequency resolution by adding intermediate interpolated bins. With zero-padding we are able to use sizes of windows that are not powers of two (requirement for using many FFT algorithm implementations) and we also obtain smoother spectra, which helps in the peak detection process, as later explained.

A final step before computing the FFT of the input signal is the circular shift already described in previous chapters. This data centering on the origin 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 input signal in order to take the next set of samples. The distance between the centers of two consecutive windows is known as hop size, H. 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 computationally 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 equation

$$A_w(m) = \sum_{n=-\infty}^{\infty} w(m - nH) \approx \text{constant}. \tag{10.7}$$

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 as to keep this factor around or below 1%

$$d_w = 100 \times \frac{\max_w [A_w(m)] - \min_w [A_w(m)]}{\max_w [A_w(m)]}. \tag{10.8}$$

After the analysis process we reverse every single step done until now, starting by computing the inverse FFT of every spectrum. If A_w is equal to a constant the output signal, $y(n)$, will be identical to the original signal, $x(n)$. Otherwise an amplitude deviation is manifested in the output sound, creating an audible distortion if the modulation is big enough.

A solution to the distortion created by the overlap-add process is to divide the output signal by the envelope A_w. If some window overlap exists, this operation surely removes any modulation coming from the window overlap process. However, this does not mean that we can apply any combination of parameters. Dividing by small numbers should be generally avoided, since noise coming from numerical errors can be greatly boosted and introduce undesired artifacts at synthesis. In addition, we have to be especially careful at beginning and end sections of the processed audio, since A_w will have values around zero. The following code implements a complete STFT analysis/synthesis system using this approach.

M-file 10.2 (stftenv.m)

```
function y = stftenv(x, w, N, H)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the short-time Fourier transform
% x: input sound, w: analysis window (odd size), N: FFT size, H: hop size
% y: output sound
M = length(w);                              % analysis window size
N2 = N/2+1;                                 % size of positive spectrum
soundlength = length(x);                    % length of input sound array
hM = (M-1)/2;                               % half analysis window size
pin = 1+hM;            % initialize sound pointer in middle of analysis window
pend = soundlength-hM;                      % last sample to start a frame
fftbuffer = zeros(N,1);                     % initialize buffer for FFT
yw = zeros(M,1);                            % initialize output sound frame
y = zeros(soundlength,1);                   % initialize output array
yenv = y;                                   % initialize window overlap envelope
w = w/sum(w);                               % normalize analysis window
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);            % window the input sound
  fftbuffer(:) = 0;                         % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);     % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                       % compute FFT
  mX = 20*log10(abs(X(1:N2)));    % magnitude spectrum of positive frequencies
  pX = unwrap(angle(X(1:N2)));    % unwrapped phase spect. of positive freq.
  %-----synthesis-----%
  Y = zeros(N,1);                                     % initialize spectrum
  Y(1:N2) = 10.^(mX/20).*exp(i.*pX);                  % generate positive freq.
  Y(N2+1:N) = 10.^(mX(N2-1:-1:2)/20).*exp(-i.*pX(N2-1:-1:2));
                                                      % generate neg.freq.
  fftbuffer = real(ifft(Y));                          % inverse FFT
  yw(1:(M-1)/2) = fftbuffer(N-(M-1)/2+1:N);           % undo zero-phase window
  yw((M+1)/2:M) = fftbuffer(1:(M+1)/2);
  y(pin-hM:pin+hM) = y(pin-hM:pin+hM) + yw(1:M);      % output signal overlap-add
  yenv(pin-hM:pin+hM) = yenv(pin-hM:pin+hM) + w;      % window overlap-add
  pin  = pin+H;                                       % advance sound pointer
end
yenvth = max(yenv)*0.1;                               % envelope threshold
yenv(find(yenv<yenvth)) = yenvth;
y = y./yenv;
```

The STFT process provides a suitable frequency-domain representation of the input signal. It is a far from trivial process and it is dependent on some low-level parameters closely related to the signal-processing domain. A little theoretical knowledge is required to understand the process, but we will also need practice to obtain the desired results for a given application.

10.3.2 Spectral peaks

The sinusoidal model assumes that each spectrum of the STFT representation can be explained by the sum of a small number of sinusoids. Given enough frequency resolution, and thus, enough points in the spectrum, the spectrum of a sinusoid can be identified by its shape. Theoretically, a sinusoid that is stable both in amplitude and in frequency, a partial of the sound, 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 (see Figure 10.7). 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.

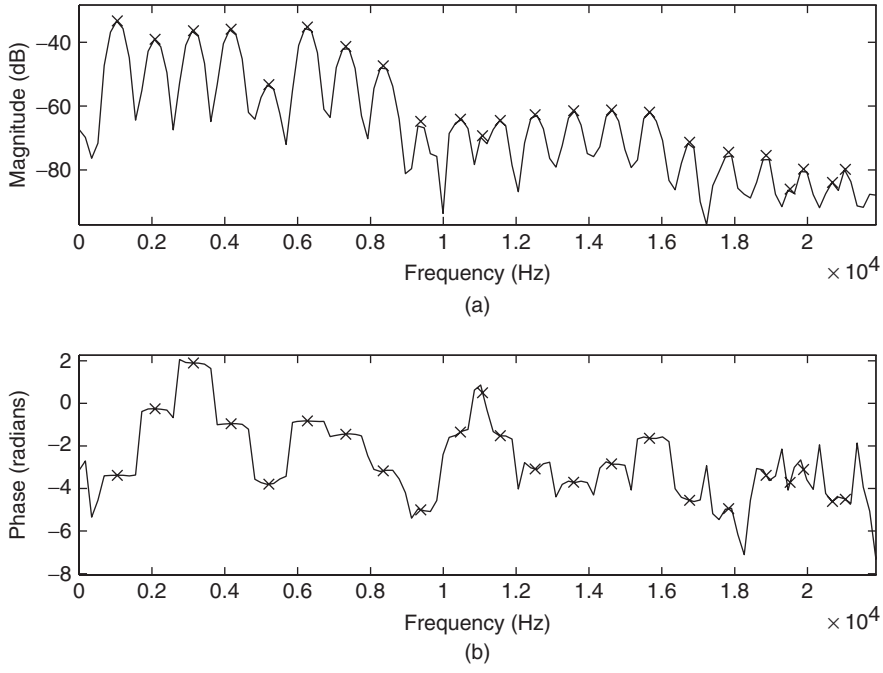


Figure 10.7 Peak detection: (a) peaks in magnitude spectrum; (b) peaks in the phase spectrum.

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 local maximum over a certain frequency range and to have an amplitude greater than a given threshold.

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.

Here we include a modified version of the STFT code in which we only use the spectral peak values to perform the inverse-FFT.

M-file 10.3 (stpt.m)

```
function y = stpt(x, w, N, H, t)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the peaks
% of the short-time Fourier transform
% x: input sound, w: analysis window (odd size), N: FFT size, H: hop size,
% t: threshold in negative dB, y: output sound
M = length(w);                                % analysis window size
N2 = N/2+1;                                   % size of positive spectrum
soundlength = length(x);                      % length of input sound aray
hM = (M-1)/2;                                 % half analysis window size
pin = 1+hM;        % initialize sound pointer at the middle of analysis window
pend = soundlength-hM;                        % last sample to start a frame
fftbuffer = zeros(N,1);                       % initialize buffer for FFT
yw = zeros(M,1);                              % initialize output sound frame
y = zeros(soundlength,1);                     % initialize output array
w = w/sum(w);                                 % normalize analysis window
sw = hanning(M);                              % synthesis window
sw = sw./sum(sw);
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);              % window the input sound
  fftbuffer(:) = 0;                           % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);       % zero-phase fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                         % compute the FFT
  mX = 20*log10(abs(X(1:N2)));    % magnitude spectrum of positive frequencies
  pX = unwrap(angle(X(1:N2)));                % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
         .* (mX(2:N2-1)>mX(1:N2-2)));         % peaks
  pmag = mX(ploc);                            % magnitude of peaks
  pphase = pX(ploc);                          % phase of peaks
  %-----synthesis-----%
  Y = zeros(N,1); % initialize output spectrum
  Y(ploc) = 10.^(pmag/20).*exp(i.*pphase);      % generate positive freq.
  Y(N+2-ploc) = 10.^(pmag/20).*exp(-i.*pphase); % generate negative freq.
  fftbuffer = real(ifft(Y));                  % real part of the inverse FFT
  yw((M+1)/2:M) = fftbuffer(1:(M+1)/2);       % undo zero phase window
  yw(1:(M-1)/2) = fftbuffer(N-(M-1)/2+1:N);
  y(pin-hM:pin+hM) = y(pin-hM:pin+hM) + H*N*sw.*yw(1:M);     % overlap-add
  pin  = pin+H;                               % advance sound pointer
end
```

This code is mostly the same as the one used for the STFT, with the addition of a peak-detection step based on a magnitude threshold, that is, we only detect peaks whose magnitude is greater than a magnitude threshold specified by the user. Then, in the synthesis stage, the spectrum used to compute the inverse-FFT has values only at the bins of the peaks, and the rest have zero magnitude. It can be shown that each of these isolated bins is precisely the transform of a stationary sinusoid (of the frequency corresponding to the bin) multiplied by a rectangular window. Therefore, we smooth the resulting signal frame with a synthesis window, $sw(n)$, to have smoother overlap-add

behavior. Nevertheless, this implementation of a sinusoidal model is very simple and thus it is not appropriate for most applications. The next implementation is much more useful.

10.3.3 Spectral sinusoids

In order to improve the implementation of the sinusoidal model, we have to use more refined methods for estimating the sinusoidal parameters and we have to implement an accurate synthesis of time-varying sinusoids. The diagram of the complete system is shown in Figure 10.8.

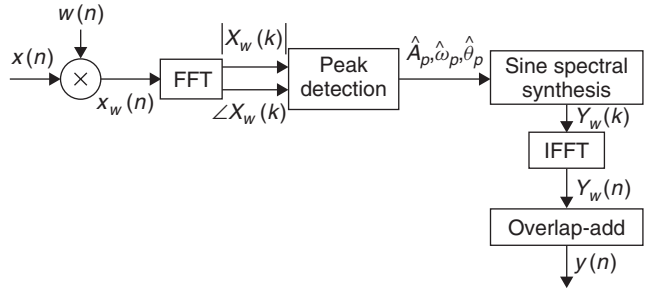


Figure 10.8 Block diagram of an analysis/synthesis system based on the sinusoidal model.

An efficient spectral interpolation scheme to better measure peak frequencies and magnitudes is to zero-pad only enough so that quadratic (or other simple) spectral interpolation, by using samples immediately surrounding the maximum-magnitude sample, suffices to refine the estimate to 0.1% accuracy. This is illustrated in Figure 10.9, where magnitudes are expressed in dB.

For a spectral peak located at bin k_p, let us define $\alpha = X_w^{dB}\left(k_p - 1\right)$, $\beta = X_w^{dB}\left(k_p\right)$ and $\gamma = X_w^{dB}\left(k_p + 1\right)$. The center of the parabola in bins is $\hat{k}_p = k_p + (\alpha - \gamma)/2\,(\alpha - 2\beta + \gamma)$, and the estimated amplitude $\hat{a}_p = \beta - (\alpha - \gamma)^2/8\,(\alpha - 2\beta + \gamma)$. Then the phase value of the peak is

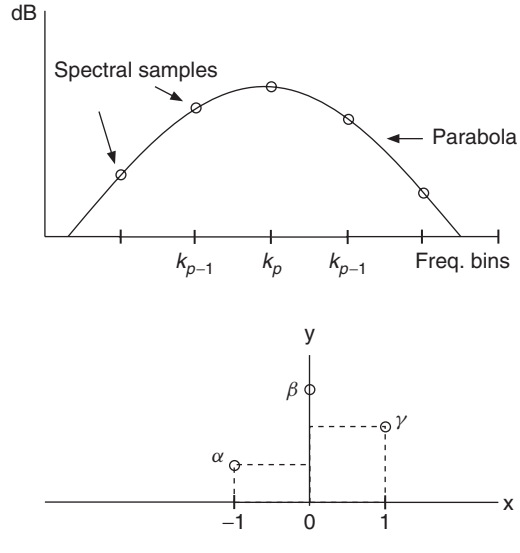


Figure 10.9 Parabolic interpolation in the peak-detection process.

measured by reading the value of the unwrapped phase spectrum at the position resulting from the frequency of the peak. This is a good first approximation, but it is still not the ideal solution, since significant deviations in amplitude and phase can be found even in the case of simple linear frequency modulations.

Here we include the code to perform the parabolic interpolation on the spectral peaks:

M-file 10.4 (peakinterp.m)

```
function [iploc, ipmag, ipphase] = peakinterp(mX, pX, ploc)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Parabolic interpolation of spectral peaks
% mX: magnitude spectrum, pX: phase spectrum, ploc: locations of peaks
% iploc, ipmag, ipphase: interpolated values
% note that ploc values are assumed to be between 2 and length(mX)-1
val = mX(ploc);                                    % magnitude of peak bin
lval = mX(ploc-1);                                 % magnitude of bin at left
rval= mX(ploc+1);                                  % magnitude of bin at right
iploc = ploc + .5*(lval-rval)./(lval-2*val+rval);  % center of parabola
ipmag = val-.25*(lval-rval).*(iploc-ploc);         % magnitude of peaks
ipphase = interp1(1:length(pX),pX,iploc,'linear'); % phase of peaks
```

In order to decide whether a peak 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 to improve the estimation of the spectral peak parameters [DH97]. More sophisticated methods make use of the spectral shape details and derive descriptors to distinguish between sinusoidal peaks, sidelobes and noise (e.g., [ZRR07]). Several other approaches target amplitude and frequency modulated sinusoids and are able to estimate their modulation parameters (e.g., [MB10]).

Peak continuation

Once the spectral peaks of a frame have been detected, a peak-continuation algorithm can organize the peaks into frequency trajectories, where each trajectory models a time-varying sinusoid (see Figure 10.10).

Several strategies have been explored during the last decades with the aim of connecting the sinusoidal components in the best possible way. McAulay and Quatieri proposed a simple sinusoidal continuation algorithm based on finding, for each spectral peak, the closest one in frequency in the next frame [MQ86]. Serra added to the continuation algorithm a set of frequency guides used to create sinusoidal trajectories [Ser89]. The frequency guide values were obtained from the peak values and their context, such as surrounding peaks and fundamental frequency. In the case of harmonic sounds, these guides were initialized according to the harmonic series of the estimated fundamental frequency. The trajectories were computed by assigning to each guide the closest peak in frequency.

There are peak-continuation methods based on hidden Markov models, which seem to be very valuable for tracking partials in polyphonic signals and complex inharmonic tones, for instance [DGR93]. Another interesting approach proposed by Peeters is to use a non-stationary sinusoid model, with linearly varying amplitude and frequency [Pee01]. Within that framework, the continuation algorithm focuses on the continuation of the value and first derivative for both amplitude and frequency polynomial parameters, combined with a measure of sinusoidality. Observation and transitions probabilities are defined, and a Viterbi algorithm is proposed for computing the sinusoidal trajectories.

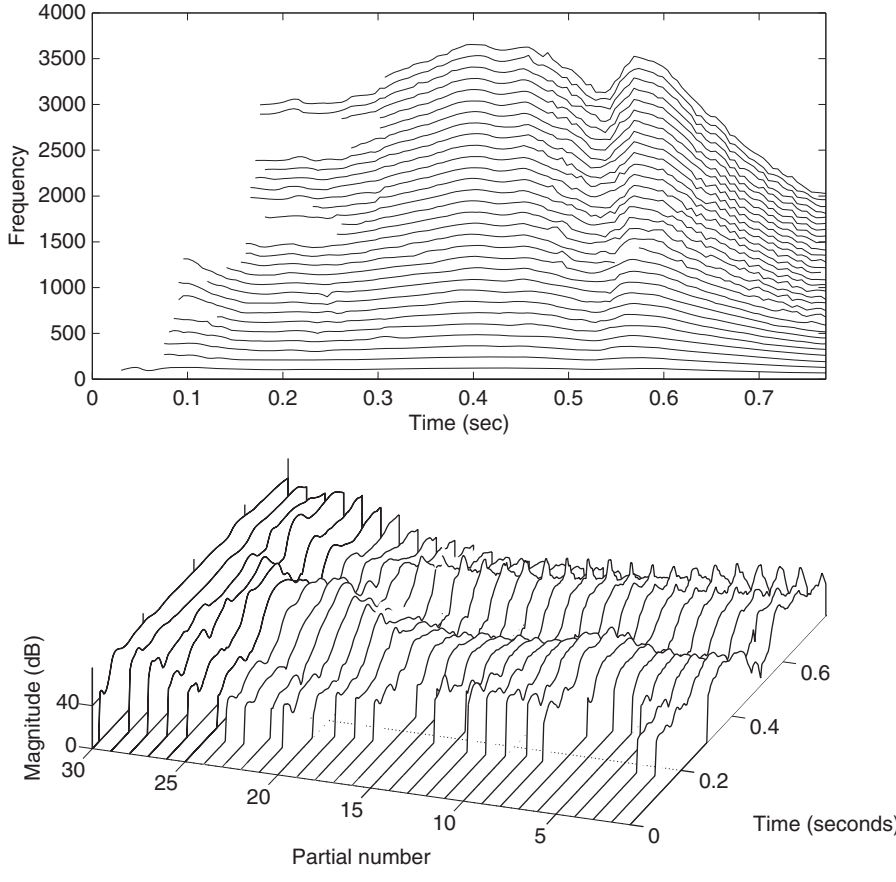


Figure 10.10 Frequency trajectories resulting from the sinusoidal analysis of a vocal sound.

Sinusoidal synthesis

Once all spectral peaks are identified and ordered, we can start the synthesis part, thus generating the sinusoids that correspond to the connected peaks. The sinusoids can be generated in the time domain with additive synthesis, controlling the instantaneous frequency and amplitude of a bank of oscillators, as shown in Figure 10.11. For this case, we define synthesis frames $d^l(m)$ as the segments determined by consecutive analysis times. For each synthesis frame, the instantaneous amplitude $\hat{A}^l(m)$ of an oscillator is obtained by linear interpolation of the surrounding amplitude estimations,

$$\hat{A}^l(m) = \hat{A}^{l-1} + \frac{\left(\hat{A}^l - \hat{A}^{l-1}\right)}{H} m, \tag{10.9}$$

where $m = 0, 1, \ldots, H-1$ is the time sample within the frame.

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

$$\hat{\omega}^l(m) = \hat{\omega}^{l-1} + \frac{\left(\hat{\omega}^l - \hat{\omega}^{l-1}\right)}{H} m, \tag{10.10}$$

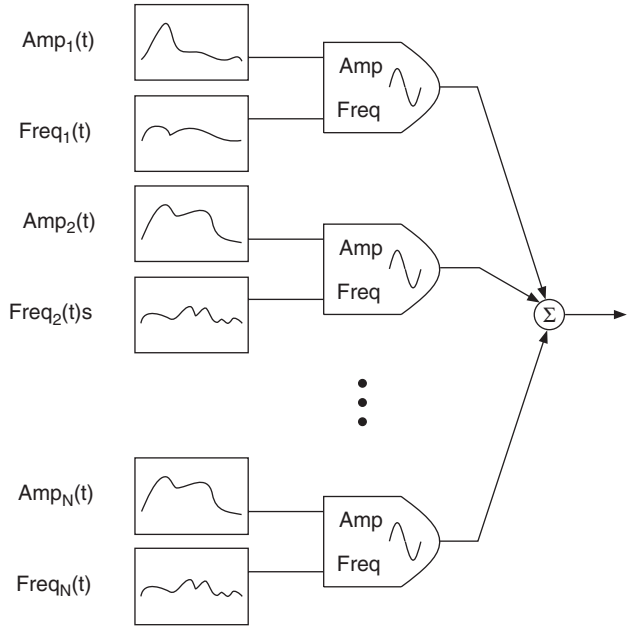


Figure 10.11 Additive synthesis block diagram.

and the instantaneous phase is

$$\hat{\theta}^l(m) = \hat{\theta}^{l-1}(H-1) + \sum_{s=0}^{m} \hat{\omega}^l(s). \tag{10.11}$$

Finally, the synthesis equation of a frame becomes

$$d^l(m) = \sum_{r=1}^{R^l} \hat{A}_r^l(m) \cos\left[\hat{\theta}_r^l(m)\right], \tag{10.12}$$

where r is the oscillator index and R^l the number of sinusoids existing on the lth frame. The output sound $d(n)$ is obtained by adding all frame segments together,

$$d(n) = \sum_{l=0}^{L-1} d^l(n - lH). \tag{10.13}$$

A much more efficient implementation of additive synthesis, where the instantaneous phase is not preserved, is based on the inverse-FFT [RD92]. While this approach loses 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 stationary 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 amplitude relevance. A sinusoid can be approximated in the spectral domain by computing 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 into 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.

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 as 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 used to generate per sinusoid. A good window choice is the Blackman-Harris 92 dB (BH92) because 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, and thus very high frame rates. A solution to this problem is to undo the effect of the window by dividing the result of the IFFT by it (in the time domain) and applying an appropriate overlapping window (e.g., triangular) before performing the overlap-add process. This gives a good time-frequency compromise. However, it is desirable to use a hop-size significantly smaller than half the synthesis window length, so an overlap greater than 50%. Otherwise, large gains are applied at the window edges, which might introduce undesired artifacts at synthesis. This is illustrated in Figure 10.12, where a triangular window is used. Note that for a 50% overlap ($Ns = 1024$, $H = 512$) the gain at the window edges reaches values greater than 150.

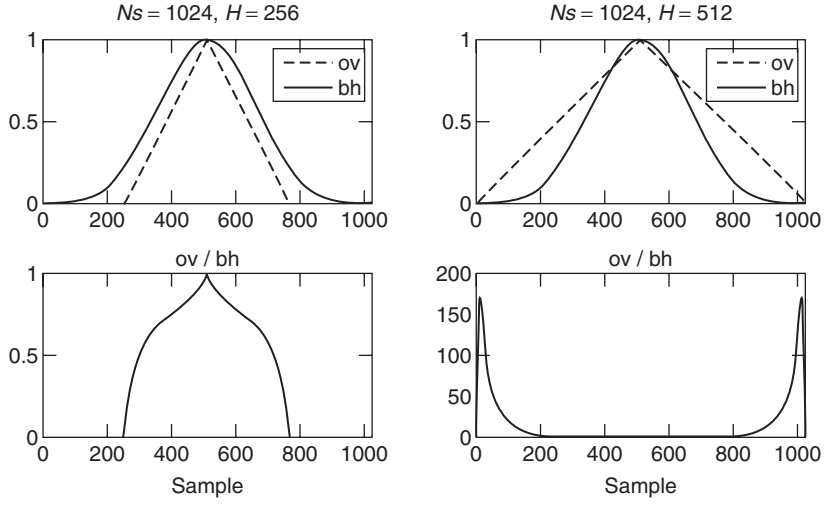


Figure 10.12 Overlapping strategy based on dividing by the synthesis window and multiplying by an overlapping window. Undesired artifacts can be introduced at synthesis if the overlap is not significantly greater than 50%.

The zero-centered BH92 window is defined as

$$w_{BH92}(n) = 0.35875 + 0.48829\cos\left(\frac{2\pi n}{N}\right) + 0.14128\cos\left(\frac{4\pi n}{N}\right) + \\ +0.01168\cos\left(\frac{6\pi n}{N}\right), \tag{10.14}$$

where N is the window length. Since each cosine function is the sum of two complex exponentials, then its transform can be expressed by a sum of rectangular window transforms (or Dirichlet kernels) shifted to the cosine (positive and negative) frequencies. The following **MATLAB** code implements the computation of the Blackman-Harris 92 dB transform, which will be used to generate sinusoids in the spectral domain:

M-file 10.5 (genbh92lobe.m)

```
function y = genbh92lobe(x)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Calculate transform of the Blackman-Harris 92dB window
% x: bin positions to compute (real values)
% y: transform values
N = 512;
f = x*pi*2/N;                                          % frequency sampling
df = 2*pi/N;
y = zeros(size(x));                                    % initialize window
consts = [.35875, .48829, .14128, .01168];             % window constants
for m=0:3
    y = y + consts(m+1)/2*(D(f-df*m,N)+D(f+df*m,N));   % sum Dirichlet kernels
end
y = y/N/consts(1);                                     % normalize
end

function y = D(x,N)
% Calculate rectangular window transform (Dirichlet kernel)
y = sin(N*x/2)./sin(x/2);
y(find(y~=y))=N;                                       % avoid NaN if x==0
end
```

Once we can generate a single sinusoid in the frequency domain, we can also generate a complete complex spectrum of a series of sinusoids from their frequency, magnitude and phase values.

M-file 10.6 (genspecsines.m)

```
function Y = genspecsines(ploc, pmag, pphase, N)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Compute a spectrum from a series of sine values
% iploc, ipmag, ipphase: sine locations, magnitudes and phases
% N: size of complex spectrum
% Y: generated complex spectrum of sines
Y =zeros(N,1); % initialize output spectrum
hN = N/2+1;                              % size of positive freq. spectrum
for i=1:length(ploc);                    % generate all sine spectral lobes
  loc = ploc(i);                         % location of peak (zero-based indexing)
                                         % it should be in range ]0,hN-1[
  if (loc<=1||loc>=hN-1) continue; end; % avoid frequencies out of range
  binremainder = round(loc)-loc;
  lb = [binremainder-4:binremainder+4]'; % main lobe (real value) bins to read
  lmag = genbh92lobe(lb)*10.^(pmag(i)/20); % lobe magnitudes of the
                                         % complex exponential
  b = 1+[round(loc)-4:round(loc)+4]';    % spectrum bins to fill
                                         % (1-based indexing)
  for m=1:9
      if (b(m)<1)                        % peak lobe croses DC bin
          Y(2-b(m)) = Y(2-b(m)) + lmag(m)*exp(-1i*pphase(i));
      elseif (b(m)>hN)                   % peak lobe croses Nyquist bin
          Y(2*hN-b(m)) = Y(2*hN-b(m)) + lmag(m)*exp(-1i*pphase(i));
      else                               % peak lobe in positive freq. range
          Y(b(m)) = Y(b(m)) + lmag(m)*exp(1i*pphase(i)) ...
                    + lmag(m)*exp(-1i*pphase(i))*(b(m)==1||b(m)==hN);
      end
  end
  Y(hN+1:end) = conj(Y(hN-1:-1:2));      % fill the rest of the spectrum
end
```

In this code we place each sinusoid in its spectral location with the right amplitude and phase. What complicates the code a bit is the inclusion of special conditions when spectral lobe values cross the DC or the Nyquist bin locations.

Now that we have an improved implementation for detecting and synthesizing sinusoids we can implement a complete analysis/synthesis system based on this sinusoidal model.

M-file 10.7 (sinemodel.m)

```
function y = sinemodel(x, w, N, t)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the sinusoidal model
% x: input sound, w: analysis window (odd size), N: FFT size,
% t: threshold in negative dB, y: output sound
M = length(w);                                  % analysis window size
Ns= 1024;                                       % FFT size for synthesis (even)
H = 256;                                        % analysis/synthesishop size
N2= N/2+1;                                      % size of positive spectrum
soundlength = length(x);                        % length of input sound array
hNs = Ns/2;                                     % half synthesis window size
hM = (M-1)/2;                                   % half analysis window size
pin = max(hNs+1,1+hM); % initialize sound pointer to middle of analysis window
pend = soundlength-max(hNs,hM);                   % last sample to start a frame
fftbuffer = zeros(N,1);                         % initialize buffer for FFT
y = zeros(soundlength,1);                             % initialize output array
w = w/sum(w);                                   % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                             % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;                      % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);                        % synthesis window
bh = bh ./ sum(bh);                             % normalize synthesis window
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);                % window the input sound
  fftbuffer(:) = 0;                             % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);         % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                                % compute the FFT
  mX = 20*log10(abs(X(1:N2)));    % magnitude spectrum of positive frequencies
  pX = unwrap(angle(X(1:N/2+1)));                % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                  .* (mX(2:N2-1)>mX(1:N2-2)));     % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);     % refine peak values
  %-----synthesis-----%
  plocs = (ploc-1)*Ns/N;             % adapt peak locations to synthesis FFT
  Y = genspecsines(plocs,pmag,pphase,Ns);          % generate spec sines
  yw = fftshift(real(ifft(Y)));                    % time domain of sinusoids
  y(pin-hNs:pin+hNs-1) = y(pin-hNs:pin+hNs-1) + sw.*yw(1:Ns);    % overlap-add
  pin  = pin+H;                                    % advance the sound pointer
end
```

The basic difference from the analysis/synthesis implementation based on spectral peaks is in the improvement of the peak detection using a parabolic interpolation, peakinterp, and on the implementation of the synthesis part. We now have a real additive synthesis section, which is completely independent of the analysis part. Thus it is no longer an FFT transform followed by its inverse-FFT transform. We could have implemented an additive synthesis in the time domain, but instead we have implemented an additive synthesis in the frequency domain, using the IFFT. We have fixed the synthesis FFT-size, 1204, and hop-size, 256, and they are independent of the

analysis window type, window size and FFT size, which are defined by the user as input parameters of the function and are set depending on the sound to be processed. Since this method generates stationary sinusoids, there is no need to estimate the instantaneous phase or to interpolate the sinusoidal parameters, and therefore we can omit the peak-continuation algorithm. This would not be the case if we wanted to apply transformations.

10.3.4 Spectral harmonics

A very useful constraint to be included in the sinusoidal model is to restrict the sinusoids to being harmonic partials, thus to assume that the input sound is monophonic and harmonic. With this constraint it should be possible to identify the fundamental frequency, F_0, at each frame, and to have a much more compact and flexible spectral representation.

Given this restriction and the set of spectral peaks of a frame, with magnitude and frequency values for each one, there are many possible F_0 estimation strategies, none of them perfect, e.g., [Hes83, MB94, Can98]. An obvious approach is to define F_0 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 F_0 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 of 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 estimator used.

In the framework of many spectral models 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 frequency 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 the two-way mismatch algorithm for fundamental frequency detection:

M-file 10.8 (`f0detectiontwm.m`)

```
function f0 = f0detectiontwm(mX, fs, ploc, pmag, ef0max, minf0, maxf0)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Fundamental frequency detection function
% mX: magnitude spectrum, fs: sampling rate, ploc, pmag: peak loc and mag,
% ef0max: maximim error allowed, minf0: minimum f0, maxf0: maximum f0
% f0: fundamental frequency detected in Hz
N = length(mX)*2;                       % size of complex spectrum
nPeaks = length(ploc);                  % number of peaks
f0 = 0;                                 % initialize output
if(nPeaks>3)            % at least 3 peaks in spectrum for trying to find f0
  nf0peaks = min(50,nPeaks);            % use a maximum of 50 peaks
  [f0,f0error] = TWM(ploc(1:nf0peaks),pmag(1:nf0peaks),N,fs,minf0,maxf0);
  if (f0>0 && f0error>ef0max)           % limit the possible error by ethreshold
    f0 = 0;
  end
end;
```

```
function [f0, f0error] = TWM (ploc, pmag, N, fs, minf0, maxf0)
% Two-way mismatch algorithm (by Beauchamp&Maher)
% ploc, pmag: peak locations and magnitudes, N: size of complex spectrum
% fs: sampling rate of sound, minf0: minimum f0, maxf0: maximum f0
% f0: fundamental frequency detected, f0error: error measure
pfreq = (ploc-1)/N*fs;                    % frequency in Hertz of peaks
[zvalue,zindex] = min(pfreq);
if (zvalue==0)                            % avoid zero frequency peak
  pfreq(zindex) = 1;
  pmag(zindex) = -100;
end
ival2 = pmag;
[Mmag1,Mloc1] = max(ival2);               % find peak with maximum magnitude
ival2(Mloc1) = -100;                      % clear max peak
[Mmag2,Mloc2]= max(ival2);                % find second maximum magnitude peak
ival2(Mloc2) = -100;                      % clear second max peak
[Mmag3,Mloc3]= max(ival2);                % find third maximum magnitude peak
nCand = 3;                % number of possible f0 candidates for each max peak
f0c = zeros(1,3*nCand);                   % initialize array of candidates
f0c(1:nCand)=(pfreq(Mloc1)*ones(1,nCand))./((nCand+1-(1:nCand))); % candidates
f0c(nCand+1:nCand*2)=(pfreq(Mloc2)*ones(1,nCand))./((nCand+1-(1:nCand)));
f0c(nCand*2+1:nCand*3)=(pfreq(Mloc3)*ones(1,nCand))./((nCand+1-(1:nCand)));
f0c = f0c((f0c<maxf0)&(f0c>minf0));       % candidates within boundaries
if (isempty(f0c))                         % if no candidates exit
  f0 = 0; f0error=100;
  return
end
harmonic = f0c;
ErrorPM = zeros(fliplr(size(harmonic)));          % initialize PM errors
MaxNPM = min(10,length(ploc));
for i=1:MaxNPM % predicted to measured mismatch error
  difmatrixPM = harmonic' * ones(size(pfreq))';
  difmatrixPM = abs(difmatrixPM-ones(fliplr(size(harmonic)))*pfreq');
  [FreqDistance,peakloc] = min(difmatrixPM,[],2);
  Ponddif = FreqDistance .* (harmonic'.^(-0.5));
  PeakMag = pmag(peakloc);
  MagFactor = 10.^((PeakMag-Mmag1)./20);
  ErrorPM = ErrorPM+(Ponddif+MagFactor.*(1.4*Ponddif-0.5));
  harmonic = harmonic+f0c;
end
ErrorMP = zeros(fliplr(size(harmonic)));          % initialize MP errors
MaxNMP = min(10,length(pfreq));
for i=1:length(f0c)                   % measured to predicted mismatch error
  nharm = round(pfreq(1:MaxNMP)/f0c(i));
  nharm = (nharm>=1).*nharm + (nharm<1);
  FreqDistance = abs(pfreq(1:MaxNMP) - nharm*f0c(i));
  Ponddif = FreqDistance.* (pfreq(1:MaxNMP).^(-0.5));
  PeakMag = pmag(1:MaxNMP);
  MagFactor = 10.^((PeakMag-Mmag1)./20);
  ErrorMP(i) = sum(MagFactor.*(Ponddif+MagFactor.*(1.4*Ponddif-0.5)));
end
Error = (ErrorPM/MaxNPM) + (0.3*ErrorMP/MaxNMP);  % total errors
[f0error, f0index] = min(Error);                  % get the smallest error
f0 = f0c(f0index);                                % f0 with the smallest error
```

There also exist many time-domain methods for estimating the fundamental frequency. It is worth mentioning the Yin algorithm, which offers excellent performance in many situations [CK02]. It is based on finding minima in the cumulative mean normalized difference function

$d'(\tau)$, defined as

$$d(\tau) = \sum_{j=t+1}^{t+W} (x(j) - x(j+\tau))^2 \tag{10.15}$$

$$d'(\tau) = \begin{cases} 1 & if\ \tau = 0 \\ \frac{\tau \cdot d(\tau)}{\sum_{j=1}^{\tau} d(j)} & otherwise, \end{cases} \tag{10.16}$$

where x is the input signal, τ is the lag time, and W is the window size. Ideally, this function exhibits local minima at lag times corresponding to the fundamental period and its multiples. More refinements are described in the referenced article, although here we will only implement the basic procedure.

The following **MATLAB** code implements a simplified version of the Yin algorithm for fundamental frequency detection. We have included some optimizations in the calculations, such as performing sequential cumulative sums by adding and subtracting, respectively, the new and initial samples, or using the `xcorr` **MATLAB** function.

M-file 10.9 (`f0detectionyin.m`)

```
function f0 = f0detectionyin(x,fs,ws,minf0,maxf0)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Fundamental frequency detection function
% x: input signal, fs: sampling rate, ws: integration window length
% minf0: minimum f0, maxf0: maximum f0
% f0: fundamental frequency detected in Hz
maxlag = ws-2;                              % maximum lag
th = 0.1;                                   % set threshold
d = zeros(maxlag,1);                        % init variable
d2 = zeros(maxlag,1);                       % init variable
% compute d(tau)
x1 = x(1:ws);
cumsumx = sum(x1.^2);
cumsumxl = cumsumx;
xy = xcorr(x(1:ws*2),x1);
xy = xy(ws*2+1:ws*3-2);
for lag=0:maxlag-1
    d(1+lag) = cumsumx + cumsumxl - 2*xy(1+lag);
    cumsumxl = cumsumxl - x(1+lag).^2 + x(1+lag+ws+1)^2;
end
cumsum = 0;
% compute d'(tau)
d2(1) = 1;
for lag=1:maxlag-1
    cumsum = cumsum + d(1+lag);
    d2(1+lag) = d(1+lag)*lag./cumsum;
end
% limit the search to the target range
minf0lag = 1+round(fs./minf0);              % compute lag corresponding to minf0
maxf0lag = 1+round(fs./maxf0);              % compute lag corresponding to maxf0
if (maxf0lag>1 && maxf0lag<maxlag)
    d2(1:maxf0lag) = 100;                   % avoid lags shorter than maxf0lag
end
if (minf0lag>1 && minf0lag<maxlag)
    d2(minf0lag:end) = 100;                 % avoid lags larger than minf0lag
end
% find the best candidate
mloc = 1 + find((d2(2:end-1)<d2(3:end)).*(d2(2:end-1)<d2(1:end-2)));  % minima
```

```
candf0lag = 0;
if (length(mloc)>0)
    I = find(d2(mloc)<th);
    if (length(I)>0)
        candf0lag = mloc(I(1));
    else
        [Y,I2] = min(d2(mloc));
        candf0lag = mloc(I2);
    end
    candf0lag = candf0lag;                % this is zero-based indexing
    if (candf0lag>1 & candf0lag<maxlag)
        % parabolic interpolation
        lval = d2(candf0lag-1);
        val = d2(candf0lag);
        rval= d2(candf0lag+1);
        candf0lag = candf0lag + .5*(lval-rval)./(lval-2*val+rval);
    end
end
ac = min(d2);
f0lag = candf0lag-1;                      % convert to zero-based indexing
f0 = fs./f0lag;                           % compute candidate frequency in Hz
if (ac > 0.2)                             % voiced/unvoiced threshold
    f0 = 0;                               % set to unvoiced
end
```

When a relevant fundamental frequency is identified, we can decide to what harmonic number each of the peaks belongs and thus restrict the sinusoidal components to be only the harmonic ones. The diagram of the complete system is shown in Figure 10.13.

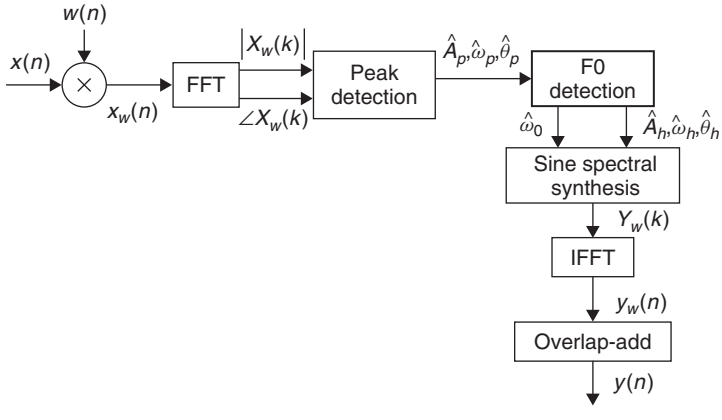


Figure 10.13 Block diagram of an analysis/synthesis system based on the harmonic sinusoidal model.

We can now add these steps into the implementation of the sinusoidal model represented in the previous section to implement a system that works just for harmonic sounds. Next is the complete **MATLAB** code for it.

M-file 10.10 (harmonicmodel.m)

```
function y = harmonicmodel(x, fs, w, N, t, nH, minf0, maxf0, f0et, maxhd)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the sinusoidal harmonic model
```

```
% x: input sound, fs: sampling rate, w: analysis window (odd size),
% N: FFT size (minimum 512), t: threshold in negative dB,
% nH: maximum number of harmonics, minf0: minimum f0 frequency in Hz,
% maxf0: maximim f0 frequency in Hz,
% f0et: error threshold in the f0 detection (ex: 5),
% maxhd: max. relative deviation in harmonic detection (ex: .2)
% y: output sound
M = length(w);                              % analysis window size
Ns= 1024;                                   % FFT size for synthesis
H = 256;                                    % hop size for analysis and synthesis
N2 = N/2+1;                                 % size postive spectrum
soundlength = length(x);                    % length of input sound array
hNs = Ns/2;                                 % half synthesis window size
hM = (M-1)/2;                               % half analysis window size
pin = max(hNS+1,1+hM);   % initialize sound pointer to middle of analysis window
pend = soundlength-max(hNs,hM);                    % last sample to start a frame
fftbuffer = zeros(N,1);                     % initialize buffer for FFT
y = zeros(soundlength+Ns/2,1);              % output sound
w = w/sum(w);                               % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                         % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;                  % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);                    % synthesis window
bh = bh ./ sum(bh);                         % normalize synthesis window
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);            % window the input sound
  fftbuffer(:) = 0;                         % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);  % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                       % compute the FFT
  mX = 20*log10(abs(X(1:N2)));              % magnitude spectrum
  pX = unwrap(angle(X(1:N/2+1)));           % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                  .* (mX(2:N2-1)>mX(1:N2-2)));            % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);            % refine peak values
  f0 = f0detectiontwm(mX,fs,ploc,pmag,f0et,minf0,maxf0);  % find f0
  hloc = zeros(nH,1);                          % initialize harmonic locations
  hmag = zeros(nH,1)-100;                      % initialize harmonic magnitudes
  hphase = zeros(nH,1);                        % initialize harmonic phases
  hf = (f0>0).*(f0.*(1:nH));                   % initialize harmonic frequencies
  hi = 1;                                      % initialize harmonic index
  npeaks = length(ploc);                       % number of peaks found
  while (f0>0 && hi<=nH && hf(hi)<fs/2)      % find harmonic peaks
    [dev,pei] = min(abs((ploc(1:npeaks)-1)/N*fs-hf(hi)));       % closest peak
    if ((hi==1 || ~any(hloc(1:hi-1)==ploc(pei))) && dev<maxhd*hf(hi))
      hloc(hi) = ploc(pei);                    % harmonic locations
      hmag(hi) = pmag(pei);                    % harmonic magnitudes
      hphase(hi) = pphase(pei);                % harmonic phases
    end
    hi = hi+1;                                 %increase harmonic index
  end
  hloc(1:hi-1) = (hloc(1:hi-1)~=0).*((hloc(1:hi-1)-1)*Ns/N); % synth. locs
  %-----synthesis-----%
  Yh = genspecsines(hloc(1:hi-1),hmag,hphase,Ns);       % generate sines
  yh = fftshift(real(ifft(Yh)));                        % sines in time domain
  y(pin-hNs:pin+hNs-1) = y(pin-hNs:pin+hNs-1) + sw.*yh(1:Ns);    % overlap-add
  pin = pin+H;                                 % advance the input sound pointer
end
```

The basic differences from the previous implementation of the sinusoidal model are the detection of the F_0 using the function `f0detectiontwm`, and the detection of the harmonic peaks by first generating a perfect harmonic series from the identified F_0 and then searching the closest peaks for each harmonic value. This is a very simple implementation of a peak-continuation algorithm, but generally works for clean recordings of harmonic sounds. The major possible problem in using this implementation is in failing to detect the correct F_0 for a given sound, causing the complete system to fail.

In the previous analysis/synthesis code we used the two-way mismatch algorithm, but we could perfectly use the Yin algorithm instead. A substantial difference between both is that whereas the first one operates on the spectra information, Yin receives a time-domain segment as input. If we prefer to use Yin, we can replace the line where `f0detectiontwm` was called in the previous code for the following ones. Note that the window length, `yinws`, has to be larger than the period corresponding to the minimum fundamental frequency to estimate.

M-file 10.11 (`callyin.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
yinws = round(fs*0.015);           % using approx. a 15 ms window for yin
yinws = yinws+mod(yinws,2);        % make it even
yb = pin-yinws/2;
ye = pin+yinws/2+yinws;
if (yb<1 || ye>length(x))          % out of boundaries
   f0 = 0;
else
   f0 = f0detectionyin(x(yb:ye),fs,yinws,minf0,maxf0);
end
```

10.3.5 Spectral harmonics plus residual

Once we have identified the harmonic partials of a sound, we can subtract them from the original signal and obtain a residual component. This subtraction can be done either in the time domain or in the frequency domain. A time-domain approach requires first to 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. This is a type of synthesis that is computationally quite expensive. On the other hand, the sinusoidal subtraction in the spectral domain is in many cases computationally simpler, and can give similar results, in terms of accuracy, as the time-domain implementation. We have to understand that the sinusoidal information obtained 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 re-generate all the spectral 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, since these are not adequate for typical residual models. In this case, we should re-analyze the sound until we get a good residual, free of harmonic partials. Ideally, for monophonic signals, the resulting residual should be as close as possible to a stochastic signal.

The first step in obtaining the residual is to synthesize the sinusoids obtained as the output of the harmonic analysis. For a time-domain subtraction (see Figure 10.14) the synthesized signal will reproduce the instantaneous phase and amplitude of the partials of the original sound with a bank of oscillators. Different approaches are possible for computing the instantaneous phase, for instance

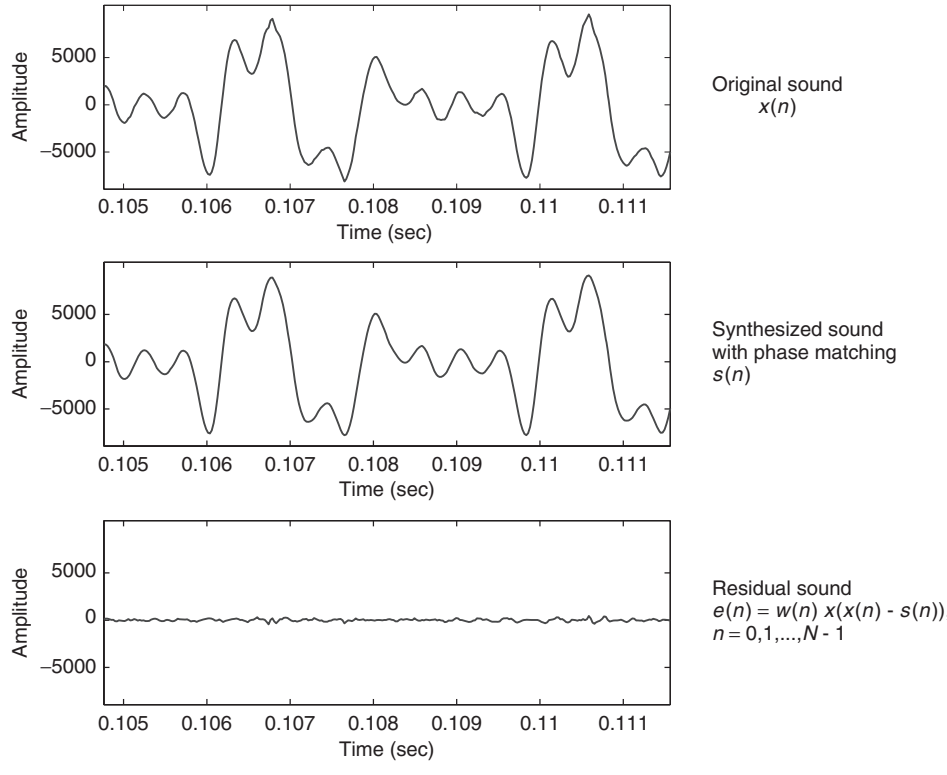


Figure 10.14 Time-domain subtraction.

[MQ86], thus being able to synthesize one frame 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. However, an efficient implementation can also be obtained by frequency-domain subtraction, where a spectral representation of stationary sinusoids is generated [RD92].

Next is an implementation of a complete system using a harmonic plus residual model and based on subtracting the harmonic component in the frequency domain (see the diagram in Figure 10.15).

M-file 10.12 (`hprmodel.m`)

```
function [y,yh,yr] = hprmodel(x,fs,w,N,t,nH,minf0,maxf0,f0et,maxhd)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the sinusoidal harmonic model
% x: input sound, fs: sampling rate, w: analysis window (odd size),
% N: FFT size (minimum 512), t: threshold in negative dB,
% nH: maximum number of harmonics, minf0: minimum f0 frequency in Hz,
% maxf0: maximim f0 frequency in Hz,
% f0et: error threshold in the f0 detection (ex: 5),
% maxhd: max. relative deviation in harmonic detection (ex: .2)
% y: output sound, yh: harmonic component, yr: residual component
M = length(w);                          % analysis window size
Ns = 1024;                              % FFT size for synthesis
H = 256;                                % hop size for analysis and synthesis
N2 = N/2+1;                             % half-size of spectrum
```

```
soundlength = length(x);                  % length of input sound array
hNs = Ns/2;                               % half synthesis window size
hM = (M-1)/2;                             % half analysis window size
pin = max(hNs+1,1+hM);   % initialize sound pointer to middle of analysis window
pend = soundlength-max(hM,hNs);            % last sample to start a frame
fftbuffer = zeros(N,1);                   % initialize buffer for FFT
yh = zeros(soundlength+Ns/2,1);           % output sine component
yr = zeros(soundlength+Ns/2,1);           % output residual component
w = w/sum(w);                             % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                       % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;                % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);                  % synthesis window
bh = bh ./ sum(bh);                       % normalize synthesis window
wr = bh;                                  % window for residual
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);          % window the input sound
  fftbuffer(:) = 0;                       % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);  % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                     % compute the FFT
  mX = 20*log10(abs(X(1:N2)));            % magnitude spectrum
  pX = unwrap(angle(X(1:N/2+1)));         % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                 .* (mX(2:N2-1)>mX(1:N2-2)));         % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);         % refine peak values
  f0 = f0detectiontwm(mX,fs,ploc,pmag,f0et,minf0,maxf0);  % find f0
  hloc = zeros(nH,1);                     % initialize harmonic locations
  hmag = zeros(nH,1)-100;                 % initialize harmonic magnitudes
  hphase = zeros(nH,1);                   % initialize harmonic phases
  hf = (f0>0).*(f0.*(1:nH));              % initialize harmonic frequencies
  hi = 1;                                 % initialize harmonic index
  npeaks = length(ploc);                  % number of peaks found
  while (f0>0 && hi<=nH && hf(hi)<fs/2)  % find harmonic peaks
    [dev,pei] = min(abs((ploc(1:npeaks)-1)/N*fs-hf(hi)));   % closest peak
    if ((hi==1 || ~any(hloc(1:hi-1)==ploc(pei))) && dev<maxhd*hf(hi))
      hloc(hi) = ploc(pei);               % harmonic locations
      hmag(hi) = pmag(pei);               % harmonic magnitudes
      hphase(hi) = pphase(pei);           % harmonic phases
    end
    hi = hi+1;                            % increase harmonic index
  end
  hloc(1:hi-1) = (hloc(1:hi-1)~=0).*((hloc(1:hi-1)-1)*Ns/N);   % synth. locs
  ri= pin-hNs;                       % input sound pointer for residual analysis
  xr = x(ri:ri+Ns-1).*wr(1:Ns);           % window the input sound
  Xr = fft(fftshift(xr));                 % compute FFT for residual analysis
  %-----synthesis-----%
  Yh = genspecsines(hloc(1:hi-1),hmag,hphase,Ns);            % generate sines
  Yr = Xr-Yh;                        % get the residual complex spectrum
  yhw = fftshift(real(ifft(Yh)));  % sines in time domain using inverse FFT
  yrw = fftshift(real(ifft(Yr)));  % residual in time domain using inverse FFT
  yh(ri:ri+Ns-1) = yh(ri:ri+Ns-1)+yhw(1:Ns).*sw;   % overlap-add for sines
  yr(ri:ri+Ns-1) = yr(ri:ri+Ns-1)+yrw(1:Ns).*sw;   % overlap-add for residual
  pin = pin+H;                                     % advance the sound pointer
end
y= yh+yr;                                          % sum sines and residual
```

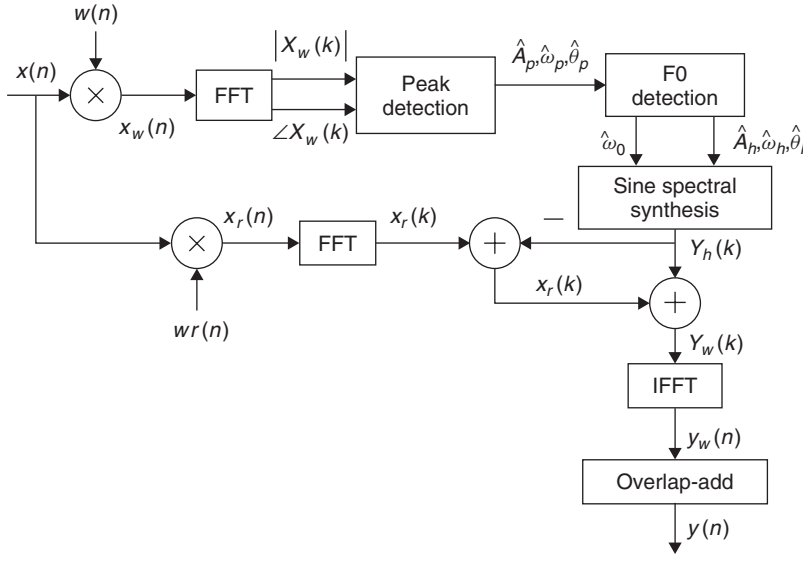


Figure 10.15 Block diagram of the harmonic plus residual analysis/synthesis system.

In order to subtract the harmonic component from the original sound in the frequency domain we have added a parallel FFT analysis of the input sound, with an analysis window which is the same than the one used in the synthesis step. Thus we can get a residual spectrum, `Xres`, and by simply computing its inverse FFT we obtain the time-domain residual. The synthesized signal, $y(n)$, is the sum of the harmonic and residual components. The function also returns the harmonic and residual components as separate signals.

10.3.6 Spectral harmonics plus stochastic residual

Once we have a decomposition of a sound into harmonic and residual components, from the residual signal we can continue our modeling strategy towards a more compact and flexible representation. When the harmonics have been well identified and subtracted the residual can be considered a stochastic signal ready to be parameterized. To model, or parameterize, the relevant parts of the residual component of a musical sound, such as the bow noise in stringed instruments or the breath noise in wind instruments, we need good time resolution and we can give up some of the frequency resolution required for modeling the harmonic component. Given the window requirements for the STFT analysis the harmonic 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. However, once the harmonic subtraction has been done, the temporal resolution to analyze the residual component can be improved by using a different analysis window (see Figure 10.16).

Since the harmonic analysis has been performed using long windows, the subtraction of the harmonic component generated from that analysis might result in smearing of the sharp discontinuities, like note attacks, which remain in the residual component. We can *fix* this smearing effect of the residual and try to preserve the sharpness of the attacks of the original sound. For example, the resulting time-domain residual can be 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 modified residual. We can also compare the synthesized harmonic signal with the original sound and whenever this signal has a greater energy than the

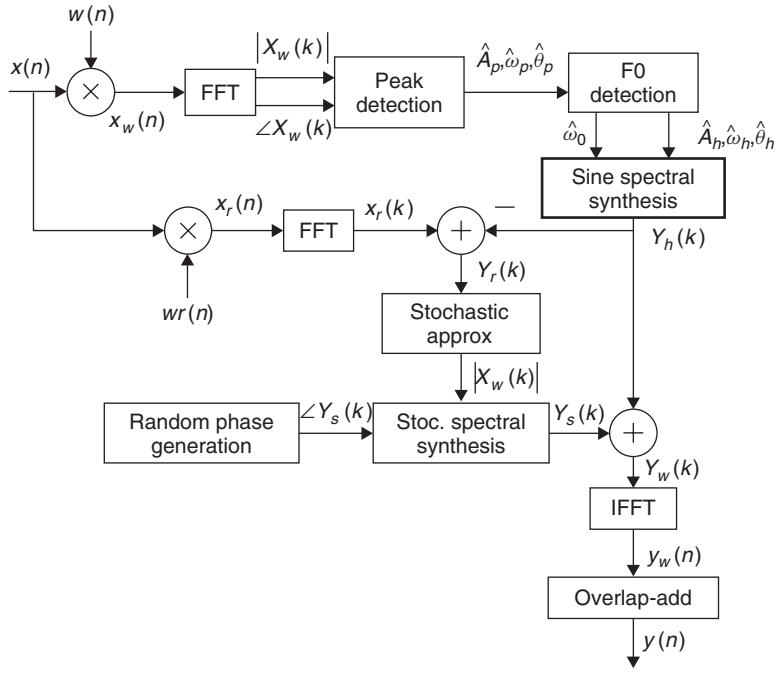


Figure 10.16 Block diagram of an analysis/synthesis system based on a harmonic plus stochastic model.

original waveform it means that a smearing of the harmonic component has been produced. This can be fixed a bit by scaling the amplitudes of the harmonic analysis in the corresponding frame using the difference between the original sound and the harmonic signal.

Residual analysis

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 (see Figure 10.17). 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 in the overall sound will be a sequence of these filters, i.e., a time-varying filter.

The filter design problem is generally solved by performing some sort of curve fitting in the magnitude spectrum of the current frame [Str80, Sed88]. Standard techniques are: spline interpolation [Cox71], the method of least squares [Sed88], or straight-line approximations.

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 [Mak75, MG75]. LPC is a popular technique used in speech research

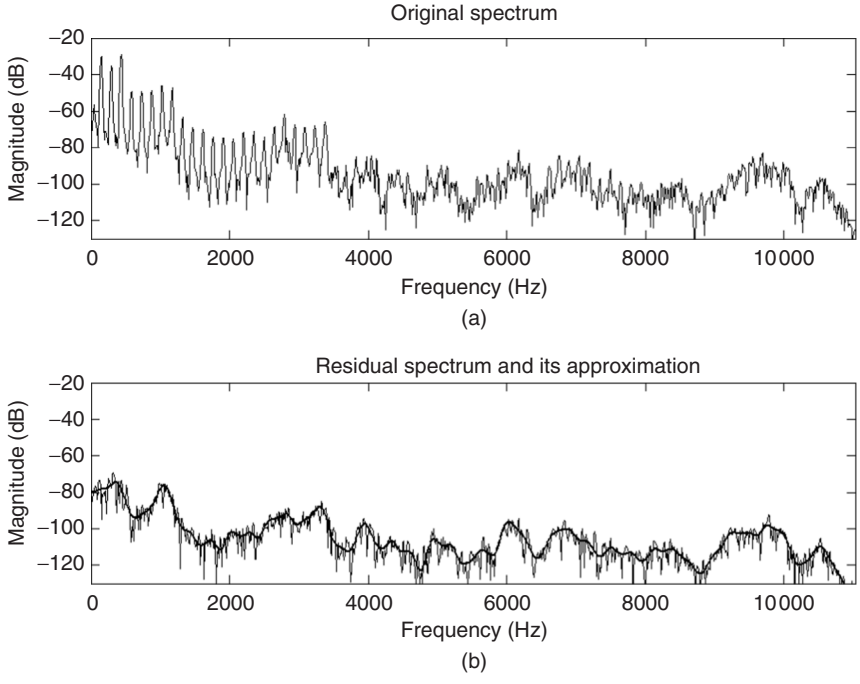


Figure 10.17 (a) Original spectrum. (b) Residual spectrum and approximation.

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 [Goo97].

Residual synthesis

From the stochastic model, parameterization of the residual component, we can synthesize a signal that should preserve the perceptually relevant characteristic of the residual signal. Like the sinusoidal synthesis, this synthesis can either be done in the time or in the frequency domains. Here we present a frequency-domain implementation. We generate a complete complex spectrum at each synthesis frame from the magnitude spectral envelope that approximated the residual.

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, 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 can instead approximate it in the frequency domain (see Figure 10.18) by creating a magnitude spectrum from the estimated envelope, or its transformation, and generating a random phase spectrum with new values at each frame in order to avoid periodicity.

Once the harmonic and stochastic spectral components have been generated, we can compute the IFFT of each one, and add the resulting time-domain signals. A more efficient alternative is to directly add the spectrum of the residual component to that of the sinusoids. However, in this case 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

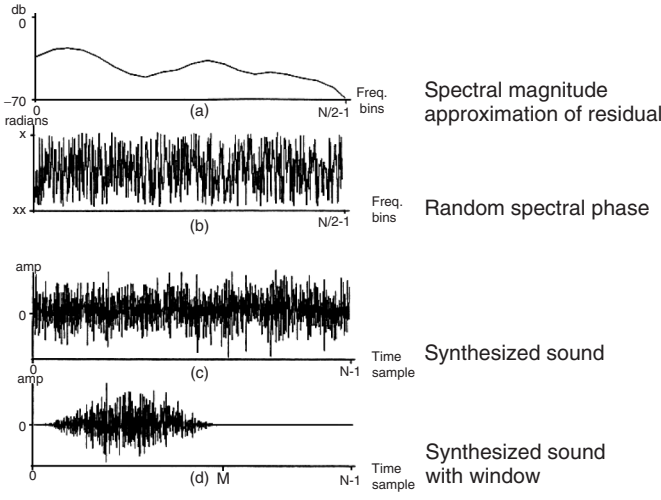


Figure 10.18 Stochastic synthesis.

smoothing consideration, but in the sinusoidal synthesis we have used a Blackman-Harris 92 dB, 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 92 dB by the noise spectrum accomplishes this, and there is only the need to use the main lobe of the window since that 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 92 dB and impose the triangular window. By an overlap-add process we combine successive frames to get the time-varying characteristics of the sound. In the implementation shown later on this is not done in order to be able to output the two components separately and thus be able to hear the harmonic and the stochastic components of the sound.

Several other approaches have been used for synthesizing the output of a sinusoidal plus stochastic analysis. These techniques, though, include modifications to the model as a whole. For instance, [FHC00] proposes to link the stochastic components to each sinusoid by using stochastic modulation to spread spectral energy away from the sinusoid's center frequency. With this technique each partial has an extra parameter, the bandwidth coefficient, which sets the balance between noise and sinusoidal energy.

Below is the code of a complete system using a harmonic plus stochastic model:

M-file 10.13 (hpsmodel.m)

```
function [y,yh,ys] = hpsmodel(x,fs,w,N,t,nH,minf0,maxf0,f0et,maxhd,stocf)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the sinusoidal harmonic model
% x: input sound, fs: sampling rate, w: analysis window (odd size),
% N: FFT size (minimum 512), t: threshold in negative dB,
% nH: maximum number of harmonics, minf0: minimum f0 frequency in Hz,
% maxf0: maximim f0 frequency in Hz,
% f0et: error threshold in the f0 detection (ex: 5),
% maxhd: max. relative deviation in harmonic detection (ex: .2)
% stocf: decimation factor of mag spectrum for stochastic analysis
% y: output sound, yh: harmonic component, ys: stochastic component
M = length(w);   % analysis window size
Ns = 1024;                              % FFT size for synthesis
```

```
H = 256;                                % hop size for analysis and synthesis
N2 = N/2+1;                             % half-size of spectrum
soundlength = length(x);                % length of input sound array
hNs = Ns/2;                             % half synthesis window size
hM = (M-1)/2;                           % half analysis window size
pin = max(hNs+1,1+hM);   % initialize sound pointer to middle of analysis window
pend = soundlength-max(hM,hNs);           % last sample to start a frame
fftbuffer = zeros(N,1);                 % initialize buffer for FFT
yh = zeros(soundlength+Ns/2,1);         % output sine component
ys = zeros(soundlength+Ns/2,1);         % output residual component
w = w/sum(w);                           % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                     % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;              % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);                % synthesis window
bh = bh ./ sum(bh);                     % normalize synthesis window
wr = bh;                                % window for residual
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
sws = H*hanning(Ns)/2;                  % synthesis window for stochastic
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);        % window the input sound
  fftbuffer(:) = 0;                     % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);  % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                   % compute the FFT
  mX = 20*log10(abs(X(1:N2)));          % magnitude spectrum
  pX = unwrap(angle(X(1:N/2+1)));       % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                 .* (mX(2:N2-1)>mX(1:N2-2)));          % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);          % refine peak values
  f0 = f0detectiontwm(mX,fs,ploc,pmag,f0et,minf0,maxf0);   % find f0
  hloc = zeros(nH,1);                   % initialize harmonic locations
  hmag = zeros(nH,1)-100;               % initialize harmonic magnitudes
  hphase = zeros(nH,1);                 % initialize harmonic phases
  hf = (f0>0).*(f0.*(1:nH));            % initialize harmonic frequencies
  hi = 1;                               % initialize harmonic index
  npeaks = length(ploc);                % number of peaks found
  while (f0>0 && hi<=nH && hf(hi)<fs/2)  % find harmonic peaks
    [dev,pei] = min(abs((ploc(1:npeaks)-1)/N*fs-hf(hi)));    % closest peak
    if ((hi==1 || ~any(hloc(1:hi-1)==ploc(pei))) && dev<maxhd*hf(hi))
      hloc(hi) = ploc(pei);             % harmonic locations
      hmag(hi) = pmag(pei);             % harmonic magnitudes
      hphase(hi) = pphase(pei);         % harmonic phases
    end
    hi = hi+1;                          % increase harmonic index
  end
  hloc(1:hi-1) = (hloc(1:hi-1)~=0).*((hloc(1:hi-1)-1)*Ns/N);  % synth. locs
  ri= pin-hNs;                      % input sound pointer for residual analysis
  xr = x(ri:ri+Ns-1).*wr(1:Ns);         % window the input sound
  Xr = fft(fftshift(xr));               % compute FFT for residual analysis
  Yh = genspecsines(hloc(1:hi-1),hmag,hphase,Ns);             % generate sines
  Yr = Xr-Yh;                           % get the residual complex spectrum
  mYr = abs(Yr(1:Ns/2+1));              % magnitude spectrum of residual
  mYsenv = decimate(mYr,stocf);         % decimate the magnitude spectrum
  %-----synthesis-----%
  mYs = interp(mYsenv,stocf);               % interpolate to original size
  roffset = ceil(stocf/2)-1;                % interpolated array offset
  mYs = [ mYs(1)*ones(roffset,1); mYs(1:Ns/2+1-roffset) ];
  pYs = 2*pi*rand(Ns/2+1,1);                % generate phase random values
  mYs1 = [mYs(1:Ns/2+1); mYs(Ns/2:-1:2)];   % create magnitude spectrum
  pYs1 = [pYs(1:Ns/2+1); -1*pYs(Ns/2:-1:2)]; % create phase spectrum
```

```
  Ys = mYs1.*cos(pYs1)+1i*mYs1.*sin(pYs1);   % compute complex spectrum
  yhw = fftshift(real(ifft(Yh)));            % sines in time domain using IFFT
  ysw = fftshift(real(ifft(Ys)));            % stoc. in time domain using IFFT
  yh(ri:ri+Ns-1) = yh(ri:ri+Ns-1)+yhw(1:Ns).*sw;   % overlap-add for sines
  ys(ri:ri+Ns-1) = ys(ri:ri+Ns-1)+ysw(1:Ns).*sws;  % overlap-add for stoch.
  pin = pin+H;                                     % advance the sound pointer
end
y= yh+ys; % sum sines and stochastic
```

This code extends the implementation of the harmonic plus residual with the approximation of the residual and the synthesis of a stochastic signal. It first approximates the residual magnitude spectrum using the function `decimate` from **MATLAB**. This function resamples a data array at a lower rate after lowpass filtering and using a decimation factor, `stocf`, given by the user. At synthesis a complete magnitude spectrum is generated by interpolating the approximated envelope using the function `interp`. This is a **MATLAB** function that re-samples data at a higher rate using lowpass interpolation. The corresponding phase spectrum is generated with a random number generator, the function `rand`. The output signal, y(n), is the sum of the harmonic and the stochastic components.

10.4 Effects

In this section we introduce a set of effects and transformations that can be added to some of the analysis-synthesis frameworks that have just been presented. To do that we first need to change a few things on the implementations given above in order to make them suitable for incorporating the code for the transformations. We have to add new variables to handle the synthesis data and for controlling effects. Other modifications in the code are related to peak continuation and phase propagation. We first focus on the sinusoidal plus residual model, and next we will use the harmonic-based model.

10.4.1 Sinusoidal plus residual

In the implementation of the sinusoidal plus residual model presented, now we use different variables for analysis and synthesis parameters and keep a history of the last frame values. This is required for continuing analysis spectral peaks and propagating synthesis phases. The peak continuation implementation is really simple: each current frame peak is connected to the closest peak in frequency of the last frame. The phase propagation is also straightforward: it assumes that the frequency of connected peaks evolves linearly between consecutive frames. Note that when computing the residual we have to be careful to subtract the analysis sinusoids (not the synthesis ones) to the input signal. Another significant change is that here we use the parameter `maxnS` to limit the number of sinusoids. With that restriction, peaks found are sorted by energy and only the first `maxnS` are taken into account.

M-file 10.14 (`spsmodel.m`)

```
function [y,yh,ys] = spsmodel(x,fs,w,N,t,maxnS,stocf)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the sinusoidal harmonic model
% x: input sound, fs: sampling rate, w: analysis window (odd size),
% N: FFT size (minimum 512), t: threshold in negative dB,
% maxnS: maximum number of sinusoids,
% stocf: decimation factor of mag spectrum for stochastic analysis
% y: output sound, yh: harmonic component, ys: stochastic component
M = length(w);                          % analysis window size
```

```
Ns = 1024;                                % FFT size for synthesis
H = 256;                                  % hop size for analysis and synthesis
N2 = N/2+1;                               % half-size of spectrum
soundlength = length(x);                  % length of input sound array
hNs = Ns/2;                               % half synthesis window size
hM = (M-1)/2;                             % half analysis window size
pin = max(hNs+1,1+hM);   % initialize sound pointer to middle of analysis window
pend = soundlength-max(hM,hNs);            % last sample to start a frame
fftbuffer = zeros(N,1);                   % initialize buffer for FFT
yh = zeros(soundlength+Ns/2,1);           % output sine component
ys = zeros(soundlength+Ns/2,1);           % output residual component
w = w/sum(w);                             % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                       % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;                % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);                  % synthesis window
bh = bh ./ sum(bh);                       % normalize synthesis window
wr = bh;                                  % window for residual
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
sws = H*hanning(Ns)/2;                % synthesis window for stochastic
lastysloc = zeros(maxnS,1);           % initialize synthesis harmonic locations
ysphase = 2*pi*rand(maxnS,1);         % initialize synthesis harmonic phases
fridx = 0;
while pin<pend
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);          % window the input sound
  fftbuffer = fftbuffer*0;                % reset buffer;
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);  % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                     % compute the FFT
  mX = 20*log10(abs(X(1:N2)));            % magnitude spectrum
  pX = unwrap(angle(X(1:N/2+1)));         % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                 .* (mX(2:N2-1)>mX(1:N2-2)));          % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);          % refine peak values
  % sort by magnitude
  [smag,I] = sort(pmag(:),1,'descend');
  nS = min(maxnS,length(find(smag>t)));
  sloc = ploc(I(1:nS));
  sphase = pphase(I(1:nS));
  if (fridx==0)
      % update last frame data for first frame
      lastnS = nS;
      lastsloc = sloc;
      lastsmag = smag;
      lastsphase = sphase;
  end
  % connect sinusoids to last frame lnS (lastsloc,lastsphase,lastsmag)
  sloc(1:nS) = (sloc(1:nS)~=0).*((sloc(1:nS)-1)*Ns/N);  % synth. locs
  lastidx = zeros(1,nS);
  for i=1:nS
    [dev,idx] = min(abs(sloc(i) - lastsloc(1:lastnS)));
    lastidx(i) = idx;
  end
  ri= pin-hNs;                      % input sound pointer for residual analysis
  xr = x(ri:ri+Ns-1).*wr(1:Ns);     % window the input sound
  Xr = fft(fftshift(xr));           % compute FFT for residual analysis
  Xh = genspecsines(sloc,smag,sphase,Ns);  % generate sines
  Xr = Xr-Xh;                               % get the residual complex spectrum
  mXr = 20*log10(abs(Xr(1:Ns/2+1)));        % magnitude spectrum of residual
  mXsenv = decimate(max(-200,mXr),stocf);  % decimate the magnitude spectrum
                                            % and avoid -Inf
```

```
  %-----synthesis data-----%
  ysloc = sloc;                              % synthesis locations
  ysmag = smag(1:nS);                        % synthesis magnitudes
  mYsenv = mXsenv;                           % synthesis residual envelope
  %-----transformations-----%
  %-----synthesis-----%
  if (fridx==0)
      lastysphase = ysphase;
  end
  if (nS>lastnS)
      lastysphase = [ lastysphase ; zeros(nS-lastnS,1) ];
      lastysloc = [ lastysloc ; zeros(nS-lastnS,1) ];
  end
  ysphase = lastysphase(lastidx(1:nS)) + 2*pi*( ...
            lastysloc(lastidx(1:nS))+ysloc)/2/Ns*H; % propagate phases
  lastysloc = ysloc;
  lastysphase = ysphase;
  lastnS = nS;                               % update last frame data
  lastsloc = sloc;                           % update last frame data
  lastsmag = smag;                           % update last frame data
  lastsphase = sphase;                       % update last frame data
  Yh = genspecsines(ysloc,ysmag,ysphase,Ns); % generate sines
  mYs = interp(mYsenv,stocf);                % interpolate to original size
  roffset = ceil(stocf/2)-1;                 % interpolated array offset
  mYs = [ mYs(1)*ones(roffset,1); mYs(1:Ns/2+1-roffset) ];
  mYs = 10.^(mYs/20);                        % dB to linear magnitude
  pYs = 2*pi*rand(Ns/2+1,1);      % generate phase spectrum with random values
  mYs1 = [mYs(1:Ns/2+1); mYs(Ns/2:-1:2)]; % create complete magnitude spectrum
  pYs1 = [pYs(1:Ns/2+1); -1*pYs(Ns/2:-1:2)]; % create complete phase spectrum
  Ys = mYs1.*cos(pYs1)+1i*mYs1.*sin(pYs1);   % compute complex spectrum
  yhw = fftshift(real(ifft(Yh)));     % sines in time domain using inverse FFT
  ysw = fftshift(real(ifft(Ys)));     % stochastic in time domain using IFFT
  yh(ri:ri+Ns-1) = yh(ri:ri+Ns-1)+yhw(1:Ns).*sw;  % overlap-add for sines
  ys(ri:ri+Ns-1) = ys(ri:ri+Ns-1)+ysw(1:Ns).*sws; % overlap-add for stochastic
  pin = pin+H;                                    % advance the sound pointer
  fridx = fridx+1;
end
y= yh+ys;                                         % sum sines and stochastic
```

Now we have an implementation of the sinusoidal plus residual model ready to be used for sound transformations. In order to use the different effects we just have to add the code of the effect after the line `%-----transformations-----%`.

Filtering

Filters are probably the paradigm of a classical effect. In our implementation of filtering we take advantage of the sinusoidal plus residual model, which allows us to modify the amplitude of any 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 (see Figure 10.19). In the example code given below, we define a band pass filter which supreses sinusoidal frequencies not included in the range [2100 3000].

M-file 10.15 (`spsfiltering.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%----- filtering -----%
Filter=[    0  2099 2100 3000 3001 fs/2;   % Hz
         -200  -200    0    0 -200 -200 ]; % db
ysmag = ysmag+interp1(Filter(1,:),Filter(2,:),ysloc/Ns*fs);
```

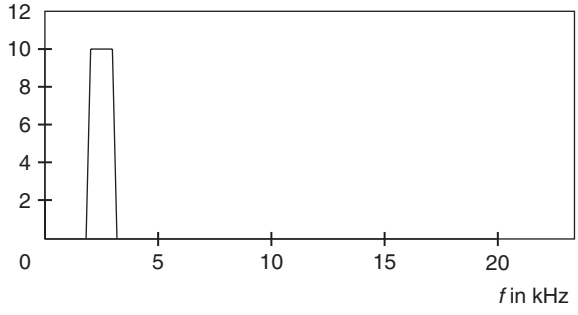


Figure 10.19 Bandpass filter with arbitrary resolution.

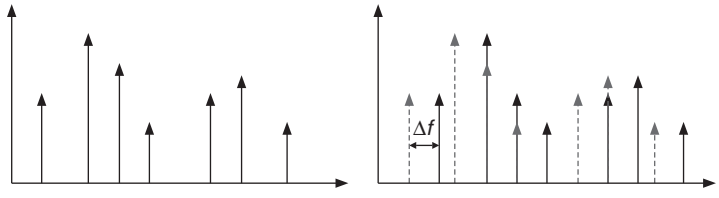


Figure 10.20 Frequency shift of the partials.

Frequency shifting

In this example we introduce a frequency shift factor to all the partials of our sound (see Figure 10.20). Note, though, that if the input sound is harmonic, then adding a constant to every partial produces a sound that will be inharmonic.

M-file 10.16 (spsfrequencyshifting.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----frequency shift-----%
fshift = 100;
ysloc = (ysloc>0).*(ysloc + fshift/fs*Ns); % frequency shift in Hz
```

Frequency stretching

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:

$$f_i = f_i \cdot f_{stretch}^{(i-1)}. \tag{10.17}$$

Figure 10.21 illustrates this frequency stretching.

M-file 10.17 (spsfrequencystretching.m)

```
%Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----frequency stretch-----%
fstretch = 1.1;
ysloc = ysloc .* (fstretch.^[0:length(ysloc)-1]');
```

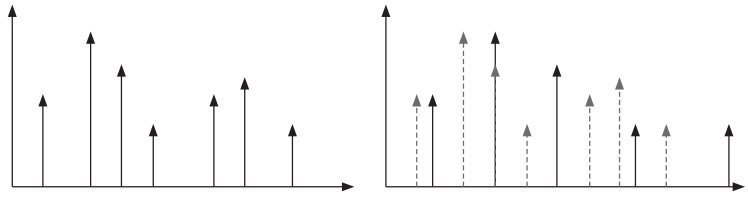


Figure 10.21 Frequency stretching.

Frequency scaling

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.

M-file 10.18 (spsfrequencyscaling.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----frequency scale-----%
fscale = 1.2;
ysloc = ysloc*fscale;
```

Time scaling

Time scaling is trickier and we have to modify some aspects of the main processing loop. We now use two different variables for the input and output times (`pin` and `pout`). The idea is that the analysis hop size will not be constant, but will depend on the time-scaling factor, while the synthesis hop size will be fixed. If the sound is played slower (time expansion), then the analysis hop size will be smaller and `pin` will be incremented by a smaller amount. Otherwise, when the sound is played faster (time compression), the hop size will be larger. Instead of using a constant time-scaling factor, here we opt for a more flexible approach by setting a mapping between input and output time. This allows, for instance, easy alignment of two audio signals by using synchronization onset times as the mapping function.

The following code sets the input/output time mapping. It has to be added at the beginning of the function. In this example, the sound is time expanded by a factor of 2.

M-file 10.19 (spstimemappingparams.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----time mapping-----%
timemapping = [ 0  1;                              % input time (sec)
                0  2 ];                            % output time (sec)
timemapping = timemapping *soundlength/fs;
outsoundlength = 1+round(timemapping(2,end)*fs); % length of output sound
```

Since the output and input durations may be different, we have to change the creation of the output signal variables so that they match the output duration.

M-file 10.20 (spstimemappinginit.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
yh = zeros(outsoundlength+Ns/2,1);                 % output sine component
ys = zeros(outsoundlength+Ns/2,1);                 % output residual component
```

Finally, we modify the processing loop. Note the changes in the last part, where the synthesis frame is added to the output vectors `yh` and `ys`. Note also that `pin` is computed from the synthesis time `pout` by interpolating the time-mapping function.

M-file 10.21 (`spstimemappingprocess.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
poutend = outsoundlength-hNs;             % last sample to start a frame
pout = pin;
minpin = 1+max(hNs,hM);
maxpin = min(length(x)-max(hNs,hM)-1);
fridx = 0;
while pout<poutend
  pin = round( interp1(TM(2,:),TM(1,:),pout/fs,'linear','extrap') * fs );
  pin = max(minpin,pin);
  pin = min(maxpin,pin);
  %-----analysis-----%
  (...)
  ro = pout-hNs;                            % output sound pointer for overlap
  yh(ro:ro+Ns-1) = yh(ro:ro+Ns-1)+yhw(1:Ns).*sw;  % overlap-add for sines
  ys(ro:ro+Ns-1) = ys(ro:ro+Ns-1)+ysw(1:Ns).*sws; % overlap-add for stochastic
  pout = pout+H;                                  % advance the sound pointer
  fridx = fridx+1;
end
y= yh+ys;                                         % sum sines and stochastic
```

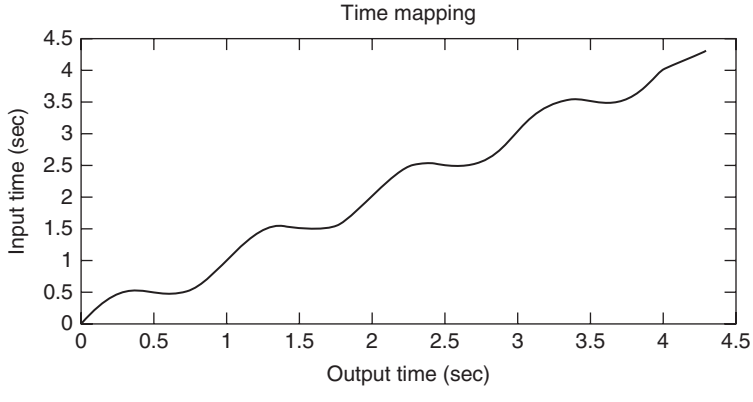


Figure 10.22 Complex time mapping.

With the proposed time-mapping control we can play the input sound in forward or reverse directions at any speed, or even freeze it. The following mapping uses a sinusoidal function to combine all these possibilities (see Figure 10.22). When applied to the *fairytale.wav* audio example, it produces a really funny result.

M-file 10.22 (`spstimemappingcomplex.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
tm = 0.01:0.01:.93;
timemapping = [  0 tm+0.05*sin(8.6*pi*tm) 1 ;  % input time --> keep end value
                 0 tm                     1 ]; % output time
timemapping = timemapping*length(x)/fs;
```

10.4.2 Harmonic plus residual

We will now use the harmonic plus residual model and also modify the implementation above. We use different variables for analysis and synthesis parameters and we keep a history of the last frame values. As before, phase propagation is implemented assuming linear frequency evolution of harmonics. Peak continuation, however, works differently: we just connect the closest peaks to each predicted harmonic frequency. In addition, we add the modifications required to able to perform time scaling. When we apply effects to harmonic sounds, sometimes we perceive the output harmonic and residual components to be somehow unrelated, in a way that loses part of the cohesion found in the original sound. Applying a comb filter to the residual determined by the synthesis fundamental frequency helps to merge both components and perceive them as a single entity. We have added such filter to the code below. Note also that the Yin algorithm is used by default when computing the fundamental frequency.

On the other hand, we can go a step further by implementing several transformations in the same code, and adding the corresponding control parameters as function arguments. This has the advantage that we can easily combine transformations to achieve more complex effects. Moreover, as we will see, this is great for generating several outputs with different effects to be concatenated or played simultaneously.

M-file 10.23 (hpsmodelparams.m)

```
function [y,yh,ys] = hpsmodelparams(x,fs,w,N,t,nH,minf0,maxf0,f0et,...
                                    maxhd,stocf,timemapping)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Analysis/synthesis of a sound using the sinusoidal harmonic model
% x: input sound, fs: sampling rate, w: analysis window (odd size),
% N: FFT size (minimum 512), t: threshold in negative dB,
% nH: maximum number of harmonics, minf0: minimum f0 frequency in Hz,
% maxf0: maximim f0 frequency in Hz,
% f0et: error threshold in the f0 detection (ex: 5),
% maxhd: max. relative deviation in harmonic detection (ex: .2)
% stocf: decimation factor of mag spectrum for stochastic analysis
% timemapping: mapping between input and output time (sec)
% y: output sound, yh: harmonic component, ys: stochastic component
if length(timemapping)==0                % argument not specified
    timemapping =[   0  length(x)/fs;    % input time
                     0  length(x)/fs ];  % output time
end
M = length(w);                           % analysis window size
Ns = 1024;                               % FFT size for synthesis
H = 256;                                 % hop size for analysis and synthesis
N2 = N/2+1;                              % half-size of spectrum
hNs = Ns/2;                              % half synthesis window size
hM = (M-1)/2;                            % half analysis window size
fftbuffer = zeros(N,1);                  % initialize buffer for FFT
outsoundlength = 1+round(timemapping(2,end)*fs);     % length of output sound
yh = zeros(outsoundlength+Ns/2,1);       % output sine component
ys = zeros(outsoundlength+Ns/2,1);       % output residual component
w = w/sum(w);                            % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                      % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;               % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);                 % synthesis window
bh = bh ./ sum(bh);                      % normalize synthesis window
wr = bh;                                 % window for residual
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
sws = H*hanning(Ns)/2;                % synthesis window for stochastic
lastyhloc = zeros(nH,1);              % initialize synthesis harmonic locations
```

```
yhphase = 2*pi*rand(nH,1);           % initialize synthesis harmonic phases
poutend = outsoundlength-max(hM,H);  % last sample to start a frame
pout = 1+max(hNs,hM);  % initialize sound pointer to middle of analysis window
minpin = 1+max(hNs,hM);
maxpin = min(length(x)-max(hNs,hM)-1);
while pout<poutend
  pin = round( interp1(timemapping(2,:),timemapping(1,:),pout/fs,'linear',...
                       'extrap') * fs );
  pin = max(minpin,pin);
  pin = min(maxpin,pin);
  %-----analysis-----%
  xw = x(pin-hM:pin+hM).*w(1:M);         % window the input sound
  fftbuffer(:) = 0;                      % reset buffer
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);  % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                    % compute the FFT
  mX = 20*log10(abs(X(1:N2)));           % magnitude spectrum
  pX = unwrap(angle(X(1:N/2+1)));        % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                 .* (mX(2:N2-1)>mX(1:N2-2)));     % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);     % refine peak values
  yinws = round(fs*0.0125);          % using approx. a 12.5 ms window for yin
  yinws = yinws+mod(yinws,2);        % make it even
  yb = pin-yinws/2;
  ye = pin+yinws/2+yinws;
  if (yb<1 || ye>length(x))          % out of boundaries
     f0 = 0;
  else
     f0 = f0detectionyin(x(yb:ye),fs,yinws,minf0,maxf0); % compute f0
  end
  hloc = zeros(nH,1);                    % initialize harmonic locations
  hmag = zeros(nH,1)-100;                % initialize harmonic magnitudes
  hphase = zeros(nH,1);                  % initialize harmonic phases
  hf = (f0>0).*(f0.*(1:nH));             % initialize harmonic frequencies
  hi = 1;                                % initialize harmonic index
  npeaks = length(ploc);                 % number of peaks found
  while (f0>0 && hi<=nH && hf(hi)<fs/2)  % find harmonic peaks
    [dev,pei] = min(abs((ploc(1:npeaks)-1)/N*fs-hf(hi)));   % closest peak
    if ((hi==1 || ~any(hloc(1:hi-1)==ploc(pei))) && dev<maxhd*hf(hi))
      hloc(hi) = ploc(pei);              % harmonic locations
      hmag(hi) = pmag(pei);              % harmonic magnitudes
      hphase(hi) = pphase(pei);          % harmonic phases
    end
    hi = hi+1;                           % increase harmonic index
  end
  hloc(1:hi-1) = (hloc(1:hi-1)~=0).*((hloc(1:hi-1)-1)*Ns/N);   % synth. locs
  ri= pin-hNs;                       % input sound pointer for residual analysis
  xr = x(ri:ri+Ns-1).*wr(1:Ns);          % window the input sound
  Xr = fft(fftshift(xr));                % compute FFT for residual analysis
  Xh = genspecsines(hloc(1:hi-1),hmag,hphase,Ns);           % generate sines
  Xr = Xr-Xh;                            % get the residual complex spectrum
  mXr = 20*log10(abs(Xr(1:Ns/2+1)));     % magnitude spectrum of residual
  mXsenv = decimate(max(-200,mXr),stocf); % decimate the magnitude spectrum
                                         % and avoid -Inf
  %-----synthesis data-----%
  yhloc = hloc;                          % synthesis harmonics locs
  yhmag = hmag;                          % synthesis harmonic amplitudes
  mYsenv = mXsenv;                       % synthesis residual envelope
  yf0 = f0;                              % synthesis f0
  %-----transformations-----%
  %-----synthesis-----%
  yhphase = yhphase + 2*pi*(lastyhloc+yhloc)/2/Ns*H;   % propagate phases
  lastyhloc = yhloc;
```

```
  Yh = genspecsines(yhloc,yhmag,yhphase,Ns);    % generate sines
  mYs = interp(mYsenv,stocf);                   % interpolate to original size
  roffset = ceil(stocf/2)-1;                    % interpolated array offset
  mYs = [ mYs(1)*ones(roffset,1); mYs(1:Ns/2+1-roffset) ];
  mYs = 10.^(mYs/20);                           % dB to linear magnitude
  if (f0>0)
      mYs = mYs .* cos(pi*[0:Ns/2]'/Ns*fs/yf0).^2;  % filter residual
  end
  fc = 1+round(500/fs*Ns);                          % 500 Hz
  mYs(1:fc) = mYs(1:fc) .* ([0:fc-1]'/(fc-1)).^2;   % HPF
  pYs = 2*pi*rand(Ns/2+1,1);      % generate phase spectrum with random values
  mYs1 = [mYs(1:Ns/2+1); mYs(Ns/2:-1:2)]; % create complete magnitude spectrum
  pYs1 = [pYs(1:Ns/2+1); -1*pYs(Ns/2:-1:2)];  % create complete phase spectrum
  Ys = mYs1.*cos(pYs1)+1i*mYs1.*sin(pYs1);    % compute complex spectrum
  yhw = fftshift(real(ifft(Yh)));       % sines in time domain using IFFT
  ysw = fftshift(real(ifft(Ys)));       % stochastic in time domain using IFFT
  ro= pout-hNs;                             % output sound pointer for overlap
  yh(ro:ro+Ns-1) = yh(ro:ro+Ns-1)+yhw(1:Ns).*sw;  % overlap-add for sines
  ys(ro:ro+Ns-1) = ys(ro:ro+Ns-1)+ysw(1:Ns).*sws; % overlap-add for stochastic
  pout = pout+H;                                % advance the sound pointer
end
y= yh+ys;                                       % sum sines and stochastic
```

In the previous code the phase of each partial is propagated independently using the estimated frequencies. This has two disadvantages: (a) inaccuracies in frequency estimation add errors to the phase propagation that are accumulated over time; (b) the phase information of the input signal is not used, so its phase behavior will not be transferred to the output signal. In many cases, this results not only in a loss of presence and definition, often referred to as *phasiness*, but also makes the synthetic signal to sound artificial, especially at low pitches and transitions. This issue is very much related to the concept of shape invariance, which we introduce next.

Although we focus the following discussion on the human voice, it can be generalized to many musical instruments and sound-production systems. In a simplified model of voice production, a train of impulses at the pitch rate excites a resonant filter (i.e., the vocal tract). According to this model, a speaker or singer changes the pitch of his voice by modifying the rate at which those impulses occur. An interesting observation is that the shape of the time-domain waveform signal around the impulse onsets is roughly independent of the pitch, but it depends mostly on the impulse response of the vocal tract. This characteristic is called *shape invariance*. In terms of frequency domain, this shape is related to the amplitude, frequency and phase values of the harmonics at the impulse onset times. A given processing technique will be shape invariant if it preserves the phase coherence between the various harmonics at estimated pulse onsets. Figure 10.23 compares the results of a time-scaling transformation using shape-variant and -invariant methods.

Several algorithms have been proposed in the literature regarding the harmonic phase-coherence for both phase-vocoder and sinusoidal modeling (for example [DiF98, Lar03]). Most of them are based on the idea of defining pitch-synchronous input and output onset times and reproducing at the output onset times the phase relationship existing in the original signal at the input onset times. Nevertheless, in order to obtain the best sound quality, it is desirable that for speech signals those detected onsets match the actual glottal pulse onsets [Bon08].

The following code uses this idea to implement a shape-invariant transformation. It should be inserted in the previous code after the line where synthesis phases are propagated, just at the beginning of the `%-----synthesis-----%` section.

M-file 10.24 (`shapeinvariance.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% shape invariance:
% assume pulse onsets to match zero phase of the fundamental
```

```
% and ideal harmonic distribution
pos = mod(hphase(1),2*pi)/2/pi;          % input normalized period position
ypos = mod(yhphase(1),2*pi)/2/pi;        % output normalized period position
yhphase = hphase + (ypos-pos)*2*pi*[1:length(yhloc)]';
```

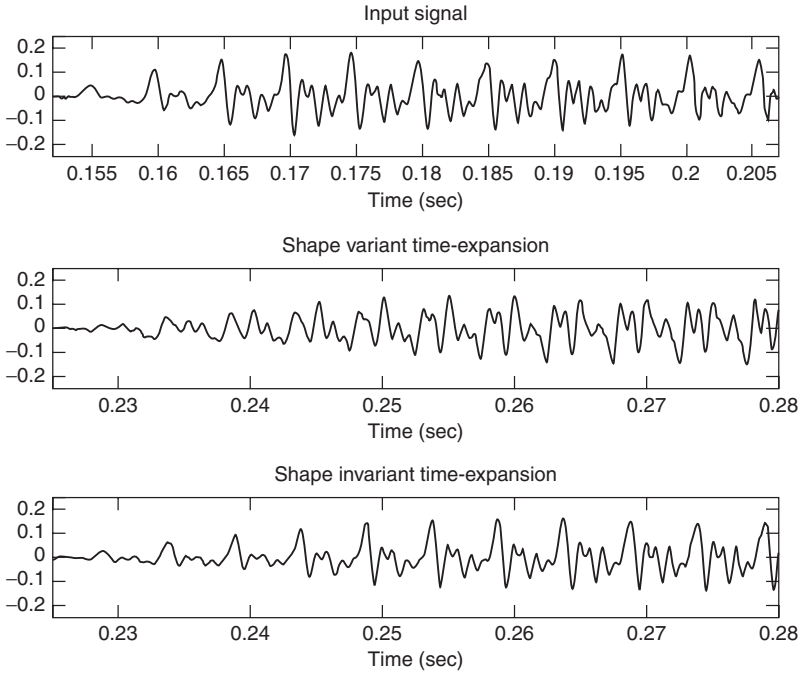


Figure 10.23 Shape-variant and -invariant transformations. In this example the input signal is time-expanded by a factor of 1.5.

Harmonic filtering

As shown, a filter does not need to be characterized by a traditional transfer function, but a more complex filter can be defined. 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.

M-file 10.25 (hpsmodelharmfiltering.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----voice to clarinet-----%
yhloc(2:2:end)=0; % set to zero the frequency of even harmonics
                  % so that they won't be synthesized
```

Pitch discretization

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 quantified to one of the 12 semitones into 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 the vocoder effect.

M-file 10.26 (hpsmodelpitchdisc.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----pitch discretization to temperate scale-----%
if (f0>0) % it has harmonics
  nst = round(12*log2(f0/55));     % closest semitone
  discpitch = 55*2^(nst/12);       % discretized pitch
  fscale = discpitch/f0 ;          % pitch transposition factor
  yf0 = f0*fscale;                 % synthesis f0
  yhloc = yhloc*fscale;
end
```

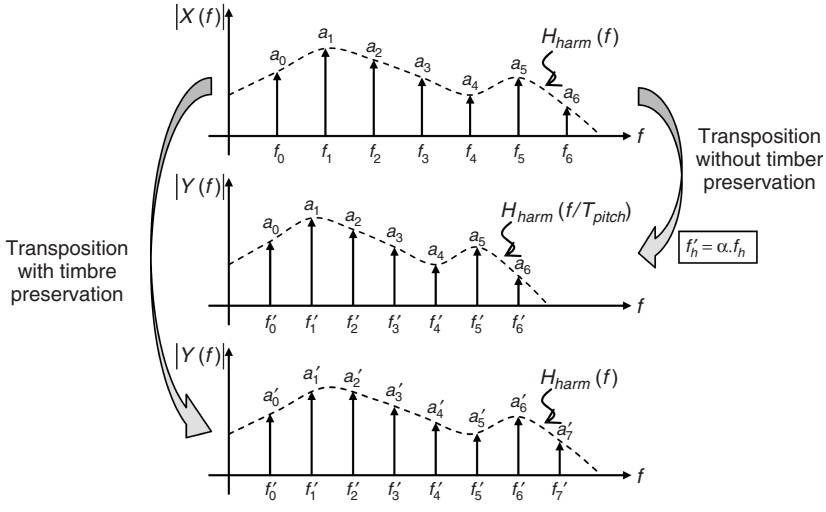


Figure 10.24 Pitch transposition. The signal whose transform is represented in the top view is transposed to a lower pitch. The middle view shows the result when only harmonic frequencies are modified, whereas in the bottom view representation both harmonic frequencies and amplitudes have been modified so that the timbre is preserved.

Pitch transposition with timbre preservation

Pitch transposition is the scaling of all the partials of a sound by the same multiplying factor. An undesirable effect is obtained if the amplitude of partials is not modified. In that case, the timbre is scaled as well, producing the typical *Mickey Mouse* effect. This undesired effect can be avoided using the analysis harmonic spectral envelope, which is obtained from the interpolation of the harmonic amplitude values. Then, when synthesizing, we compute harmonic amplitudes interpolating this same envelope at their scaled frequency position. This is illustrated in Figure 10.24, where α is the frequency scaling factor, and H_{harm} the harmonic spectral envelope.

M-file 10.27 (transpositionpreservingtimbre.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----pitch transposition and timbre scaling-----%
fscale = 0.9;
yhloc = yhloc*fscale;                % scale harmonic frequencies
yf0 = f0*fscale;                     % synthesis fundamental frequency
```

```
% harmonics
if (f0>0)
  thloc = interp1( timbremapping(2,:), timbremapping(1,:), ...
                   yhloc/Ns*fs) / fs*Ns; % mapped harmonic freqs.
  idx = find(hloc>0 & hloc<Ns*.5);       % harmonic indexes in frequency range
  yhmag = interp1([0; hloc(idx); Ns],[ hmag(1); hmag(idx); hmag(end) ],thloc);
                                         % interpolated envelope
end
% residual
  % frequency (Hz) of the last  coefficient
  frescoef = fs/2*length(mYsenv)*stocf/length(mXr);
  % mapped coef. indexes
  trescoef = interp1( timbremapping(2,:), timbremapping(1,:), ...
             min(fs/2,[0:length(mYsenv)-1]'/(length(mYsenv)-1)*frescoef) );
  % interpolated envelope
  mYsenv = interp1([0:length(mYsenv)-1],mYsenv, ...
                 trescoef/frescoef*(length(mYsenv)-1));
```

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. The control parameters are expressed in perceptually meaning units so to ease an intuitive manipulation.

M-file 10.28 (hpsmodelvibandtrem.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----vibrato and tremolo-----%
if (f0>0)
  vtf = 5;                                % vibrato-tremolo frequency in Hz
  va  = 50;                               % vibrato depth in cents
  td  = 3;                                % tremolo depth in dB
  sfscale = fscale*2^(va/1200*sin(2*pi*vtf*pin/fs));
  yhloc = yhloc*sfscale;                  % synthesis harmonic locs
  yf0 = f0*sfscale;                       % synthesis f0
  idx = find(hloc>0 & hloc<Ns*.5);
  yhmag = interp1([0; hloc(idx); Ns],[ hmag(1); hmag(idx); ...
                  hmag(end) ],yhloc);     % interpolated envelope
  yhmag = yhmag + td*sin(2*pi*vtf*pin/fs);  % tremolo
end
```

Timbre scaling

This is a very common, but still exciting transformation. The character of a voice changes significantly when timbre is scaled. Compressing the timbre produces deeper voices, whereas expanding it generates thinner and more female or childish-sounding utterances. Also, with the appropriate parameters, we can convincingly sound like a different person. In the following implementation, we define the timbre scaling with a mapping function between input and output frequencies. This is a very flexible and intuitive way of controlling the effect, and it is found in several voice transformation plug-ins that work in real time. In the following example, the low-frequency band of the spectrum (up to 5Khz) is compressed, so that deeper voices are generated. Note that the same timbre frequency mapping is applied to harmonic and residual components, although it would be straightforward to use separate controls.

M-file 10.29 (hpsmodeltimbrescaling.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----timbre scaling-----%
timbremapping = [ 0  5000  fs/2;   % input frequency
                  0  4000  fs/2 ]; % output frequency
% harmonics
if (f0>0)
  thloc = interp1( timbremapping(2,:), timbremapping(1,:), ...
                   yhloc/Ns*fs) / fs*Ns; % mapped harmonic freqs.
  idx = find(hloc>0 & hloc<Ns*.5); % harmonic indexes in frequency range
  yhmag = interp1([0; hloc(idx); Ns],[ hmag(1); hmag(idx); hmag(end) ],thloc);
  % interpolated envelope
end
% residual
  % frequency (Hz) of the last  coefficient
  frescoef = fs/2*length(mYsenv)*stocf/length(mXr);
  % mapped coef. indexes
  trescoef = interp1( timbremapping(2,:), timbremapping(1,:), ...
             min(fs/2,[0:length(mYsenv)-1]'/(length(mYsenv)-1)*frescoef) );
  % interpolated envelope
  mYsenv = interp1([0:length(mYsenv)-1],mYsenv, ...
                   trescoef/frescoef*(length(mYsenv)-1));
```

Roughness

Although roughness is sometimes thought of as a symptom of some kind of vocal disorder [Chi94], this effect has been widely used by singers in order to resemble the voices 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.

M-file 10.30 (hpsmodelroughness.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----roughness-----%
if f0>0
        mYsenv = mXsenv + 12; % gain factor applied to the residual (dB)
end
```

10.4.3 Combined effects

By combining *basic* effects like the ones presented, we are able to create more complex effects. Especially 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... Next we present some examples of those transformations.

Gender change

Using some of the previous effects we can change the gender of a vocal sound. In the following example we want to convert a male voice into a female one. Female fundamental frequencies are typically higher than male ones, so we first apply a pitch transposition upwards. Although we could apply any transposition factors, one octave is a common choice, especially for singing, since then background music does not need to be transposed as well. A second required transformation is timbre scaling, more concretely timbre expansion. The reason is that females typically have shorter vocal tracts than males, which results in higher formant (i.e., resonance) frequencies.

It is also interesting to consider a shift in the spectral shape, especially if we want to generate convincing female singing of certain specific musical styles such as opera. The theoretical explanation is that trained female opera singers move the formants along with the fundamental, especially in the high pitch range, to avoid the fundamental being higher than the first formant frequency, which would result in a loss of intelligibility and effort efficiency. Although not implemented in the following code, this feature can be simply added by subtracting the frequency shift value in Hz to the variable `thloc` and limiting resulting values to the sound frequency range.

M-file 10.31 (`hpsmodeltranspandtimbrescaling.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----pitch transposition and timbre scaling-----%
yhloc = yhloc*fscale;                % scale harmonic frequencies
yf0 = f0*fscale;                     % synthesis fundamental frequency
% harmonics
if (f0>0)
  thloc = interp1( timbremapping(2,:), timbremapping(1,:), ...
                   yhloc/Ns*fs) / fs*Ns; % mapped harmonic freqs.
  idx = find(hloc>0 & hloc<Ns*.5);       % harmonic indexes in frequency range
  yhmag = interp1([0; hloc(idx); Ns],[hmag(1); hmag(idx); hmag(end)],thloc);
                                         % interpolated envelope
end
% residual
  % frequency (Hz) of the last  coefficient
  frescoef = fs/2*length(mYsenv)*stocf/length(mXr);
  % mapped coef. indexes
  trescoef = interp1( timbremapping(2,:), timbremapping(1,:), ...
             min(fs/2,[0:length(mYsenv)-1]'/(length(mYsenv)-1)*frescoef) );
  % interpolated envelope
  mYsenv = interp1([0:length(mYsenv)-1],mYsenv, ...
                 trescoef/frescoef*(length(mYsenv)-1));
```

Next, we have to add the effect controls `fscale` and `timbremapping` to M-file 10.23 as function arguments, as shown below.

M-file 10.32 (`hpsmodelparamscall.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
function [y,yh,ys] =
  hpsmodelparams(x,fs,w,N,t,nH,minf0,maxf0,f0et,maxhd,stocf,timemapping,...
                 fscale,timbremapping)
(...)
```

And finally we can call the function with appropriate parameters for generating the male to female gender conversion.

M-file 10.33 (`maletofemale.m`)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----male to female-----%
[x,fs] = wavread('basket.wav');
w=[blackmanharris(1024);0];
fscale = 2;
timbremapping = [ 0  4000  fs/2;   % input frequency
                  0  5000  fs/2 ]; % output frequency
[y,yh,yr] = hpsmodelparams(x,fs,w,2048,-150,200,100,400,1,.2,10,...
                           [],fscale,timbremapping);
```

Now it is straightforward to apply other gender conversions, just modifying the parameters and calling the function again.

M-file 10.34 (maletochild.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----male to child-----%
[x,fs] = wavread('basket.wav');
w=[blackmanharris(1024);0];
fscale = 2;
timbremapping = [ 0  3600  fs/2;   % input frequency
                  0  5000  fs/2 ]; % output frequency
[y,yh,yr] = hpsmodelparams(x,fs,w,2048,-150,200,100,400,1,.2,10,...
                           [],fscale,timbremapping);
```

M-file 10.35 (femaletomale.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----female to male-----%
[x,fs] = wavread('meeting.wav');
w=[blackmanharris(1024);0];
fscale = 0.5;
timbremapping = [ 0  5000  fs/2;   % input frequency
                  0  4000  fs/2 ]; % output frequency
[y,yh,yr] = hpsmodelparams(x,fs,w,2048,-150,200,100,400,1,.2,10,...
                           [],fscale,timbremapping);
```

Adding even more transformations we can achieve more sophisticated gender changes. For instance, an exaggerated vibrato applied to speech helps to emulate the fundamental instability typical of an old person. Note that vibrato parameters and the corresponding transformation have to be added to the hpsmodelparams function.

M-file 10.36 (genderchangetoold.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----change to old-----%
[x,fs] = wavread('fairytale.wav');
w=[blackmanharris(1024);0];
fscale = 2;
timbremapping = [ 0  fs/2;   % input frequency
                  0  fs/2 ]; % output frequency
vtf = 6.5;                   % vibrato-tremolo frequency in Hz
va  = 400;                   % vibrato depth in cents
td  = 3;                     % tremolo depth in dB
[y,yh,yr] = hpsmodelparams(x,fs,w,4096,-150,200,50,600,2,.3,1,...
                           [],fscale,timbremapping,vtf,va,td);
```

Harmonizer

In order to create the effect of harmonizing vocals, we can add pitch-shifted versions of the original voice (with the same or different timbres) and force them to be in tune with the accompanying melodies. In this example we generate two voices: a more female-sounding singer a major third upwards, and a more male-sounding vocalization a perfect fourth downwards. Then we add all of them together. The method is very simple and actually generates voices that behave too similarly. More natural results could be obtained by adding random variations in timing and tuning to the different voices.

M-file 10.37 (harmonizer.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
[x,fs]=wavread('tedeum2.wav');
w=[blackmanharris(1024);0];
% female voice
fscale = 2^(4/12); % 4 semitones upwards
timbremapping = [ 0  4000  fs/2;   % input frequency
                  0  5000  fs/2 ]; % output frequency
[y,yh,yr] = hpsmodelparams(x,fs,w,2048,-150,200,100,400,1,.2,10,...
                           [],fscale,timbremapping);
% male voice
fscale = 2^(-5/12); % 5 semitones downwards
timbremapping = [ 0  5000  fs/2;   % input frequency
                  0  4000  fs/2 ]; % output frequency
[y2,yh2,yr2] = hpsmodelparams(x,fs,w,2048,-150,200,100,400,1,.2,10,...
                             [],fscale,timbremapping);
% add voices
l = min([length(x) length(y) length(y2) ]);
ysum = x(1:l)+y(1:l)+y2(1:l);
```

Choir

We can simulate a choir from a single vocalization by generating many clones of the input voice. It is important that each clone has slightly different characteristics, such as different timing, timbre and tuning. We can use the same function `hpsmodelparams` used before. However, as it is, pitch transposition is a constant, so we would get too similar fundamental behaviors. This can be easily improved by allowing the `fscale` parameter to be a vector with random transposition factors. Using the following code to compute the transposition parameter before performing pitch transposition and timbre scaling solves the problem. It has to be inserted in the `%-----transformations-----%` section of M-file 10.23, and the transformation parameters have to be added to the function header, as in M-file 10.32.

M-file 10.38 (hpsmodeltranspositionenv.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
%-----pitch transposition and timbre scaling-----%
if (size(fscale,1)>1) % matrix
    cfscale = interp1(fscale(1,:),fscale(2,:),pin/fs,'spline','extrap');
    yhloc = yhloc*cfscale;                % synthesis harmonic locs
    yf0 = f0*cfscale;                     % synthesis fundamental frequency
else
    yhloc = yhloc*fscale;                 % scale harmonic frequencies
    yf0 = f0*fscale;                      % synthesis fundamental frequency
end
% harmonics
if (f0>0)
  thloc = interp1( timbremapping(2,:), timbremapping(1,:), ...
                   yhloc/Ns*fs) / fs*Ns; % mapped harmonic freqs.
  idx = find(hloc>0 & hloc<Ns*.5);       % harmonic indexes in frequency range
  yhmag = interp1([0; hloc(idx); Ns],[ hmag(1); hmag(idx); hmag(end) ],thloc);
                                        % interpolated envelope
end
% residual
  % frequency (Hz) of the last  coefficient
  frescoef = fs/2*length(mYsenv)*stocf/length(mXr);
  % mapped coef. indexes
```

```
  trescoef = interp1( timbremapping(2,:), timbremapping(1,:), ...
             min(fs/2,[0:length(mYsenv)-1]'/(length(mYsenv)-1)*frescoef) );
  % interpolated envelope
  mYsenv = interp1([0:length(mYsenv)-1],mYsenv, ...
                 trescoef/frescoef*(length(mYsenv)-1));
```

Now we have all the ingredients ready for generating the choir. In the following example we generate up to 15 voices, each one with random variations, which are added to the original one afterwards. For improving the transformation, each clone is added to a random panning position within a stereo output.

M-file 10.39 (choir.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
[x,fs]=wavread('tedeum.wav');
w=[blackmanharris(2048);0];
dur = length(x)/fs;
fn = ceil(dur/0.2);
fscale = zeros(2,fn);
fscale(1,:) = [0:fn-1]/(fn-1)*dur;
tn = ceil(dur/0.5);
timemapping = zeros(2,tn);
timemapping(1,:) = [0:tn-1]/(tn-1)*dur;
timemapping(2,:) = timemapping(1,:);
ysum = [ x x ]; % make it stereo
for i=1:15 % generate 15 voices
  disp([ ' processing ',num2str(i) ]);
  fscale(2,:) = 2.^(randn(1,fn)*30/1200);
  timemapping(2,2:end-1) = timemapping(1,2:end-1) + ...
                           randn(1,tn-2)*length(x)/fs/tn/6;
  timbremapping = [ 0 1000:1000:fs/2-1000  fs/2;
   0 (1000:1000:fs/2-1000).*(1+.1*randn(1,length(1000:1000:fs/2-1000))) fs/2];
  [y,yh,yr] = hpsmodelparams(x,fs,w,2048,-150,200,100,400,1,.2,10,...
                             timemapping,fscale,timbremapping);
  pan = max(-1,min(1,randn(1)/3.)); % [0,1]
  l = cos(pan*pi/2);%.^2;
  r = sin(pan*pi/2);%1-l;
  ysum = ysum + [l*y(1:length(x)) r*y(1:length(x))];
end
```

Note that this code is not optimized, there are a lot of redundant computations. An obvious way to improve it would be to first analyze the whole input sound, and then to use the analysis data to synthesize each of the clone voices. Nevertheless, there is no limit in the number of clone voices.

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 Sections 8.3.1 and 8.3.3 respectively).

In the following **MATLAB** code we introduce a morphing algorithm based on the interpolation of the harmonic and residual components of two sounds. In our implementation, we provide three interpolation factors that independently control the fundamental frequency, the harmonic timbre and the residual envelope. Playing with those control parameters we can smoothly go from one sound to the other, or combine characteristics of both.

M-file 10.40 (hpsmodelmorph.m)

```
function [y,yh,ys] = hpsmodelmorph(x,x2,fs,w,N,t,nH,minf0,maxf0,f0et,...
                                   maxhd,stocf,f0intp,htintp,rintp)
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
% Morph between two sounds using the harmonic plus stochastic model
% x,x2: input sounds, fs: sampling rate, w: analysis window (odd size),
% N: FFT size (minimum 512), t: threshold in negative dB,
% nH: maximum number of harmonics, minf0: minimum f0 frequency in Hz,
% maxf0: maximim f0 frequency in Hz,
% f0et: error threshold in the f0 detection (ex: 5),
% maxhd: max. relative deviation in harmonic detection (ex: .2)
% stocf: decimation factor of mag spectrum for stochastic analysis,
% f0intp: f0 interpolation factor,
% htintp: harmonic timbre interpolation factor,
% rintp: residual interpolation factor,
% y: output sound, yh: harmonic component, ys: stochastic component
if length(f0intp)==1
    f0intp =[ 0      length(x)/fs;    % input time
              f0intp f0intp       ];  % control value
end
if length(htintp)==1
    htintp =[ 0      length(x)/fs;    % input time
              htintp htintp       ];  % control value
end
if length(rintp)==1
    rintp =[ 0     length(x)/fs;      % input time
             rintp rintp         ];   % control value
end
M = length(w);                        % analysis window size
Ns = 1024;                            % FFT size for synthesis
H = 256;                              % hop size for analysis and synthesis
N2 = N/2+1;                           % half-size of spectrum
soundlength = length(x);              % length of input sound array
hNs = Ns/2;                           % half synthesis window size
hM = (M-1)/2;                         % half analysis window size
pin = 1+max(hNs,hM);   % initialize sound pointer to middle of analysis window
pend = soundlength-max(hM,hNs);         % last sample to start a frame
fftbuffer = zeros(N,1);               % initialize buffer for FFT
yh = zeros(soundlength+Ns/2,1);       % output sine component
ys = zeros(soundlength+Ns/2,1);       % output residual component
w = w/sum(w);                         % normalize analysis window
sw = zeros(Ns,1);
ow = triang(2*H-1);                   % overlapping window
ovidx = Ns/2+1-H+1:Ns/2+H;            % overlap indexes
sw(ovidx) = ow(1:2*H-1);
bh = blackmanharris(Ns);              % synthesis window
bh = bh ./ sum(bh);                   % normalize synthesis window
wr = bh;                              % window for residual
sw(ovidx) = sw(ovidx) ./ bh(ovidx);
sws = H*hanning(Ns)/2;                % synthesis window for stochastic
lastyhloc = zeros(nH,1);              % initialize synthesis harmonic locs.
yhphase = 2*pi*rand(nH,1);            % initialize synthesis harmonic phases
minpin2 = max(H+1,1+hM);              % minimum sample value for x2
maxpin2 = min(length(x2)-hM-1);       % maximum sample value for x2
```

```
while pin<pend
  %-----first sound analysis-----%
  [f0,hloc,hmag,mXsenv] = hpsanalysis(x,fs,w,wr,pin,M,hM,N,N2,Ns,hNs,...
                                    nH,t,f0et,minf0,maxf0,maxhd,stocf);
  %-----second sound analysis-----%
  pin2 = round(pin/length(x)*length(x2)); % linear time mapping between inputs
  pin2 = min(maxpin2,max(minpin2,pin2));
  [f02,hloc2,hmag2,mXsenv2] = hpsanalysis(x2,fs,w,wr,pin2,M,hM,N,N2,Ns,hNs,...
                                        nH,t,f0et,minf0,maxf0,maxhd,stocf);
  %-----morph-----%
  cf0intp = interp1(f0intp(1,:),f0intp(2,:),pin/fs);   % get control value
  chtintp = interp1(htintp(1,:),htintp(2,:),pin/fs);   % get control value
  crintp = interp1(rintp(1,:),rintp(2,:),pin/fs);      % get control value
  if (f0>0 && f02>0)
      outf0 = f0*(1-cf0intp) + f02*cf0intp;         % both inputs are harmonic
      yhloc = [1:nH]'*outf0/fs*Ns;         % generate synthesis harmonic serie
      idx = find(hloc>0 & hloc<Ns*.5);
      yhmag = interp1([0;hloc(idx);Ns], [hmag(1);hmag(idx);hmag(end)],yhloc);
                                                  % interpolated envelope
      idx2 = find(hloc2>0 & hloc2<Ns*.5);
      yhmag2 = interp1([0; hloc2(idx2); Ns],...
             [hmag2(1);hmag2(idx2);hmag2(end)],yhloc); % interpolated envelope
      yhmag = yhmag*(1-chtintp) + yhmag2*chtintp;      % timbre morphing
  else
      outf0 = 0;                                     % remove harmonic content
      yhloc = hloc.*0;
      yhmag = hmag.*0;
  end
  mYsenv = mXsenv*(1-crintp) + mXsenv2*crintp;
  %-----synthesis-----%
  yhphase = yhphase + 2*pi*(lastyhloc+yhloc)/2/Ns*H; % propagate phases
  lastyhloc = yhloc;
  Yh = genspecsines(yhloc,yhmag,yhphase,Ns);    % generate sines
  mYs = interp(mYsenv,stocf);                   % interpolate to original size
  roffset = ceil(stocf/2)-1;                    % interpolated array offset
  mYs = [ mYs(1)*ones(roffset,1); mYs(1:Ns/2+1-roffset) ];
  mYs = 10.^(mYs/20);                           % dB to linear magnitude
  pYs = 2*pi*rand(Ns/2+1,1);      % generate phase spectrum with random values
  mYs1 = [mYs(1:Ns/2+1); mYs(Ns/2:-1:2)]; % create complete magnitude spectrum
  pYs1 = [pYs(1:Ns/2+1); -1*pYs(Ns/2:-1:2)]; % create complete phase spectrum
  Ys = mYs1.*cos(pYs1)+1i*mYs1.*sin(pYs1);   % compute complex spectrum
  yhw = fftshift(real(ifft(Yh)));            % sines in time domain using IFFT
  ysw = fftshift(real(ifft(Ys)));       % stochastic in time domain using IFFT
  ro = pin-hNs;                         % output sound pointer for overlap
  yh(ro:ro+Ns-1) = yh(ro:ro+Ns-1)+yhw(1:Ns).*sw;  % overlap-add for sines
  ys(ro:ro+Ns-1) = ys(ro:ro+Ns-1)+ysw(1:Ns).*sws; % overlap-add for stochastic
  pin = pin+H;                                  % advance the sound pointer
end
y= yh+ys;                                       % sum sines and stochastic
end

function [f0,hloc,hmag,mXsenv] = hpsanalysis(x,fs,w,wr,pin,M,hM,N,N2,...
                                  Ns,hNs,nH,t,f0et,minf0,maxf0,maxhd,stocf)
  xw = x(pin-hM:pin+hM).*w(1:M);              % window the input sound
  fftbuffer = zeros(N,1);                     % initialize buffer for FFT
  fftbuffer(1:(M+1)/2) = xw((M+1)/2:M);       % zero-phase window in fftbuffer
  fftbuffer(N-(M-1)/2+1:N) = xw(1:(M-1)/2);
  X = fft(fftbuffer);                         % compute the FFT
  mX = 20*log10(abs(X(1:N2)));                % magnitude spectrum
  pX = unwrap(angle(X(1:N/2+1)));             % unwrapped phase spectrum
  ploc = 1 + find((mX(2:N2-1)>t) .* (mX(2:N2-1)>mX(3:N2)) ...
                 .* (mX(2:N2-1)>mX(1:N2-2)));    % find peaks
  [ploc,pmag,pphase] = peakinterp(mX,pX,ploc);    % refine peak values
```

```
  f0 = f0detectiontwm(mX,fs,ploc,pmag,f0et,minf0,maxf0);    % find f0
  hloc = zeros(nH,1);                          % initialize harmonic locations
  hmag = zeros(nH,1)-100;                      % initialize harmonic magnitudes
  hphase = zeros(nH,1);                        % initialize harmonic phases
  hf = (f0>0).*(f0.*(1:nH));                   % initialize harmonic frequencies
  hi = 1;                                      % initialize harmonic index
  npeaks = length(ploc);                       % number of peaks found
  while (f0>0 && hi<=nH && hf(hi)<fs/2)        % find harmonic peaks
    [dev,pei] = min(abs((ploc(1:npeaks)-1)/N*fs-hf(hi)));  % closest peak
    if ((hi==1 || ~any(hloc(1:hi-1)==ploc(pei))) && dev<maxhd*hf(hi))
      hloc(hi) = ploc(pei);                    % harmonic locations
      hmag(hi) = pmag(pei);                    % harmonic magnitudes
      hphase(hi) = pphase(pei);                % harmonic phases
    end
    hi = hi+1;                                 % increase harmonic index
  end
  hloc(1:hi-1) = (hloc(1:hi-1)~=0).*((hloc(1:hi-1)-1)*Ns/N); % synth. locs
  ri = pin-hNs;                     % input sound pointer for residual analysis
  xr = x(ri:ri+Ns-1).*wr(1:Ns);     % window the input sound
  Xr = fft(fftshift(xr));           % compute FFT for residual analysis
  Xh = genspecsines(hloc(1:hi-1),hmag,hphase,Ns);        % generate sines
  Xr = Xr-Xh;                                  % get the residual complex spectrum
  mXr = 20*log10(abs(Xr(1:Ns/2+1)));           % magnitude spectrum of residual
  mXsenv = decimate(max(-200,mXr),stocf);  % decimate the magnitude spectrum
end
```

Compared to the previous code M-file 10.23, we added a second analysis section followed by the morph section. For simplicity, we put the analysis in a separate function, and removed the time-scaling transformation. Besides, we constrained the morph of the deterministic component to frames where both input signals have pitch. Otherwise, the deterministic part is muted. Note also that if the sounds have different durations, the audio resulting from the morphing will have the duration of the input.

The next code performs a morph between a soprano and a violin, where the timbre of the violin is applied to the voice.

M-file 10.41 (morphvocalviolin.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
[x,fs]=wavread('soprano-E4.wav');
[x2,fs]=wavread('violin-B3.wav');
w=[blackmanharris(1024);0];
f0intp = 0;
htintp = 1;
rintp = 0;
[y,yh,ys] = hpsmodelmorph(x,x2,fs,w,2048,-150,200,100,400,1500,1.5,10,...
                          f0intp,htintp,rintp);
```

In this other example we use time-varying functions to control the morph so that we smoothly change the characteristics from the voice to the violin.

M-file 10.42 (morphvocalviolin2.m)

```
% Authors: J. Bonada, X. Serra, X. Amatriain, A. Loscos
[x,fs]=wavread('soprano-E4.wav');
[x2,fs]=wavread('violin-B3.wav');
```

```
w=[blackmanharris(1024);0];
dur = (length(x)-1)/fs;
f0intp = [ 0 dur; 0 1];
htintp = [ 0 dur; 0 1];
rintp = [ 0 dur; 0 1];
[y,yh,ys] = hpsmodelmorph(x,x2,fs,w,2048,-150,200,100,400,1500,1.5,10,...
                          f0intp,htintp,rintp);
```

10.5 Conclusions

In this chapter, we have shown how the use of spectral models based on a sinusoidal plus residual decomposition 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 use 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, such as the ones presented in this chapter. Under many circumstances, spectral models based on the sinusoidal plus residual decomposition offer much more flexibility and processing capabilities than more immediate representations of the sound signal.

In general, higher-level sound representations 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 a 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 in taking the right decision in order to get the desired results.

References

[Bon08] J. Bonada. Wide-band harmonic sinusoidal modeling. In *Proc. 11th Int. Conf. Digital Audio Effects (DAFx-08)*, 2008.

[Can98] P. Cano. Fundamental frequency estimation in the SMS analysis. In *Proc. DAFX-98 Digital Audio Effects Workshop*, pp. 99–102, 1998.

[Chi94] D. G. Childers. Measuring and modeling vocal source-tract interaction. *IEEE Trans. Biomed. Eng*., 41(7): 663–671, 1994.

[CK02] A. Cheveigné and H. Kawahara. YIN, A fundamental frequency estimator for speech and music. *J. Acoust. Soc. Am*., 111(4): 1917–1930, 2002.

[Cox71] M. G. Cox. An algorithm for approximating convex functions by means of first-degree splines. *Comp. J*., 14: 272–275, 1971.

[DGR93] Ph. Depalle, G. Garcia and X. Rodet. Analysis of sound for additive synthesis: tracking of partials using Hidden Markov models. In *Proc. Int. Comp. Music Conf*., pp. 94–97, 1993.

[DH97] Ph. Depalle and T. Hélie. Extraction of spectral peak parameters using a short-time Fourier transform modeling and no sidelobe windows. In *Proc. 1997 IEEE Workshop Appl. Signal Process. Audio Acoust*., pp. 298–231, 1997.

[DiF98] R. DiFederico. Waveform preserving time stretching and pitch shifting for sinusoidal models of sound. In *Proc. DAFX-98 Digital Audio Effects Workshop*, 1998.

[DQ97] Y. Ding and X. Qian. Sinusoidal and residual decomposition and residual modeling of musical tones using the QUASAR signal model. In *Proc. Int. Comp. Music Conf*., pp. 35–42, 1997.

[FHC00] K. Fitz, L. Haken, and P. Christensen. A new algorithm for bandwidth association in bandwidth-enhanced additive sound modeling. In *Proc. Int. Com. Music Conf*., pp. 384–387, 2000.

[Goo97] M. Goodwin. *Adaptative signal models: Theory, algorithms and audio applications*. PhD thesis, University of California, Berkeley, 1997.

[Hes83] W. Hess. *Pitch Determination of Speech Signals*. Springer-Verlag, 1983.

[Lar03] J. Laroche. Frequency-domain techniques for high-quality voice modification. In *Proc. 6th Int. Conf Digital Audio Effects (DAFx-03)*, 2003.

[Mak75] J. Makhoul. Linear prediction: a tutorial review. *Proce IEEE*, 63(4): 561–580, 1975.

[MB94] R. C. Maher and J. W. Beauchamp. Fundamental frequency estimation of musical signals using a two-way mismatch procedure. *J. Acoust. Soc. Am*., 95(4): 2254–2263, 1994.

[MB10] S. Musevic and J. Bonada. Comparison of non-stationary sinusoid estimation methods using reassignment and derivatives. In *Proc. Sound Music Computing Conf*. 2010.

[MG75] J. D. Markel and A. H. Gray. *Linear Prediction of Speech*. Springer-Verlag, 1975.

[MQ86] R. J. McAulay and T. F. Quatieri. Speech analysis/synthesis based on a sinusoidal representation. *IEEE Trans Acoustics Speech Signal Process*, 34(4): 744–754, 1986.

[Pee01] G. Peeters. *Modéles et modification du signal sonore adaptès á ses caractéristiques locales*. PhD thesis, spécialité Acoustique Traitement du signal et Informatique Appliqués à la Musique, Universite Paris 6, 2001.

[RD92] X. Rodet and Ph. Depalle. Spectral envelopes and inverse FFT synthesis. *Proc. 93rd AES Conve*. AES Preprint No. 3393 (H-3), 1992.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[Ser89] X. Serra. *A System for Sound Analysis/Transformation/Synthesis based on a Deterministic plus Stochastic Decomposition*. PhD thesis, Stanford University, 1989.

[Ser96] X. Serra. Musical sound modeling with sinusoids plus noise. In G. De Poli, A. Picialli, S. T. Pope, and C. Roads (eds), *Musical Signal Processing*, pp. 91–122, Swets & Zeitlinger Publishers, 1996.

[SS87] J. O. Smith and X. Serra. PARSHL: an analysis/synthesis program for non-harmonic sounds based on a sinusoidal representation. In *Proc. Int. Comp. Music Conf*., pp. 290–297, 1987.

[SS90] X. Serra and J. Smith. Spectral modeling synthesis: a sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Comp. Music J*., 14(4): 12–24, 1990.

[Str80] J. Strawn. Approximation and syntactic analysis of amplitude and frequency functions for digital sound synthesis. *Comp. Music J*., 4(3): 3–24, 1980.

[VM98] T. S. Verma and T. H. Y. Meng. Time scale modification using a sines+transients+noise signal model. In *Proc. DAFX-98 Digital Audio Effects Workshop*, pp. 49–52, 1998.

[VM00] T. S. Verma and T. H. Y. Meng. Extending spectral modeling synthesis with transient modeling synthesis. *Comp. Music J*., 24(2): 47–59, 2000.

[ZRR07] M. Zivanovic, A. Röbel and X. Rodet. Adaptive threshold determination for spectral peak classification. In *Proc. 10th Int. Conf. Digital Audio Effects (DAFx-07)*, 2007.