7

Time-frequency processing

D. Arfib, F. Keiler, U. Zölzer, V. Verfaille and J. Bonada

7.1 Introduction

This chapter describes the use of time-frequency representations of signals in order to produce transformations of sounds. A very interesting (and intuitive) way of modifying a sound is to make a two-dimensional representation of it, modify this representation in some way and reconstruct a new signal from this representation (see Figure 7.1). Consequently a digital audio effect based on time-frequency representations requires three steps: an analysis (sound to representation), a transformation (of the representation) and a resynthesis (getting back to a sound).

The direct scheme of spectral analysis, transformation and resynthesis will be discussed in Section 7.2. We will explore the modification of the magnitude $|X(k)|$ and phase $\varphi(k)$ of these representations before resynthesis. The analysis/synthesis scheme is termed the *phase vocoder* (see Figure 7.2). The input signal $x(n)$ is multiplied by a sliding window of finite length N, which yields successive windowed signal segments. These are transformed to the spectral domain by FFTs. In this way a time-varying spectrum $X(n,k) = |X(n,k)|\mathrm{e}^{j\varphi(n,k)}$ with $k = 0, 1, \ldots, N-1$ is computed for each windowed segment. The short-time spectra can be modified or transformed for a digital audio effect. Then each modified spectrum is applied to an IFFT and windowed in the time domain. The windowed output segments are overlapped and added yielding the output signal. It is also possible to complete this time-frequency processing by spectral processing, which is dealt with in the next two chapters.

7.2 Phase vocoder basics

The concepts of short-time Fourier analysis and synthesis have been widely described in the literature [Por76, Cro80, CR83]. We will briefly summarize the basics and define our notation of terms for application to digital audio effects.

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

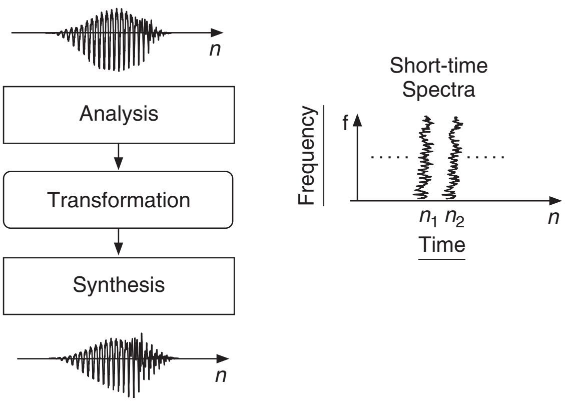


Figure 7.1 Digital audio effects based on analysis, transformation and synthesis (resynthesis).

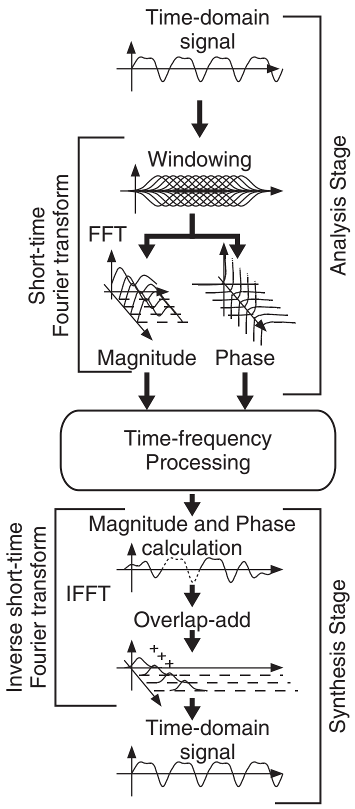


Figure 7.2 Time-frequency processing based on the phase vocoder: analysis, transformation and synthesis.

The short-time Fourier transform (STFT) of the signal $x(n)$ is given by

$$X(n,k) = \sum_{m=-\infty}^{\infty} x(m)h(n-m)W_N^{mk}, \quad k = 0, 1, \ldots, N-1 \tag{7.1}$$

$$W_N = \mathrm{e}^{-j2\pi/N} \tag{7.2}$$

$$= X_R(n,k) + jX_I(n,k) = |X(n,k)| \cdot \mathrm{e}^{j\varphi(n,k)}. \tag{7.3}$$

$X(n,k)$ is a complex number and represents the magnitude $|X(n,k)|$ and phase $\varphi(n,k)$ of a time-varying spectrum with frequency bin (index) $0 \le k \le N-1$ and time index n. Note that the summation index is m in (7.1). At each time index n the signal $x(m)$ is weighted by a finite length window $h(n-m)$. Thus the computation of (7.1) can be performed by a finite sum over m with an FFT of length N. Figure 7.3 shows the input signal $x(m)$ and the sliding window $h(n-m)$ for three time indices of n. The middle plot shows the finite length windowed segments $x(m) \cdot h(n-m)$. These segments are transformed by the FFT, yielding the short-time spectra $X(n,k)$ given by (7.1). The lower two rows in Figure 7.3 show the magnitude and phase spectra of the corresponding time segments.

7.2.1 Filter bank summation model

The computation of the time-varying spectrum of an input signal can also be interpreted as a parallel bank of N bandpass filters, as shown in Figure 7.4, with impulse responses and Fourier transforms given by

$$h_k(n) = h(n)W_N^{-nk}, \quad k = 0, 1, \ldots, N-1 \tag{7.4}$$

$$H_k(\mathrm{e}^{j\omega}) = H(\mathrm{e}^{j(\omega-\omega_k)}), \omega_k = \frac{2\pi}{N}k. \tag{7.5}$$

Each bandpass signal $y_k(n)$ is obtained by filtering the input signal $x(n)$ with the corresponding bandpass filter $h_k(n)$. Since the bandpass filters are complex-valued, we get complex-valued output signals $y_k(n)$, which will be denoted by

$$y_k(n) = \tilde{X}(n,k) = |X(n,k)| \cdot \mathrm{e}^{j\tilde{\varphi}(n,k)}. \tag{7.6}$$

These filtering operations are performed by the convolutions

$$y_k(n) = \sum_{m=-\infty}^{\infty} x(m)h_k(n-m) = \sum_{m=-\infty}^{\infty} x(m)h(n-m)W_N^{-(n-m)k} \tag{7.7}$$

$$= W_N^{-nk} \sum_{m=-\infty}^{\infty} x(m)W_N^{mk}h(n-m) = W_N^{-nk}X(n,k). \tag{7.8}$$

From (7.6) and (7.8) it is important to notice that

$$\tilde{X}(n,k) = W_N^{-nk}X(n,k) = W_N^{-nk}|X(n,k)|\mathrm{e}^{j\varphi(n,k)} \tag{7.9}$$

$$\tilde{\varphi}(n,k) = \frac{2\pi k}{N}n + \varphi(n,k). \tag{7.10}$$

Based on Equations (7.7) and (7.8) two different implementations are possible, as shown in Figure 7.4. The first implementation is the so-called complex baseband implementation according to (7.8). The baseband signals $X(n,k)$ (short-time Fourier transform) are computed by modulation of $x(n)$ with W_N^{nk} and lowpass filtering for each channel k. The modulation of $X(n,k)$ by W_N^{-nk} yields the bandpass signal $\tilde{X}(n,k)$. The second implementation is the so-called complex bandpass

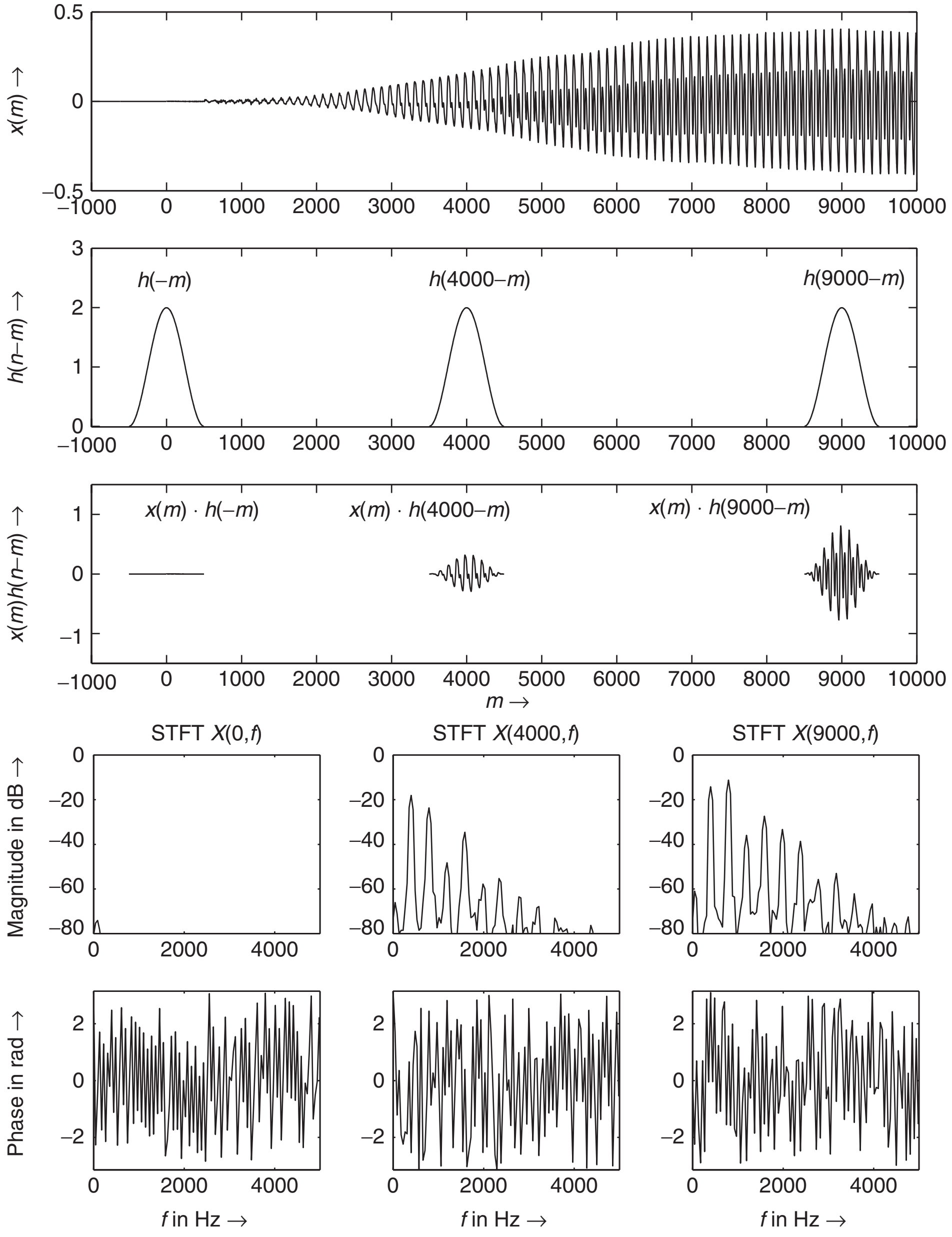


Figure 7.3 Sliding analysis window and short-time Fourier transform.

implementation, which filters the input signal with $h_k(n)$ given by (7.4), as shown in the lower left part of Figure 7.4. This implementation leads directly to the complex-valued bandpass signals $\tilde{X}(n,k)$. If the equivalent baseband signals $X(n,k)$ are necessary, they can be computed by multiplication with W_N^{nk}. The operations for the modulation by W_N^{nk} yielding $X(n,k)$ and back modulation by W_N^{-nk} (lower left part of Figure 7.4) are only shown to point out the equivalence of both implementations.

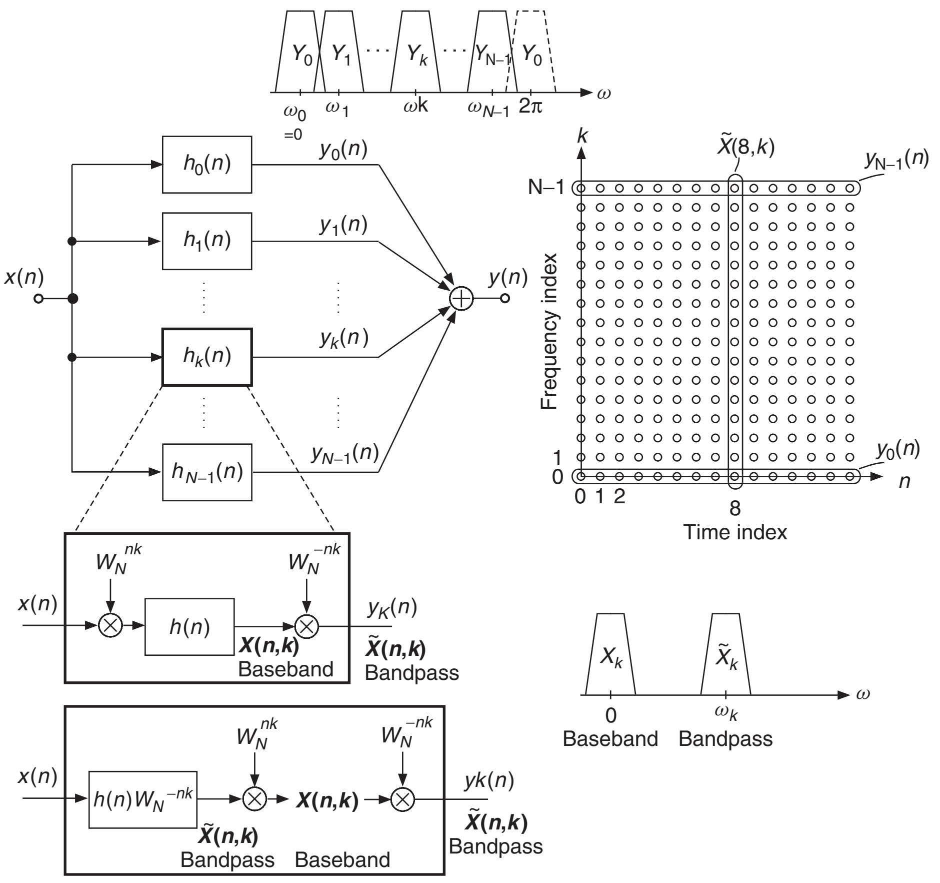

Y_0 Y_1 Y_k Y_{N-1} Y_0
$\omega_0 = 0$ ω_1 ωk ω_{N-1} 2π ω
$x(n)$
$h_0(n)$ $h_1(n)$ $h_k(n)$ $h_{N-1}(n)$
$y_0(n)$ $y_1(n)$ $y_k(n)$ $y_{N-1}(n)$
$y(n)$
k $\tilde{X}(8,k)$ $y_{N-1}(n)$
N−1
Frequency index
1
0
$y_0(n)$
n
0 1 2
8
Time index
W_N^{nk} W_N^{-nk}
$h(n)$
$y_K(n)$
$X(n,k)$ Baseband
$\tilde{X}(n,k)$ Bandpass
X_k $\tilde{X}_k$
0 Baseband
ω_k Bandpass
ω
$h(n)W_N^{-nk}$
$X(n,k)$
$yk(n)$
$\tilde{X}(n,k)$ Bandpass Baseband
$\tilde{X}(n,k)$ Bandpass

Figure 7.4 Filter bank description of the short-time Fourier transform. Two implementations of the kth channel are shown in the lower left part. The discrete-time and discrete-frequency plane is shown in the right part. The marked bandpass signals $y_k(n)$ are the horizontal samples $\tilde{X}(n, k)$. The different frequency bands Y_k corresponding to each bandpass signal are shown on top of the filter bank. The frequency bands for the baseband signal $X(n, k)$ and the bandpass signal $\tilde{X}(n, k)$ are shown in the lower right part.

The output sequence $y(n)$ is the sum of the bandpass signals according to

$$y(n) = \sum_{k=0}^{N-1} y_k(n) = \sum_{k=0}^{N-1} \tilde{X}(n, k) = \sum_{k=0}^{N-1} X(n, k)W_N^{-nk}. \tag{7.11}$$

The output signals $y_k(n)$ are complex-valued sequences $\tilde{X}(n, k)$. For a real-valued input signal $x(n)$ the bandpass signals satisfy the property $y_k(n) = \tilde{X}(n, k) = \tilde{X}^*(n, N - k) = y^*_{N-k}(n)$. For a channel stacking with $\omega_k = \frac{2\pi k}{N}$ we get the frequency bands shown in the upper part of Figure 7.4. The property $\tilde{X}(n, k) = \tilde{X}^*(n, N - k)$, together with the channel stacking can be used for the formulation of real-valued bandpass signals (real-valued kth channel)

$$\hat{y}_k(n) = \tilde{X}(n, k) + \tilde{X}(n, N - k) = \tilde{X}(n, k) + \tilde{X}^*(n, k) \tag{7.12}$$

$$= |X(n, k)| \cdot \left[e^{j\tilde{\varphi}(n,k)} + e^{-j\tilde{\varphi}(n,k)}\right] \tag{7.13}$$

$$= 2|X(n,k)| \cdot \cos[\tilde{\varphi}(n,k)]$$
$$\text{for}\, k = 1, \ldots, N/2 - 1. \tag{7.14}$$

This leads to

$$\hat{y}_0(n) = y_0(n), \quad k = 0 \tag{7.15}$$

dc channel

$$\hat{y}_k(n) = \underbrace{2|X(n,k)|}_{A(n,k)} \cdot \cos\underbrace{[\omega_k n + \varphi(n,k)]}_{\tilde{\varphi}(n,k)}, \quad k = 1, \ldots, N/2 - 1 \tag{7.16}$$

bandpass channels

$$\hat{y}_{N/2}(n) = y_{N/2}(n), \quad k = N/2 \tag{7.17}$$

highpass channel.

Besides a dc and a highpass channel we have $N/2 - 1$ cosine signals with fixed frequencies ω_k and time-varying amplitude and phase. This means that we can add real-valued output signals $\hat{y}_k(n)$ to yield the output signal

$$y(n) = \sum_{k=0}^{N/2} \hat{y}_k(n). \tag{7.18}$$

This interpretation offers analysis of a signal by a filter bank, modification of the short-time spectrum $\tilde{X}(n,k)$ on a sample-by-sample basis and synthesis by a summation of the bandpass signals $y_k(n)$. Due to the fact that the baseband signals are bandlimited by the lowpass filter $h(n)$, a sampling rate reduction can be performed in each channel to yield $X(sR,k)$, where only every Rth sample is taken and s denotes the new time index. This leads to a short-time transform $X(sR,k)$ with a hop size of R samples. Before the synthesis upsampling and interpolation filtering have to be performed [CR83].

7.2.2 Block-by-block analysis/synthesis model

A detailed description of a phase vocoder implementation using the FFT can be found in [Por76, Cro80, CR83]. The analysis and synthesis implementations are precisely described in [(CR83, p. 318, Figure 7.19 and 7.19 p. 321, Figure 7.20)]. A simplified analysis and synthesis implementation, where the window length is less or equal to the FFT length, were proposed in [Cro80]. The analysis and synthesis algorithm and the discrete-time and discrete-frequency plane are shown in Figure 7.5. The analysis algorithm [Cro80] is given by

$$X(sR_a,k) = \sum_{m=-\infty}^{\infty} x(m)h(sR_a - m)W_N^{mk} \tag{7.19}$$

$$= W_N^{sR_ak} \sum_{m=-\infty}^{\infty} x(m)h(sR_a - m)W_N^{-(sR_A-m)k} \tag{7.20}$$

$$= W_N^{sR_ak} \cdot \tilde{X}(sR_a,k) \tag{7.21}$$

$$= X_R(sR_a,k) + jX_I(sR_a,k) = |X(sR_a,k)| \cdot e^{j\varphi(sR_a,k)} \tag{7.22}$$

$$k = 0, 1, \ldots, N-1,$$

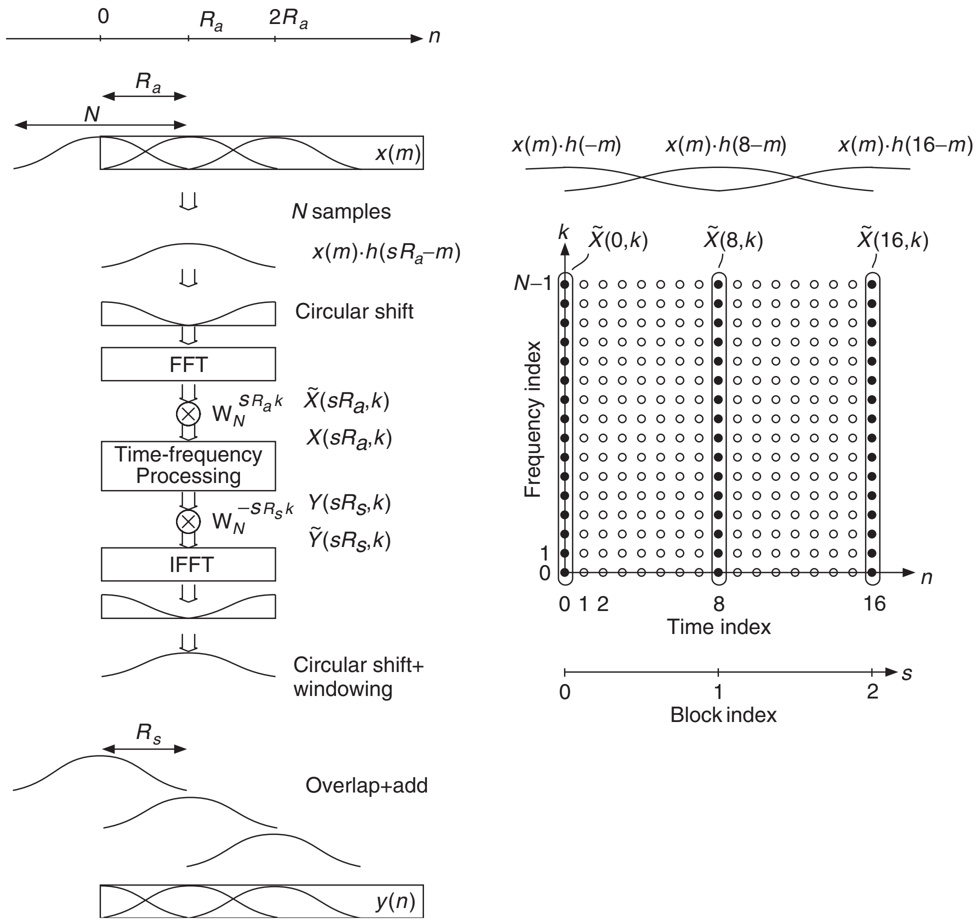


Figure 7.5 Phase vocoder using the FFT/IFFT for the short-time Fourier transform. The analysis hop size R_a determines the sampling of the two-dimensional time-frequency grid. Time-frequency processing allows the reconstruction with a synthesis hop size R_s.

where the short-time Fourier transform is sampled every R_a samples in time and s denotes the time index of the short-time transform at the decimated sampling rate. This means that the time index is now $n = sR_a$, where R_a denotes the analysis hop size. The analysis window is denoted by $h(n)$. Notice that $X(n,k)$ and $\tilde{X}(n,k)$ in the FFT implementation can also be found in the filter bank approach. The circular shift of the windowed segment before the FFT and after the IFFT is derived in [CR83] and provides a zero-phase analysis and synthesis regarding the center of the window. Further details will be discussed in the next section. Spectral modifications in the time-frequency plane can now be done, which yields $Y(sR_s,k)$, where R_s is the synthesis hop size. The synthesis algorithm [Cro80] is given by

$$y(n) = \sum_{s=-\infty}^{\infty} f(n - sR_s)y_s(n - sR_s) \tag{7.23}$$

$$\text{with} \quad y_s(n) = \frac{1}{N}\sum_{k=0}^{N-1}\left[W_N^{-sR_sk}Y(sR_s,k)\right]W_N^{-nk},$$

where $f(n)$ denotes the synthesis window. Finite length signals $y_s(n)$ are derived from inverse transforms of short-time spectra $Y(sR_s, k)$. These short-time segments are weighted by the synthesis window $f(n)$ and then added by the overlap-add procedure given by (7.23) (see Figure 7.5).

7.3 Phase vocoder implementations

This section describes several phase vocoder implementations for digital audio effects. A useful representation is the time-frequency plane where one displays the values of the magnitude $|X(n,k)|$ and phase $\tilde{\varphi}(n,k)$ of the $\tilde{X}(n,k)$ signal. If the sliding Fourier transform is used as an analysis scheme, this graphical representation is the combination of the spectrogram, which displays the magnitude values of this representation, and the *phasogram*, which displays the phase. However, phasograms are harder to read when the hop size is not small. Figure 7.6 shows a spectrogram and a phasogram which correspond to the discrete-time and discrete-frequency plane achieved by a filter bank (see Figure 7.4) or a block-by-block FFT analysis (see Figure 7.5) described in the previous section. In the horizontal direction a line represents the output magnitude $|X(n,k)|$ and the phase $\tilde{\varphi}(n,k)$ of the kth analysis bandpass filter over the time index n. In the vertical direction a line represents the magnitude $|X(n,k)|$ and phase $\tilde{\varphi}(n,k)$ for a fixed time index n, which corresponds to a short-time spectrum over frequency bin k at the center of the analysis window located at time index n. The spectrogram in Figure 7.6 with frequency range up to 2 kHz shows five horizontal rays over the time axis, indicating the magnitude of the harmonics of the analyzed sound segment. The phasogram shows the corresponding phases for all five horizontal rays $\tilde{\varphi}(n,k)$, which rotate according to the frequencies of the five harmonics. With a hop size of one we get a visible tree structure. For a larger hop size we get a sampled version, where the tree structure usually disappears.

The analysis and synthesis part can come from the filter bank summation model (see basics), in which case the resynthesis part consists in summing sinusoids, whose amplitudes and frequencies are coming from a parallel bank of filters. The analysis part can also come from a sliding FFT algorithm, in which case it is possible to perform the resynthesis with either a summation of sinusoids or an IFFT approach.

7.3.1 Filter bank approach

From a musician's point of view the idea behind this technique is to represent a sound signal as a sum of sinusoids. Each of these sinusoids is modulated in amplitude and frequency. They represent filtered versions of the original signal. The manipulation of the amplitudes and frequencies of these individual signals will produce a digital effect, including time stretching or pitch shifting.

One can use a filter bank, as shown in Figure 7.7, to split the audio signal into several filtered versions. The sum of these filtered versions reproduces the original signal. For a perfect reconstruction the sum of the filter frequency responses should be unity. In order to produce a digital audio effect, one needs to alter the intermediate signals that are analytical signals consisting of real and imaginary parts (double lines in Figure 7.7). The implementation of each filter can be performed by a heterodyne filter, as shown in Figure 7.8.

The implementation of a stage of a heterodyne filter consists of a complex-valued oscillator with a fixed frequency ω_k, a multiplier and an FIR filter. The multiplication shifts the spectrum of the sound, and the FIR filter limits the width of the frequency-shifted spectrum. This heterodyne filtering can be used to obtain intermediate analytic signals, which can be put in the form

$$X(n,k) = \left[x(n)\cdot \mathrm{e}^{-j\omega_k n}\right] * h(n) = X_R(n,k) + jX_I(n,k) \tag{7.24}$$

$$= |X(n,k)|\mathrm{e}^{j\varphi(n,k)} \tag{7.25}$$

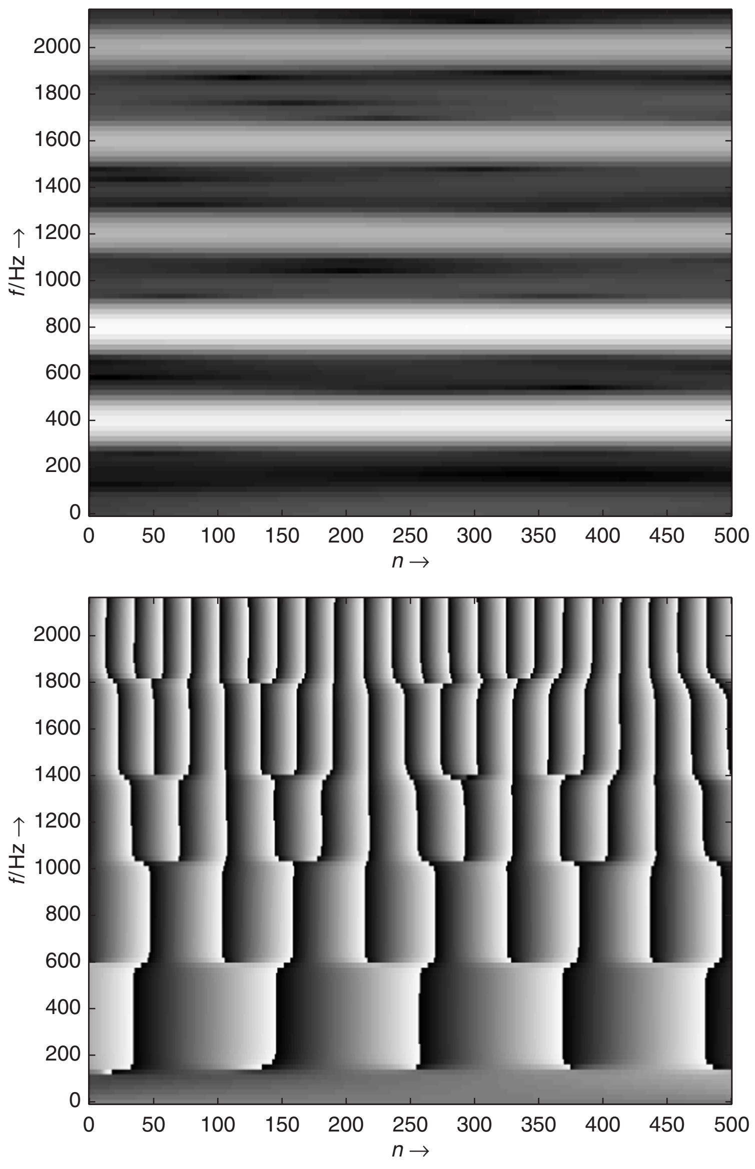


Figure 7.6 Magnitude $|X(n,k)|$ (upper plot) and phase $\tilde{\varphi}(n,k)$ (lower plot) display of a sliding Fourier transform with a hop size $R_a = 1$ or a filter-bank analysis approach. For the upper display the grey value (black = 0 and white = maximum amplitude) represents the magnitude range. In the lower display the phase values are in the range $-\pi \leq \tilde{\varphi}(n,k) \leq \pi$.

$$X_R(n,k) = |X(n,k)| \cos(\varphi(n,k)) \tag{7.26}$$

$$X_I(n,k) = |X(n,k)| \sin(\varphi(n,k)). \tag{7.27}$$

The difference from classical bandpass filtering is that here the output signal is located in the baseband. This representation leads to a slowly varying phase $\varphi(n,k)$ and the derivation of the

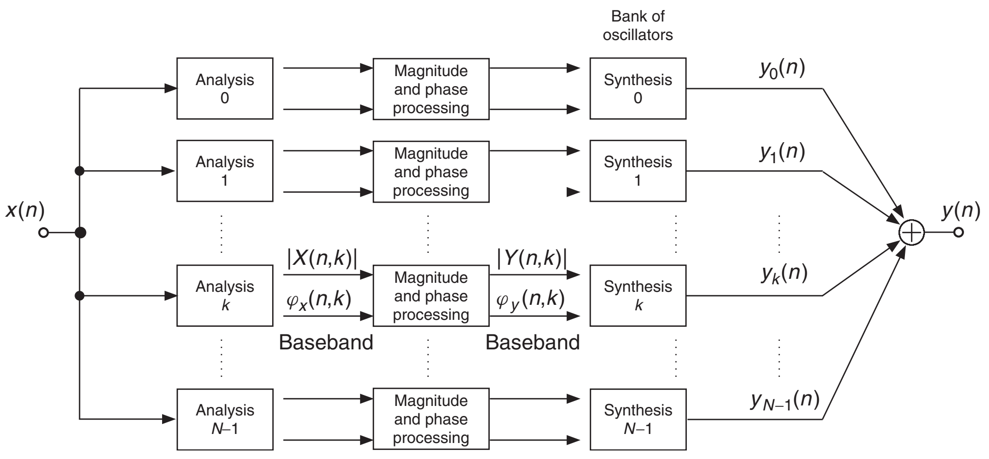


Figure 7.7 Filter-bank implementation.

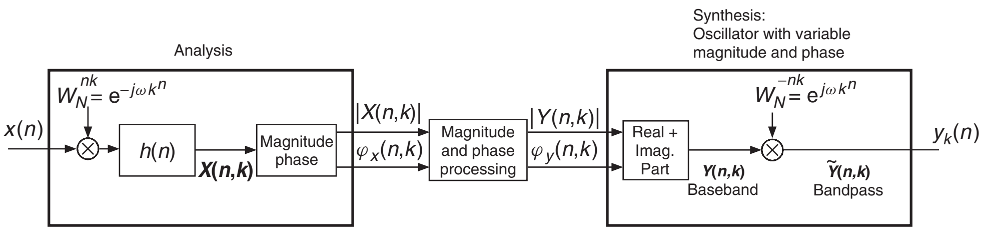


Figure 7.8 Heterodyne-filter implementation.

phase is a measure of the frequency deviation from the center frequency ω_k. A sinusoid $x(n) = \cos[\omega_k n + \varphi_0]$ with frequency ω_k can be written as $x(n) = \cos[\tilde{\varphi}(n)]$, where $\tilde{\varphi}(n) = \omega_k n + \varphi_0$. The derivation of $\tilde{\varphi}(n)$ gives the frequency $\omega_k = \frac{d\tilde{\varphi}(n)}{dn}$. The derivation of the phase $\tilde{\varphi}(n, k)$ at the output of a bandpass filter is termed the *instantaneous frequency* given by

$$\omega_i(n,k) = 2\pi f_i(n,k)/f_S \tag{7.28}$$

$$= \frac{\mathrm{d}}{\mathrm{d}n}\tilde{\varphi}(n,k) \tag{7.29}$$

$$= \omega_k + \frac{\mathrm{d}}{\mathrm{d}n}\varphi(n,k) \tag{7.30}$$

$$= \omega_k + \varphi(n,k) - \varphi(n-1,k) \tag{7.31}$$

$$\Rightarrow f_i(n,k) = \left(\frac{k}{N} + \frac{\varphi(n,k) - \varphi(n-1,k)}{2\pi}\right) \cdot f_S. \tag{7.32}$$

The instantaneous frequency can be described in a musical way as the frequency of the filter output signal in the filter-bank approach. The phase of the baseband output signal is $\varphi(n, k)$ and the phase of the bandpass output signal is $\tilde{\varphi}(n, k) = \omega_k n + \varphi(n, k)$ (see Figure 7.8). As soon as we have the instantaneous frequencies, we can build an oscillator bank and eventually change the amplitudes and frequencies of this bank to build a digital audio effect. The recalculation of the phase from a modified instantaneous frequency is done by computing the phase according to

$$\tilde{\varphi}(n,k) = \tilde{\varphi}(0,k) + \int_0^{nT} 2\pi f_i(\tau,k)\mathrm{d}\tau. \tag{7.33}$$

The result of the magnitude and phase processing can be written as $Y(n,k) = |Y(n,k)|\mathrm{e}^{j\varphi_y(n,k)}$, which is then used as the magnitude and phase for the complex-valued oscillator running with frequency ω_k. The output signal is then given by

$$\tilde{Y}(n,k) = |Y(n,k)|\mathrm{e}^{j\varphi_y(n,k)} \cdot \mathrm{e}^{j\omega_k n} \tag{7.34}$$

$$= Y(n,k) \cdot \mathrm{e}^{j\omega_k n} = \left[Y_R(n,k) + jY_I(n,k)\right] \cdot \mathrm{e}^{j\omega_k n}. \tag{7.35}$$

The resynthesis of the output signal can then be performed by summing all the individual back-shifted signals (oscillator bank) according to

$$y(n) = \sum_{k=0}^{N-1} \tilde{Y}(n,k) = \sum_{k=0}^{N-1} Y(n,k) \cdot \mathrm{e}^{j\Omega_k n} \tag{7.36}$$

$$= \sum_{k=0}^{N/2} A(n,k)\cos\left[\omega_k n + \varphi_y(n,k)\right], \tag{7.37}$$

where (7.37) was already introduced by (7.18). The modification of the phases and frequencies for time stretching and pitch shifting needs further explanation and will be treated in a following subsection.

The following M-file 7.1 shows a filter-bank implementation with heterodyne filters, as shown in Figure 7.8 (see also Figure 7.4).

M-file 7.1 (VX_het_nothing.m)

```
% VX_het_nothing.m   [DAFXbook, 2nd ed., chapter 7]
clear; clf
%===== This program (i) implements a heterodyne filter bank,
%===== then (ii) filters a sound through the filter bank
%===== and (iii) reconstructs a sound

%----- user data -----
fig_plot      = 0;    % use any value except 0 or [] to plot figures
s_win         = 256;  % window size
n_channel     = 128;  % nb of channels
s_block       = 1024; % computation block size (must be a multiple of s_win)
[DAFx_in, FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
window   = hanning(s_win, 'periodic');
s_buffer = length(DAFx_in);
DAFx_in  = [DAFx_in; zeros(s_block,1)]/max(abs(DAFx_in)); %0-pad & normalize
DAFx_out = zeros(length(DAFx_in),1);
X        = zeros(s_block, n_channel);
z        = zeros(s_win-1, n_channel);
```

```
%----- initialize the heterodyn filters -----
t   = (0:s_block-1)';
het = zeros(s_block,n_channel);
for k=1:n_channel
  wk        = 2*pi*i*(k/s_win);
  het(:,k)  = exp(wk*(t+s_win/2));
  het2(:,k) = exp(-wk*t);
end

%----- displays the phase of the filter -----
if(fig_plot)
  colormap(gray); imagesc(angle(het)'); colorbar;
  axis('xy'); xlabel('n \rightarrow'); ylabel('k \rightarrow');
  title('Heterodyn filter bank: initial \phi(n,k)'); pause;
end

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pend = length(DAFx_in) - s_block;
while pin<pend
  grain = DAFx_in(pin+1:pin+s_block);
%============================================
  %----- filtering through the filter bank -----
  for k=1:n_channel
    [X(:,k), z(:,k)] = filter(window, 1, grain.*het(:,k), z(:,k));
  end
  X_tilde = X.*het2;
  %----- drawing -----
  if(fig_plot)
    imagesc(angle(X_tilde')); axis('xy'); colorbar;
    xlabel('n \rightarrow'); ylabel('k \rightarrow');
    txt = sprintf('Heterodyn filter bank: \\phi(n,k),t=%6.3f s',(pin+1)/FS);
    title(txt); drawnow;
  end
  %----- sound reconstruction -----
  res = real(sum(X_tilde,2));
%============================================
  DAFx_out(pin+1:pin+s_block) = res;
  pin = pin + s_block;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
DAFx_out = DAFx_out(n_channel+1:n_channel+s_buffer) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_het_nothing.wav');
```

M-file 7.2 demonstrates the second-filter bank implementation with complex-valued bandpass filters, as shown in Figure 7.4.

M-file 7.2 (VX_filter_nothing.m)

```
% VX_filter_nothing.m   [DAFXbook, 2nd ed., chapter 7]
clear; clf

%===== This program (i) performs a complex-valued filter bank
%===== then (ii) filters a sound through the filter bank
%===== and (iii) reconstructs a sound

%----- user data -----
fig_plot     = 0;    % use any value except 0 or [] to plot figures
s_win        = 256;  % window size
nChannel     = 128;  % nb of channels
n1           = 1024; % block size for calculation
[DAFx_in,FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
window   = hanning(s_win, 'periodic');
L        = length(DAFx_in);
DAFx_in  = [DAFx_in; zeros(n1,1)] / max(abs(DAFx_in)); % 0-pad & normalize
DAFx_out = zeros(length(DAFx_in),1);
X_tilde  = zeros(n1,nChannel);
z        = zeros(s_win-1,nChannel);

%----- initialize the complex-valued filter bank -----
t    = (-s_win/2:s_win/2-1)';
filt = zeros(s_win, nChannel);
for k=1:nChannel
  wk        = 2*pi*i*(k/s_win);
  filt(:,k) = window.*exp(wk*t);
end

if(fig_plot), colormap(gray); end

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pend = length(DAFx_in) - n1;
while pin<pend
  grain = DAFx_in(pin+1:pin+n1);
%============================================
  %----- filtering -----
  for k=1:nChannel
    [X_tilde(:,k),z(:,k)] = filter(filt(:,k),1,grain,z(:,k));
  end
  if(fig_plot)
    imagesc(angle(X_tilde')); axis('xy'); colorbar;
    xlabel('n \rightarrow'); ylabel('k \rightarrow');
    txt = sprintf('Complex-valued fil. bank: \\phi(n,k), t=%6.3f s', (pin+1)/FS);
   title(txt); drawnow;
  end
    %----- sound reconstruction -----
  res = real(sum(X_tilde,2));
%============================================
  DAFx_out(pin+1:pin+n1) = res;
  pin = pin + n1;
  end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc
```

```
%----- listening and saving the output -----
DAFx_out = DAFx_out(nChannel+1:nChannel+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_filter_nothing.wav');
```

7.3.2 Direct FFT/IFFT approach

The FFT algorithm also calculates the values of the magnitudes and phases within a time frame, allowing a shorter calculation time. So many analysis-synthesis algorithms use this transform. There are different ways to interpret a sliding Fourier transform, and consequently to invent a method of resynthesis starting from this time-frequency representation. The first one is to apply the inverse FFT on each short-time spectrum and use the overlap-add method to reconstruct the signal. The second one is to consider a horizontal line of the time-frequency representation (constant frequency versus time) and to reconstruct a filtered version for each line. The third one is to consider each point of the time-frequency representation and to make a sum of small grains called *gaborets*. In each interpretation one must test the ability of obtaining a perfect reconstruction if one does not modify the representation. Another important fact is the ability to provide effect implementations that do not have too many artifacts when one modifies on purpose the values of the sliding FFT, especially in operations such as time stretching or filtering.

We now describe the direct FFT/IFFT approach. A time-frequency representation can be seen as a series of overlapping FFTs with or without windowing. As the FFT is invertible, one can reconstruct a sound by adding the inverse FFT of a vertical line (constant time versus frequency), as shown in Figure 7.9.

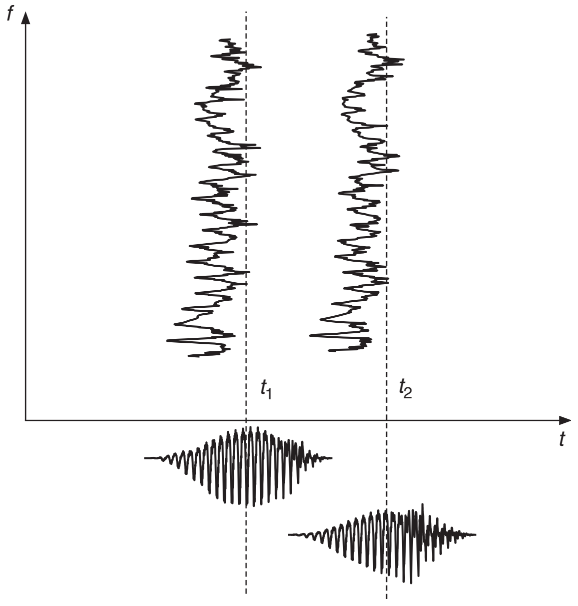


Figure 7.9 FFT and IFFT: vertical line interpretation. At two time instances two spectra are used to compute two time segments.

A perfect reconstruction can be achieved, if the sum of the overlapping windows is unity (see Figure 7.10). A modification of the FFT values can produce time aliasing, which can be avoided

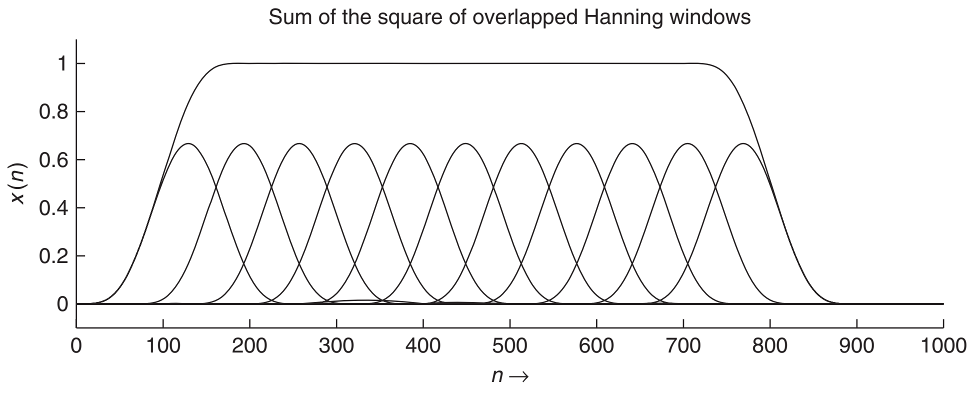


Figure 7.10 Sum of small windows.

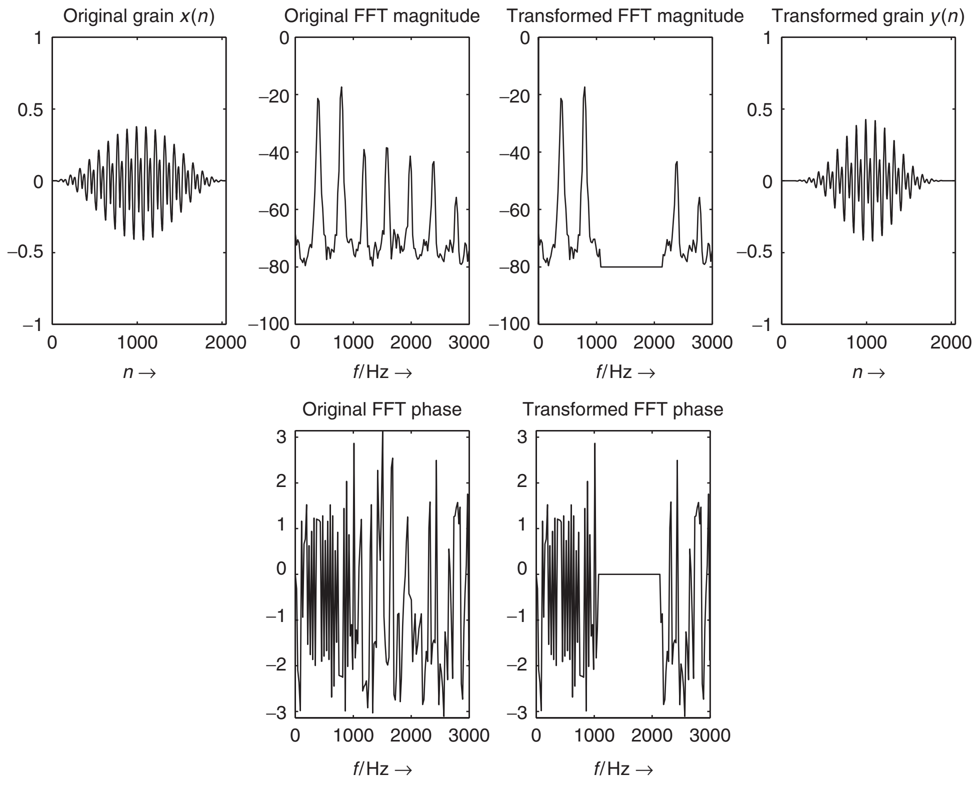


Figure 7.11 Sound windowing, FFT modification and IFFT.

by either zero-padded windows or using windowing after the inverse FFT. In this case the product of the two windows has to be unity. An example is shown in Figure 7.11. This implementation will be used most frequently in this chapter.

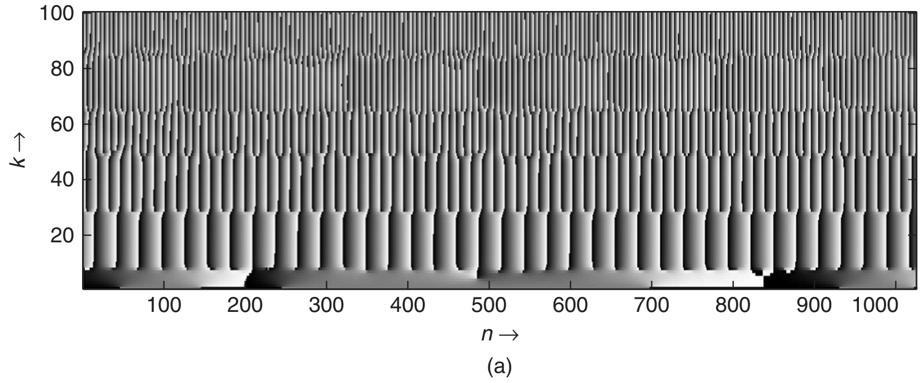
The following M-file 7.3 shows a phase vocoder implementation based on the direct FFT/IFFT approach, where the routine itself is given two vectors for the sound, a window and a hop size.

M-file 7.3 (VX_pv_nothing.m)

```
% VX_pv_nothing.m   [DAFXbook, 2nd ed., chapter 7]
%===== this program implements a simple phase vocoder
clear; clf

%----- user data -----
fig_plot      = 0;     % use any value except 0 or [] to plot figures
n1            = 512;   % analysis step [samples]
n2            = n1;    % synthesis step [samples]
s_win         = 2048;  % window size [samples]
[DAFx_in, FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
w1            = hanning(s_win, 'periodic'); % input window
w2            = w1;    % output window
L             = length(DAFx_in);
DAFx_in       = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));  % 0-pad & normalize
DAFx_out      = zeros(length(DAFx_in),1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
  f     = fft(fftshift(grain));  % FFT
  r     = abs(f);                % magnitude
  phi   = angle(f);              % phase
  ft    = (r.* exp(i*phi));      % reconstructed FFT
  grain = fftshift(real(ifft(ft))).*w2;
% ===========================================
  DAFx_out(pout+1:pout+s_win) = ...
     DAFx_out(pout+1:pout+s_win) + grain;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc
%----- listening and saving the output -----
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_pv_nothing.wav');
```

The kernel algorithm performs successive FFTs, inverse FFTs and overlap-add of successive grains. The key point of the implementations is how to go from the FFT representation, where the time origin is at the beginning of the window, to the phase vocoder representation used in Section 7.2, either in its filter-bank description or its block-by-block approach.

The first problem we have to solve is the fact that the time origin for an FFT is on the left of the window. We would like to have it centered, so that, for example, the FFT of a centered impulse would be zero phase. This is done by a circular shift of the signal, which is a commutation of the first and second part of the buffer. The discrete-time Fourier transform of $\hat{x}(n) = x(n - N/2)$ is $\hat{X}(e^{j\omega}) = e^{-j\omega\frac{N}{2}} X(e^{j\omega})$. With $\omega_k = \frac{2\pi}{N}k$ the discrete Fourier transform gives $\hat{X}(k) = e^{-j\frac{2\pi}{N}k\frac{N}{2}} X(k)$, which is equivalent to $\hat{X}(k) = (-1)^k X(k)$. The circular shift in time domain can be achieved by multiplying the result of the FFT by $(-1)^k$. With this circular shift, the output of the FFT is equivalent to a filter bank, with zero phase filters. When analyzing a sine wave, the display of the values of the phase $\tilde{\varphi}(n, k)$ of the time-frequency representation will follow the phase of the sinusoid. When analyzing a harmonic sound, one obtains a tree with successive branches corresponding to every harmonic (top of Figure 7.12).



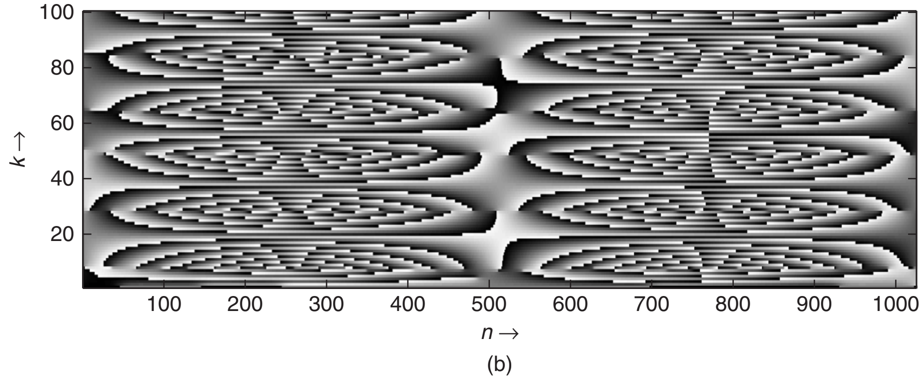


Figure 7.12 Different phase representations: (a) $\tilde{\varphi}(n, k)$ and (b) $\varphi(n, k) = \tilde{\varphi}(n, k) - 2\pi mk/N$.

If we want to take an absolute value as the origin of time, we have to switch to the notation used in Section 7.2. We have to multiply the result of the FFT by W_N^{mk}, where m is the time sample in the middle of the FFT and k is the number of the bin of the FFT. In this way the display of the phase $\varphi(n, k)$ (bottom of Figure 7.12) corresponds to a frequency which is the difference between the frequency of the analyzed signal (here a sine wave) delivered by the FFT and the analyzing frequency (the center of the bin). The phase $\varphi(n, k)$ is calculated as $\varphi(n, k) = \tilde{\varphi}(n, k) - 2\pi mk/N$ (N length of FFT, k number of the bin, m time index).

7.3.3 FFT analysis/sum of sinusoids approach

Conversely, one can read a time-frequency representation with horizontal lines, as shown in Figure 7.13. Each point on a horizontal line can be seen as the convolution of the original signal

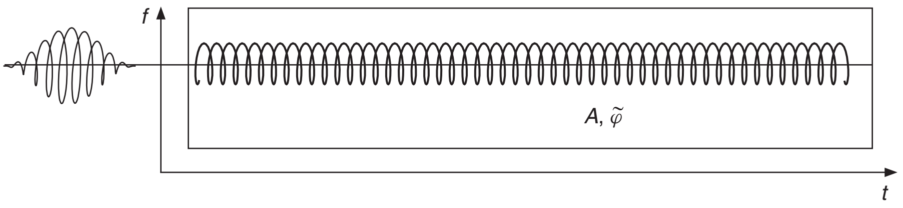


Figure 7.13 Filter-bank approach: horizontal line interpretation.

with an FIR filter, whose filter coefficients have been given by (7.4). The filter-bank approach is very close to the heterodyne-filter implementation. The difference comes from the fact that for heterodyne filtering the complex exponential is running with time and the sliding FFT is considering for each point the same phase initiation of the complex exponential. It means that the heterodyne filter measures the phase deviation between a cosine and the filtered signal, and the sliding FFT measures the phase with a time origin at zero.

The reconstruction of a sliding FFT on a horizontal line with a hop size of one is performed by filtering of this line with the filter corresponding to the frequency bin (see Figure 7.13). However, if the analysis hop size is greater than one, we need to interpolate the magnitude values $|X(n,k)|$ and phase values $\tilde{\varphi}(n,k)$. Phase interpolation is based on phase unwrapping, which will be explained in Section 7.3.5. Combining phase interpolation with linear interpolation of the magnitudes $|X(n,k)|$ allows the reconstruction of the sound by the addition of a bank of oscillators, as given in (7.37).

M-file 7.4 illustrates the interpolation and the sum of sinusoids. Starting from the magnitudes and phases taken from a sliding FFT the synthesis implementation is performed by a bank of oscillators. It uses linear interpolation of the magnitudes and phases.

M-file 7.4 (VX_bank_nothing.m)

```
% VX_bank_nothing.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs an FFT analysis and oscillator bank synthesis
clear; clf

%----- user data -----
n1            = 200;   % analysis step [samples]
n2            = n1;    % synthesis step [samples]
s_win         = 2048;  % window size [samples]
[DAFx_in, FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
w1    = hanning(s_win, 'periodic'); % input window
w2    = w1;    % output window
L     = length(DAFx_in);
DAFx_in = [zeros(s_win, 1); DAFx_in; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in)); % 0-pad & normalize
DAFx_out = zeros(length(DAFx_in),1);
ll    = s_win/2;
omega = 2*pi*n1*[0:ll-1]'/s_win;
phi0  = zeros(ll,1);
r0    = zeros(ll,1);
psi   = zeros(ll,1);
grain = zeros(s_win,1);
res   = zeros(n2,1);
```

```
tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
  fc  = fft(fftshift(grain)); % FFT
  f   = fc(1:ll);             % positive frequency spectrum
  r   = abs(f);               % magnitudes
  phi = angle(f);             % phases
  %----- unwrapped phase difference on each bin for a n2 step
  delta_phi = omega + princarg(phi-phi0-omega);
  %----- phase and magnitude increment, for linear
  %      interpolation and reconstruction -----
  delta_r   = (r-r0)/n1;    % magnitude increment
  delta_psi = delta_phi/n1; % phase increment
  for k=1:n2 % compute the sum of weighted cosine
    r0     = r0 + delta_r;
    psi    = psi + delta_psi;
    res(k) = r0'*cos(psi);
  end
  %----- for next time -----
  phi0 = phi;
  r0   = r;
  psi  = princarg(psi);
% ===========================================
  DAFx_out(pout+1:pout+n2) = DAFx_out(pout+1:pout+n2) + res;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win/2+n1+1:s_win/2+n1+L) / max(abs(DAFx_out));
soundsc(DAFx_out,FS);
wavwrite(DAFx_out, FS, 'la_bank_nothing.wav');
```

7.3.4 Gaboret approach

The idea of the “gaboret approach” is the reconstruction of a signal from a time-frequency representation with the sum of “gaborets” weighted by the values of the time-frequency representation [AD93]. The shape of a gaboret is a windowed exponential (see Figure 7.14), which can be given by $g_{\omega_k}(n) = \mathrm{e}^{-j\omega_k n} g_\alpha(n)$. The approach is based on the Gabor transform, which is a short-time Fourier transform with the smallest time-frequency window, namely a Gaussian function $g_\alpha(n) = \frac{1}{\sqrt{2\pi\alpha}} e^{-\frac{n^2}{2\alpha}}$ with $\alpha > 0$. The discrete-time Fourier transform of $g_\alpha(n)$ is again a Gaussian function in the Fourier domain. The gaboret approach is very similar to the wavelet transform [CGT89, Chu92]: one does not consider time or frequency as a privileged axis and one point of the time-frequency plane is the scalar product of the signal with a small gaboret. Further details

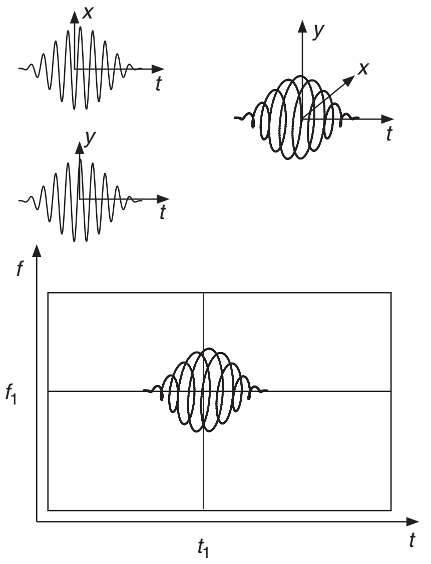


Figure 7.14 Gaboret approach: the upper left part shows real and imaginary values of a gaboret and the upper right part shows a possible 3D repesentation with axes t, x and y. The lower part shows a gaboret associated to a specific point of a time-frequency representation (for every point we can generate a gaboret in the time domain and then make the sum of all gaborets).

can be found in [QC93, WR90]. The reconstruction from a time-frequency representation is the sum of gaborets weighted by the values of this time-frequency plane according to

$$y(n) = \sum_{s=-\infty}^{\infty} \sum_{k=0}^{N-1} Y(sR_s, k) f(n - sR_s) W_N^{-nk}. \tag{7.38}$$

Although this point of view is totally equivalent to windowing plus FFT/IFFT plus windowing, it allows a good comprehension of what happens in case of modification of a point in the plane.

The reconstruction of one single point of a time-frequency representation yields a gaboret in the time domain, as shown in Figure 7.15. Then a new time-frequency representation of this gaboret is computed. We get a new image, which is the called the *reproducing kernel* associated with the transform. This new time-frequency representation is different from the single point of the original time-frequency representation.

So a time-frequency representation of a real signal has some constraints: each value of the time-frequency plane must be the convolution of the neighborhood by the *reproducing kernel* associated with the transformation. This means that if an image (time-frequency representation) is not valid and if we force the reconstruction of a sound by the weighted summation of gaborets, the time-frequency representation of this transformed sound will be in a different form than the initial time-frequency representation. There is no way to avoid this and the beautiful art of making good transforms often relies on the ability to provide “quasi-valid” representations [AD93].

This *reproducing kernel* is only a 2 D extension of the well-known problem of windowing: we find the shape of the FFT of the window around one horizontal ray. But it brings new aspects. When we have two spectral lines, their time-frequency representations are blurred and, when summed, appear as beats. Illustrative examples are shown in Figure 7.16. The shape of the *reproducing kernel*

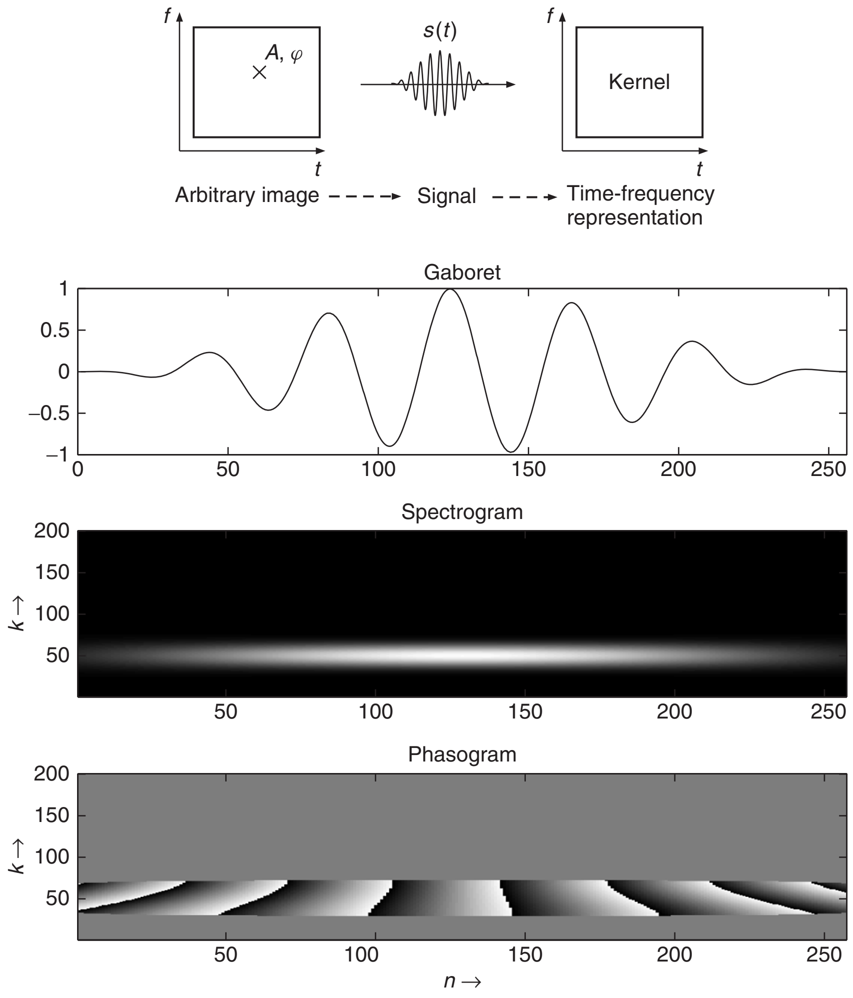


Figure 7.15 Reproducing kernel: the lower three plots represent the forced gaboret and the reproducing kernel consisting of spectrogram and phasogram. (Note: phase values only make sense when the magnitude is not too small.)

depends on the shape of the window and is the key point for differences in representations between different windows. The matter of finding spectral lines starting from time-frequency representations is the subject of Chapter 10. Here we only consider the fact that any signal can be generated as the sum of small gaborets. Frequency estimations in bins are obviously biased by the interaction between rays and additional noise.

The following M-file 7.5 demonstrates the gaboret analysis and synthesis approach.

M-file 7.5 (VX_gab_nothing.m)

```
% VX_gab_nothing.m   [DAFXbook, 2nd ed., chapter 7]
%==== This program performs signal convolution with gaborets
clear; clf
```

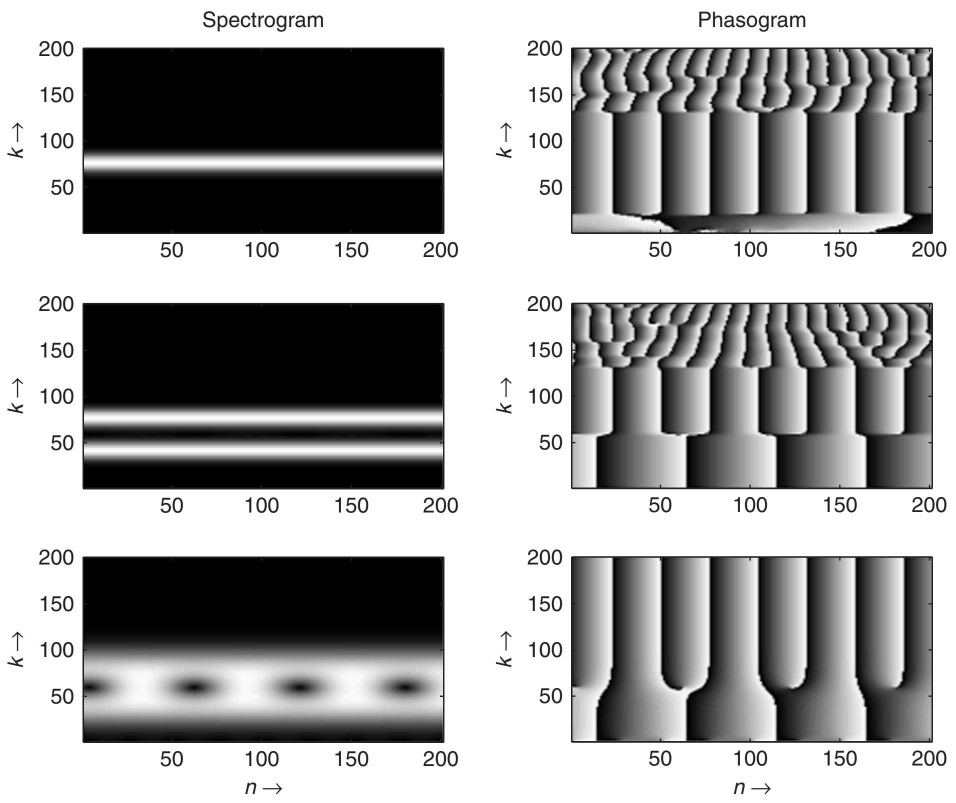


Figure 7.16 Spectrogram and phasogram examples: (a) upper part: the effect of the reproducing kernel is to thicken the line and giving a rotating phase at the frequency of the sinusoid; (b) middle part: for two sinusoids we have two lines with two rotations, if the window is large; (c) lower part: for two sinusoids with a shorter window the two lines mix and we can see beating.

```
%----- user data -----
n1            = 128;  % analysis step [samples]
n2            = n1;   % synthesis step [samples]
s_win         = 512;  % window size [samples]
[DAFx_in, FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
window   = hanning(s_win, 'periodic'); % input window
nChannel = s_win/2;
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1); % 0-pad & normalize

%----- initialize calculation of gaborets -----
t    = (-s_win/2:s_win/2-1);
gab  = zeros(nChannel,s_win);
for k=1:nChannel
  wk       = 2*pi*i*(k/s_win);
  gab(k,:) = window'.*exp(wk*t);
```

```
end

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win);
%===========================================
  %----- complex vector corresponding to a vertical line
  vec = gab*grain;
  %----- reconstruction from the vector to a grain
  res = real(gab'*vec);
% ===========================================
  DAFx_out(pout+1:pout+s_win) = DAFx_out(pout+1:pout+s_win) + res;
  pin  = pin + n1;
  pout = pout + n2;
  end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_gab_nothing.wav');
```

7.3.5 Phase unwrapping and instantaneous frequency

For the tasks of phase interpolation and instantaneous frequency calculation, for every frequency bin k we need a phase unwrapping algorithm. Starting from Figure 7.4 we perform unwrapping of

$$\tilde{\varphi}(n,k) = \underbrace{\frac{2\pi k}{N}}_{\omega_k} n + \varphi(n,k)$$

by unwrapping $\varphi(n,k)$ and adding the phase variation given by $\omega_k n$ for all k, as already shown by (7.10). We also need a special function which puts an arbitrary radian phase value into the range $[-\pi, \pi]$. We will call this function *principle argument* [GBA00], which is defined by the expression $y = \text{princarg}[2\pi m + \varphi_x] = \varphi_x$, where $-\pi < \varphi_x \leq \pi$ and m is an integer number. The corresponding MATLAB® function is shown in Figure 7.17.

M-file 7.6 `(princarg.m)`

```
function phase = princarg(phase_in)
% This function puts an arbitrary phase value into ]-pi,pi] [rad]
phase = mod(phase_in + pi,-2*pi) + pi;
```

The phase computations are based on the phase values $\tilde{\varphi}(sR_a, k)$ and $\tilde{\varphi}((s+1)R_a, k)$, which are the results of the FFTs of two consecutive frames. These phase values are shown in Figure 7.18. We now consider the phase values regardless of the frequency bin k. If a stable sinusoid with

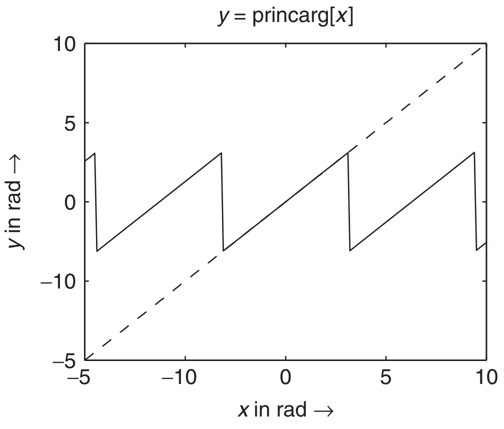


Figure 7.17 Principle argument function (**MATLAB** code and illustrative plot).

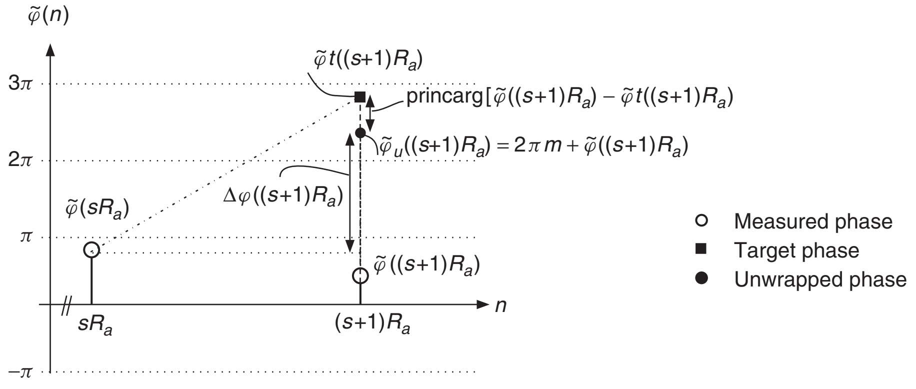

$\tilde{\varphi}(n)$
3π
2π
π
$-\pi$
$\tilde{\varphi}t((s+1)R_a)$
princarg$[\tilde{\varphi}((s+1)R_a) - \tilde{\varphi}t((s+1)R_a)$
$\tilde{\varphi}_u((s+1)R_a) = 2\pi m + \tilde{\varphi}((s+1)R_a)$
$\Delta\varphi((s+1)R_a)$
$\tilde{\varphi}(sR_a)$
$\tilde{\varphi}((s+1)R_a)$
sR_a
$(s+1)R_a$
n
○ Measured phase
■ Target phase
● Unwrapped phase

Figure 7.18 Basics of phase computations for frequency bin k.

frequency ω_k exists, we can compute a target phase $\tilde{\varphi}_t((s+1)R_a)$ from the previous phase value $\tilde{\varphi}(sR_a)$ according to

$$\tilde{\varphi}_t((s+1)R_a) = \tilde{\varphi}(sR_a) + \omega_k R_a. \tag{7.39}$$

The unwrapped phase

$$\tilde{\varphi}_u((s+1)R_a) = \tilde{\varphi}_t((s+1)R_a) + \tilde{\varphi}_d((s+1)R_a) \tag{7.40}$$

is computed by the target phase $\tilde{\varphi}_t((s+1)R_a)$ plus a deviation phase $\tilde{\varphi}_d((s+1)R_a)$. This deviation phase can be computed by the measured phase $\tilde{\varphi}((s+1)R_a)$ and the target phase $\tilde{\varphi}_t((s+1)R_a)$ according to

$$\tilde{\varphi}_d((s+1)R_a) = \text{princarg}\left[\tilde{\varphi}((s+1)R_a) - \tilde{\varphi}_t((s+1)R_a)\right]. \tag{7.41}$$

Now we formulate the unwrapped phase (7.40) with the deviation phase (7.41), which leads to the expression

$$\begin{aligned}\tilde{\varphi}_u((s+1)R_a) &= \tilde{\varphi}_t((s+1)R_a) + \text{princarg}\,[\tilde{\varphi}((s+1)R_a) - \tilde{\varphi}_t((s+1)R_a)]\\ &= \tilde{\varphi}(sR_a) + \omega_k R_a + \text{princarg}\,[\tilde{\varphi}((s+1)R_a) - \tilde{\varphi}(sR_a) - \omega_k R_a]\,.\end{aligned}$$

From the previous equation we can derive the unwrapped phase difference

$$\begin{aligned}\Delta\varphi((s+1)R_a) &= \tilde{\varphi}_u((s+1)R_a) - \tilde{\varphi}(sR_a)\\ &= \omega_k R_a + \text{princarg}\,[\tilde{\varphi}((s+1)R_a) - \tilde{\varphi}(sR_a) - \omega_k R_a] \qquad (7.42)\end{aligned}$$

between two consecutive frames. From this unwrapped phase difference we can calculate the instantaneous frequency for frequency bin k at time instant $(s+1)R_a$ by

$$f_i((s+1)R_a) = \frac{1}{2\pi}\frac{\Delta\varphi((s+1)R_a)}{R_a} f_S. \qquad (7.43)$$

The **MATLAB** instructions for the computation of the unwrapped phase difference given by (7.42) for every frequency bin k are given here:

```
omega     = 2*pi*n1*[0:ll-1]' / s_win;
% ll      = N/2 % with N length of the FFT
% n1      = R_a
delta_phi = omega + princarg(phi - phi0 - omega);
```

The term `phi` represents $\tilde{\varphi}((s+1)R_a)$ and `phi0` the previous phase value $\tilde{\varphi}(sR_a)$. In this manner `delta_phi` represents the unwrapped phase variation $\Delta\varphi((s+1)R_a)$ between two successive frames for every frequency bin k.

7.4 Phase vocoder effects

The following subsections will describe several modifications of a time-frequency representation before resynthesis in order to achieve audio effects. Most of them use the FFT analysis followed by either a summation of sinusoids or an IFFT synthesis, which is faster or more adapted to the effect. But all implementations give equivalent results and can be used for audio effects. Figure 7.19 provides a summary of those effects, with their relative operation(s) and the perceptual attribute(s) that are mainly modified.

7.4.1 Time-frequency filtering

Filtering a sound can be done with recursive (IIR) or non-recursive (FIR) filters. However, a musician would like to define or even to draw a frequency response which represents the gain for each frequency band. An intuitive way is to use a time-frequency representation and attenuate certain zones, by multiplying the FFT result in every frame by a filtering function in the frequency domain. One must be aware that in that case we are making a circular convolution (during the FFT–inverse FFT process), which leads to time aliasing, as shown in Figure 7.20. The alternative and exact technique for using time-frequency representations is the design of an FIR impulse response from the filtering function. The convolution of the signal segment $x(n)$ of length N with the impulse response of the FIR filter of length $N+1$ leads to $2N$-point sequence $y(n) = x(n) * h(n)$. This time-domain convolution or filtering can be performed more efficiently in the

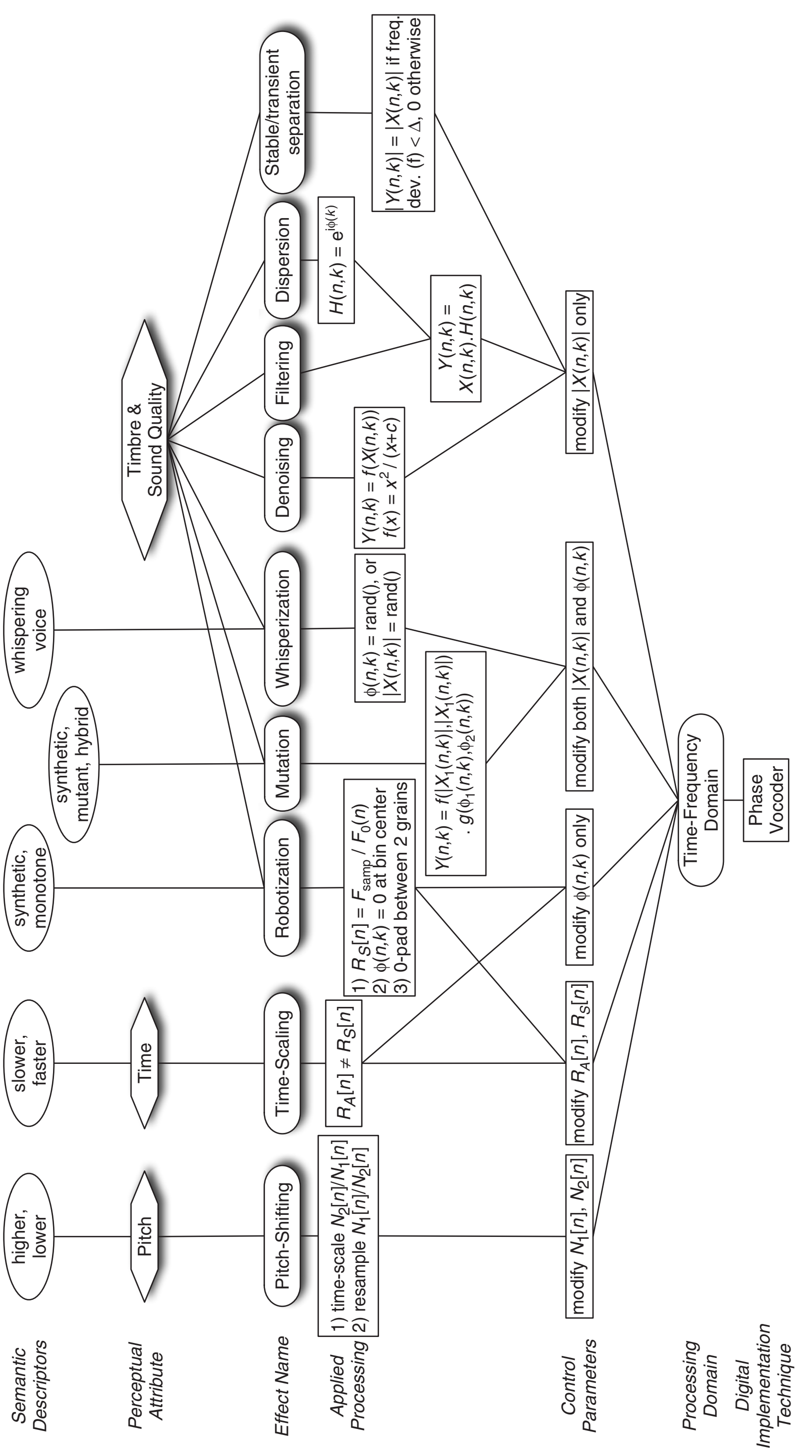

Semantic Descriptors
Perceptual Attribute
Effect Name
Applied Processing
Control Parameters
Processing Domain
Digital Implementation Technique
higher, lower
slower, faster
synthetic, monotone
synthetic, mutant, hybrid
whispering voice
Pitch
Time
Timbre & Sound Quality
Pitch-Shifting
Time-Scaling
Robotization
Mutation
Whisperization
Denoising
Filtering
Dispersion
Stable/transient separation
1) time-scale $N_2[n]/N_1[n]$ 2) resample $N_1[n]/N_2[n]$
$R_A[n] \neq R_S[n]$
1) $R_S[n] = F_{\text{samp}} / F_0(n)$ 2) $\phi(n,k) = 0$ at bin center 3) 0-pad between 2 grains
$Y(n,k) = f(|X_1(n,k)|,|X_1(n,k)|) . g(\phi_1(n,k),\phi_2(n,k))$
$\phi(n,k) = \text{rand}()$, or $|X(n,k)| = \text{rand}()$
$Y(n,k) = f(X(n,k))$ $f(x) = x^2 / (x+c)$
$Y(n,k) = X(n,k).H(n,k)$
$H(n,k) = e^{i\phi(k)}$
$|Y(n,k)| = |X(n,k)|$ if freq. dev. (f) $< \Delta$, 0 otherwise
modify $N_1[n]$, $N_2[n]$
modify $R_A[n]$, $R_S[n]$
modify $\phi(n,k)$ only
modify both $|X(n,k)|$ and $\phi(n,k)$
modify $|X(n,k)|$ only
Time-Frequency Domain
Phase Vocoder

Figure 7.19 Summary of time-frequency domain digital audio effects, with their relative operations and the perceptual attributes that are mainly modified.

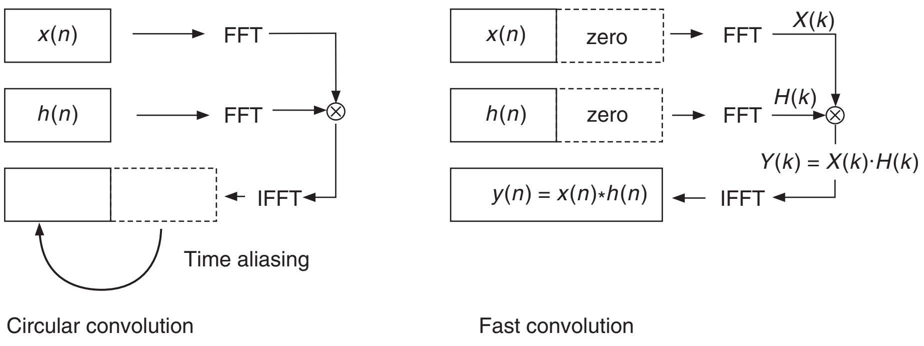


Figure 7.20 Circular convolution and fast convolution.

frequency domain by multiplication of the corresponding FFTs $Y(k) = X(k) \cdot H(k)$. This technique is called *fast convolution* (see Figure 7.20) and is performed by the following steps:

(1) Zero-pad the signal segment $x(n)$ and the impulse response $h(n)$ up to length $2N$.

(2) Take the $2N$-point FFT of these two signals.

(3) Perform multiplication $Y(k) = X(k) \cdot H(k)$ with $k = 0, 1, \ldots, 2N - 1$.

(4) Take the $2N$-point IFFT of $Y(k)$, which yields $y(n)$ with $n = 0, 1, \ldots, 2N - 1$.

Now we can work with successive segments of length N of the original signal (which is equivalent to using a rectangular window), zero-pad each segment up to the length $2N$ and perform the fast convolution with the filter impulse response. The results of each convolution are added in an overlap-add procedure, as shown in Figure 7.21. The algorithm can be summarized as:

(1) Start from an FIR filter of length $N + 1$, zero pad it to $2N$ and take its FFT $\Rightarrow H(k)$.

(2) Partition the signal into segments $x_i(n)$ of length N and zero-pad each segment up to length $2N$.

(3) For each zero-padded segment $s_i(n)$ perform the FFT $X_i(k)$ with $k = 0, 1, \ldots, 2N - 1$.

(4) Perform the multiplication $Y_i(k) = X_i(k) \cdot H(k)$.

(5) Take the inverse FFT of these products $Y_i(k)$.

(6) Overlap-add the convolution results (see Figure 7.21).

The following M-file 7.7 demonstrates the FFT filtering algorithm.

M-file 7.7 (VX_filter.m)

```
% VX_filter.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs time-frequency filtering
%===== after calculation of the fir (here band pass)

clear; clf

%----- user data -----
fig_plot      = 0;       % use any value except 0 or [] to plot figures
s_FIR         = 1280;    % length of the fir [samples]
```

```
s_win         = 2*s_FIR; % window size [samples] for zero padding
[DAFx_in, FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
L         = length(DAFx_in);
DAFx_in   = [DAFx_in;  zeros(s_win-mod(L,s_FIR),1)] ...
   / max(abs(DAFx_in));             % 0-pad & normalize
DAFx_out  = zeros(length(DAFx_in)+s_FIR,1);
grain     = zeros(s_win,1);         % input grain
vec_pad   = zeros(s_FIR,1);     % padding array

%----- initialize calculation of fir -----
x         = (1:s_FIR);
fr        = 1000/FS;
alpha     = -0.002;
fir       = (exp(alpha*x).*sin(2*pi*fr*x))'; % FIR coefficients
fir2      = [fir; zeros(s_win-s_FIR,1)];
fcorr     = fft(fir2);

%----- displays the filter' simpulse response -----
if(fig_plot)
  figure(1); clf;
  subplot(2,1,1); plot(fir); xlabel('n [samples] \rightarrow');
  ylabel('h(n) \rightarrow'); axis tight;
  title('Impulse response of the FIR')
  subplot(2,1,2);
  plot((0:s_FIR-1)/s_FIR*FS, 20*log10(abs(fft(fftshift(fir)))));
  xlabel('k \rightarrow'); ylabel('|F(n,k)| / dB \rightarrow');
  title('Magnitude spectrum of the FIR'); axis([0 s_FIR/2, -40, 50])
  pause
end

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_FIR;

while pin<pend
  grain = [DAFx_in(pin+1:pin+s_FIR); vec_pad];
%============================================
  ft    = fft(grain) .* fcorr;
  grain = (real(ifft(ft)));
%============================================
  DAFx_out(pin+1:pin+s_win) = ...
    DAFx_out(pin+1:pin+s_win) + grain;
  pin   = pin + s_FIR;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_filter.wav');
```

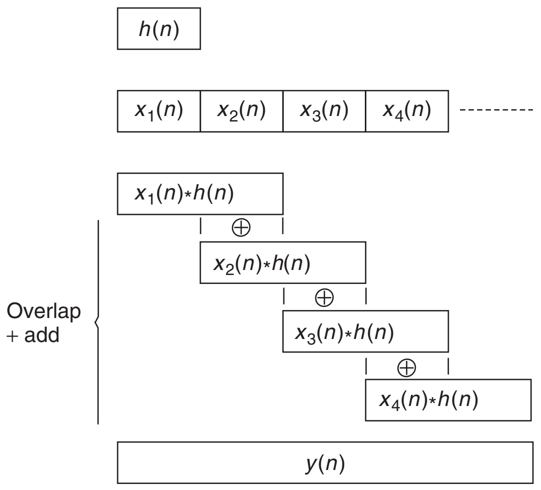


Figure 7.21 FFT filtering.

The design of an N-point FIR filter derived from frequency-domain specifications is a classical problem of signal processing. A simple design algorithm is the frequency sampling method [Zöl05].

7.4.2 Dispersion

When a sound is transmitted over telecommunications lines, some of the frequency bands are delayed. This spreads a sound in time, with some components of the signal being delayed. It is usually considered a default in telecommunications, but can be used musically. This dispersion effect is especially significant on transients, where the sound loses its coherence, but can also *blur* the steady-state parts. Actually, this is what happens with the phase vocoder when time scaling and if the phase is not locked. Each frequency bin coding the same partial has a different phase unfolding and then frequency, and the resulting phase of the synthesized partial may differ from the one in the original sound, resulting in dispersion.

A dispersion effect can be simulated by a filter, especially an FIR filter, whose frequency response has a frequency-dependent time delay. The only change to the previous program is to change the calculation of the FIR vector `fir`. We will now describe several filter designs for a dispersion effect.

Design 1 As an example, a linear chirp signal is a sine wave with linearly increasing frequency and has the property of having a time delay proportional to its frequency. A mathematical definition of a linear chirp signal starting from frequency zero and going to frequency f_1 during time t_1 is given by

$$\mathrm{Chirp}(t) = \sin(\alpha t^2) \text{ with } \alpha = \pi \frac{f_1}{t_1}. \tag{7.44}$$

Sampling of this chirp signal yields the coefficients for an FIR filter. Time-frequency representations of a linear and an exponential chirp signal are shown in Figure 7.22a.

Design 2 It is also possible to numerically approximate a chirp by integrating an arbitrary frequency function of time. In this case the **MATLAB** function `cumsum` can be used to calculate the phase $\varphi(n) = \int_0^{nT} 2\pi f(\tau)\mathrm{d}\tau + \varphi(0)$ as the integral of the time-dependent frequency $f(t)$.

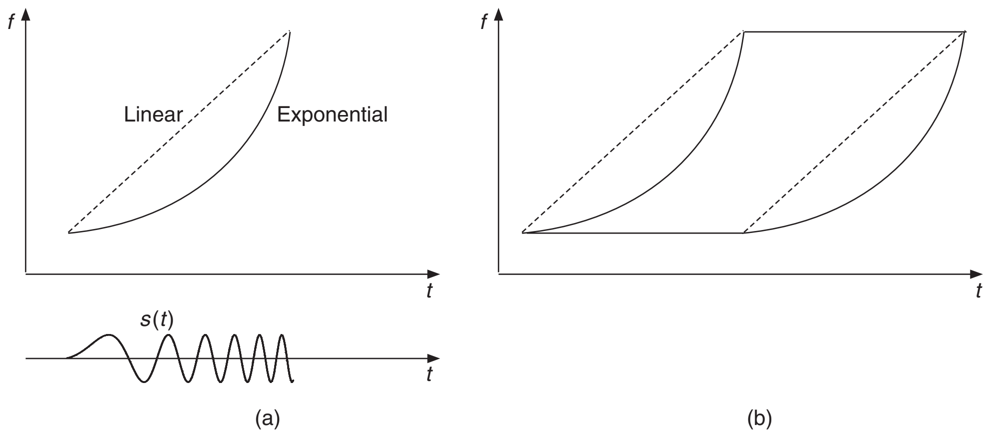


Figure 7.22 Time-frequency representations: (a) linear/exponential chirp signal and (b) time-frequency warping for the linear/exponential chirp.

A linear chirp with 300 samples can be computed by the **MATLAB** instructions:

```
n    = 300;
x    = (1:n)/n;
f0   = 50;
f1   = 4000;
freq = 2*pi * (f0+(f1-f0)*x) / 44100;
fir  = (sin(cumsum(freq)))';
```

and an exponential chirp by

```
n    = 300;
x    = (1:n)/n;
f0   = 50;
f1   = 4000;
rap  = f1/f0;
freq = (2*pi*f0/44100) * (rap.^x);
fir  = (sin(cumsum(freq)))';
```

Any other frequency function $f(t)$ can be used for the calculation of `freq`.

Design 3 Nevertheless these chirp signals deliver the frequency as a function of time delay. We would be more likely to define the time delay as a function of frequency. This is only possible with the previous technique if the function is monotonous. Thus in a more general case we can use the phase information of an FFT as an indication of the time delay corresponding to a frequency bin: the phase of a delayed signal $x(n-M)$, which has a discrete Fourier transform $X(k)\mathrm{e}^{-jM\frac{2\pi k}{N}}$ with $k = 0, 1, \ldots, N/2$, is $\varphi(k) = -M\frac{2\pi k}{N}$, where M is the delay in samples, k is the number of the frequency bin and N is the length of the FFT.

A variable delay for each frequency bin can be achieved by replacing the fixed value M (the delay of each frequency bin) by a function $M(k)$, which leads to $X(k)\mathrm{e}^{-jM(k)\frac{2\pi k}{N}}$. For example, a linearly increasing time delay for each frequency bin is given by $M(k) = M \cdot \frac{k}{N}$ with $k = 0, 1, \ldots, N/2 - 1$. The derivation of the FIR coefficients can be achieved by performing an IFFT of the positive part of the spectrum and then taking the real part of the resulting complex-valued coefficients. With this technique a linear chirp signal centered around the middle of the window can be computed by the following **MATLAB** instructions:

```
M    = 300;
WLen = 1024;
mask = [1; 2*ones(WLen/2-1,1); 1 ; zeros(WLen/2-1,1)];
fs   = M*(0:WLen/2)' / WLen; % linear increasing delay
teta = [-2*pi*fs.*(0:WLen/2)'/WLen ; zeros(WLen/2-1,1)];
f2   = exp(i*teta);
fir  = fftshift(real(ifft(f2.*mask)));
```

It should be noted that this technique can produce time aliasing. The length of the FIR filter will be greater than M. A proper choice of N is needed, for example $N > 2M$.

Design 4 A final technique is to draw an arbitrary curve on a time-frequency representation, which is an invalid image, and then resynthesize a signal by forcing a reconstruction, for example, by using a summation of gaborets. Then we can use this reconstructed signal as the impulse response of the FIR filter. If the curve displays the dispersion of a filter, we get a dispersive filter.

In conclusion, we can say that dispersion, which is a filtering operation, can be perceived as a delay operation. This leads to a warping of the time-frequency representation, where each horizontal line of this representation is delayed according to the dispersion curve (see Figure 7.22b).

7.4.3 Time stretching

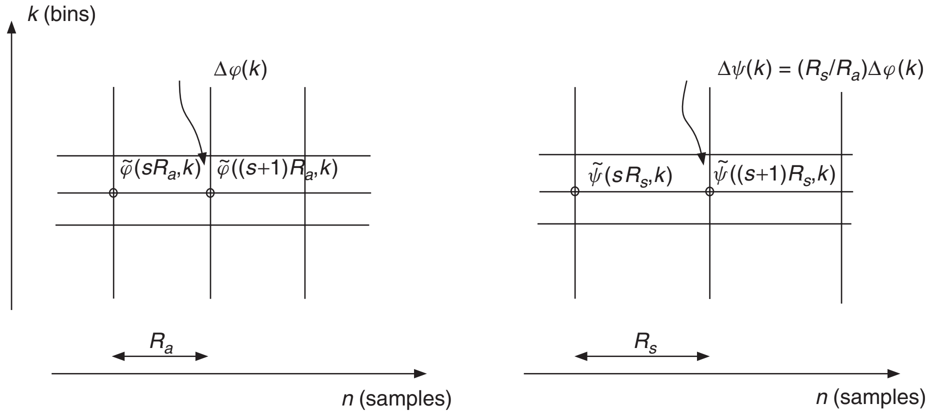
Time-frequency scaling is one of the most interesting and difficult tasks that can be assigned to time-frequency representations: changing the time scale independently of the "frequency content." For example, one can change the rhythm of a song without changing its pitch, or conversely transpose a song without any time change. Time stretching is not a problem that can be stated outside of the perception: we know, for example, that a sum of two sinusoids is equivalent to a product of a carrier and a modulator. Should time stretching of this signal still be a sum of two sinusoids or the same carrier with a lower modulation? This leads us to the perception of tremolo tones or vibrato tones. One generally agrees that tremolos and vibratos under 10 Hz are perceived as such and those over are perceived as a sum of sinusoids. Another example is the exponential decay of sounds produced by percussive gesture (percussions, plucked strings, etc.): should the time-scaled version of such a sound have a natural exponential decay? When time scaling without modeling such aspects, the decay is stretched too, so the exponential decay is modified, potentially resulting in non-physically realistic sounds. A last example is voice (sung or spoken): the perception of some consonants is modified into others when the sound is time scaled, highlighting the fact that it can be interesting to refine such models by modeling the sound content and using non-linear or adaptive processing. Right now, we explain straight time stretching with time-frequency representations and constant control parameters.

One technique has already been evaluated in the time domain (see PSOLA in Section 6.3.3). Here we will deal with another technique in the time-frequency domain using the phase vocoder implementations of Section 7.3. There are two implementations for time-frequency scaling by the "traditional" phase vocoder. Historically, the first one uses a bank of oscillators, whose amplitudes and frequencies vary over time. If we can manage to model a sound by the sum of sinusoids, time stretching and pitch shifting can be performed by expanding the amplitude and frequency functions. The second implementation uses the sliding Fourier transform as the model for resynthesis: if we can manage to spread the image of a sliding FFT over time and calculate new phases, then we can reconstruct a new sound with the help of inverse FFTs. Both of these techniques rely on phase interpolation, which need an unwrapping algorithm at the analysis stage, or equivalently an instantaneous frequency calculation, as introduced in Section 7.3.5.

The time-stretching algorithm mainly consists of providing a synthesis grid which is different from the analysis grid, and to find a way to reconstruct a signal from the values on this grid. Though it is possible to use any stretching factor, we will here only deal with the case where we use an integer both for the analysis hop size R_a, and for the synthesis hop size R_s.

As seen in Section 7.3, changing the values and their coordinates on a time-frequency representation is generally not a valid operation, in the sense that the resulting representation is not the sliding Fourier transform of a real signal. However, it is always possible to force the reconstruction of a sound from an arbitrary image, but the time-frequency representation of the signal issued from this forced synthesis will be different from what was expected. The goal of a good transformation algorithm is to find a strategy that preserves the time-stretching aspect without introducing too many artifacts.

The classical way of using a phase vocoder for time stretching is to keep the magnitude unchanged and to modify the phase in such a way that the instantaneous frequencies are preserved. Providing that the grid is enlarged from an analysis hop size R_a to a synthesis hop size R_s, this means that the new phase values must satisfy $\Delta\psi(k) = \frac{R_s}{R_a}\Delta\varphi(k)$ (see Figure 7.23). Once the grid is filled with these values one can reconstruct a signal using either the filter-bank approach or the block-by-block IFFT approach.



k (bins)
$\Delta\varphi(k)$
$\tilde{\varphi}(sR_a,k)$
$\tilde{\varphi}((s+1)R_a,k)$
R_a
n (samples)
$\Delta\psi(k) = (R_s/R_a)\Delta\varphi(k)$
$\tilde{\psi}(sR_s,k)$
$\tilde{\psi}((s+1)R_s,k)$
R_s
n (samples)

Figure 7.23 Time-stretching principle: analysis with hop size R_a gives the time-frequency grid shown in the left part, where $\Delta\varphi(k) = \tilde{\varphi}((s+1)R_a, k) - \tilde{\varphi}(sR_a, k)$ denotes the phase difference between the unwrapped phases. The synthesis is performed from the modified time-frequency grid with hop size R_s and the phase difference $\Delta\psi(k) = \tilde{\psi}((s+1)R_s, k) - \tilde{\psi}(sR_s, k)$, which is illustrated in the right part.

Filter-bank approach (sum of sinusoids)

In the FFT analysis/sum of sinusoids synthesis approach, we calculate the instantaneous frequency for each bin and integrate the corresponding phase increment in order to reconstruct a signal as the weighted sum of cosines of the phases. However, here the hop size for the resynthesis is different from the analysis. Therefore the following steps are necessary:

(1) Calculate the phase increment per sample by $d\psi(k) = \Delta\varphi(k)/R_a$.

(2) For the output samples of the resynthesis integrate this value according to $\tilde{\psi}(n+1, k) = \tilde{\psi}(n, k) + d\psi(k)$.

(3) Sum the intermediate signals which yields $y(n) = \sum_{k=0}^{N/2} A(n,k)\cos(\tilde{\psi}(n,k))$ (see Figure 7.24).

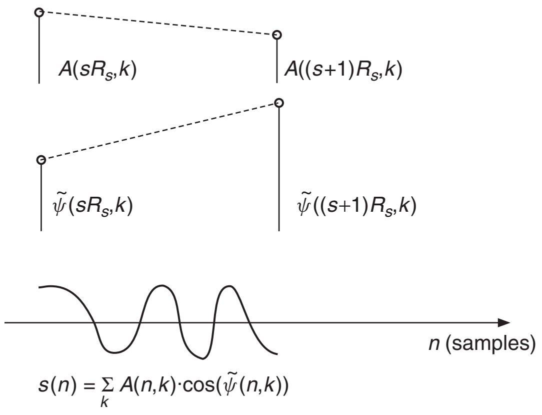

$A(sR_s,k)$
$A((s+1)R_s,k)$
$\tilde{\psi}(sR_s,k)$
$\tilde{\psi}((s+1)R_s,k)$
n (samples)
$s(n) = \sum_k A(n,k)\cdot\cos(\tilde{\psi}(n,k))$

Figure 7.24 Calculation of time-frequency samples. Given the values of A and $\tilde{\psi}$ on the representation grid, we can perform linear interpolation with a hop size between two successive values on the grid. The reconstruction is achieved by a summation of weighted cosines.

A complete **MATLAB** program for time stretching is given by M-file 7.8.

M-file 7.8 (VX_tstretch_bank.m)

```
% VX_tstretch_bank.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs time stretching
%===== using the oscillator bank approach
clear; clf
%----- user data -----
n1            = 256;  % analysis step increment [samples]
n2            = 512;  % synthesis step increment [samples]
s_win         = 2048; % analysis window length [samples]
[DAFx_in, FS]  = wavread('la.wav');
%----- initialize windows, arrays, etc -----
tstretch_ratio = n2/n1
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;         % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in)); % 0-pad & normalize
DAFx_out = zeros(s_win+ceil(length(DAFx_in)*tstretch_ratio),1);
grain    = zeros(s_win,1);
ll       = s_win/2;
omega    = 2*pi*n1*[0:ll-1]'/s_win;
phi0     = zeros(ll,1);
r0       = zeros(ll,1);
psi      = zeros(ll,1);
res      = zeros(n2,1);
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
```

```
pend = length(DAFx_in)-s_win;
while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%============================================
  fc  = fft(fftshift(grain));
  f   = fc(1:ll);
  r   = abs(f);
  phi = angle(f);
  %----- calculate phase increment per block -----
  delta_phi = omega + princarg(phi-phi0-omega);
  %----- calculate phase & mag increments per sample -----
  delta_r   = (r-r0) / n2;    % for synthesis
  delta_psi = delta_phi / n1; % derived from analysis
  %----- computing output samples for current block -----
  for k=1:n2
    r0  = r0 + delta_r;
    psi = psi + delta_psi;
    res(k) = r0'*cos(psi);
  end
  %----- values for processing next block -----
  phi0 = phi;
  r0   = r;
  psi  = princarg(psi);
% ============================================
%  DAFx_out(pout+1:pout+n2) = DAFx_out(pout+1:pout+n2)+res;
  DAFx_out(pout+1:pout+n2) = res;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
%----- listening and saving the output -----
% DAFx_in = DAFx_in(s_win+1:s_win+L);
DAFx_out=DAFx_out(s_win/2+n1+1:length(DAFx_out))/max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_tstretch_bank.wav');
```

This program first extracts a series of sound segments called grains. For each grain the FFT is computed to yield a magnitude and phase representation every `n1` samples (`n1` is the analysis hop size R_a). It then calculates a sequence of `n2` samples (`n2` is the synthesis hop size R_s) of the output signal by interpolating the values of `r` and calculating the phase `psi` in such a way that the instantaneous frequency derived from `psi` is equal to the one derived from `phi`. The unwrapping of the phase is then done by calculating `(phi-phi0-omega)`, putting it in the range $[-\pi, \pi]$ and again adding `omega`. A phase increment per sample `d_psi` is calculated from `delta_phi/n1`. The calculation of the magnitude and phase at the resynthesis is done in the loop `for k=1:n2` where `r` and `psi` are incremented by `d_r` and `d_psi`. The program uses the vector facility of **MATLAB** to calculate the sum of the cosine of the angles weighted by magnitude in one step. This gives a buffer `res` of `n2` output samples which will be inserted into the `DAFx_out` signal.

Block-by-block approach (FFT/IFFT)

Here we follow the FFT/IFFT implementation used in Section 7.3, but the hop size for resynthesis is different from the analysis. So we have to calculate new phase values in order to preserve the instantaneous frequencies for each bin. This is again done by calculating an unwrapped phase difference for each frequency bin, which is proportional to $\frac{R_s}{R_a}$. We also have to take care of some implementation details, such as the fact that the period of the window has to be equal to the length of the FFT (this is not the case for the standard **MATLAB** functions). The synthesis hop size should at least allow a minimal overlap of windows, or should be a submultiple of it. It is suggested to use overlap values of at least 75% (i.e., $R_a \leq n_1/4$). The following M-file 7.9 demonstrates the block-by-block FFT/IFFT implementation.

M-file 7.9 (VX_tstretch_real_pv.m)

```
% VX_tstretch_real_pv.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs time stretching
%===== using the FFT-IFFT approach, for real ratios
clear; clf

%----- user data -----
n1           = 200;    % analysis step [samples]
n2           = 512;    % synthesis step ([samples]
s_win        = 2048;   % analysis window length [samples]
[DAFx_in,FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
tstretch_ratio = n2/n1
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(s_win+ceil(length(DAFx_in)*tstretch_ratio),1);
omega    = 2*pi*n1*[0:s_win-1]'/s_win;
phi0     = zeros(s_win,1);
psi      = zeros(s_win,1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in)-s_win;
while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
  f     = fft(fftshift(grain));
  r     = abs(f);
  phi   = angle(f);
  %---- computing input phase increment ----
  delta_phi = omega + princarg(phi-phi0-omega);
  %---- computing output phase increment ----
  psi   = princarg(psi+delta_phi*tstretch_ratio);
  %---- comouting synthesis Fourier transform & grain ----
  ft    = (r.* exp(i*psi));
  grain = fftshift(real(ifft(ft))).*w2;
  % plot(grain);drawnow;
% ===========================================
  DAFx_out(pout+1:pout+s_win) = ...
     DAFx_out(pout+1:pout+s_win) + grain;
  %----- for next block -----
  phi0 = phi;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
%DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:length(DAFx_out))/max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_tstretch_noint_pv.wav');
```

This program is much faster than the preceding one. It extracts grains of the input signal by windowing the original signal `DAFx_in`, makes a transformation of these grains and overlap-adds these transformed grains to get a sound `DAFx_out`. The transformation consists of performing the FFT of the grain and computing the magnitude and phase representation `r` and `phi`. The unwrapping of the phase is then done by calculating `(phi-phi0-omega)`, putting it in the range $[-\pi, \pi]$ and again adding `omega`. The calculation of the phase `psi` of the transformed grain is then achieved by adding the phase increment `delta_phi` multiplied by the stretching factor `ral` to the previous unwrapped phase value. As seen before, this is equivalent to keeping the same instantaneous frequency for the synthesis as is calculated for the analysis. The new output grain is then calculated by an inverse FFT, windowed again and overlap-added to the output signal.

Hints and drawbacks As we have noticed, phase vocoding can produce artifacts. It is important to know them in order to face them.

(1) Changing the phases before the IFFT is equivalent to using an all pass filter whose Fourier transform contains the phase correction that is being applied. If we do not use a window for the resynthesis, we can ensure the circular convolution aspect of this filtering operation. We will have discontinuities at the edges of the signal buffer. So it is necessary to use a synthesis window.

(2) Nevertheless, even with a resynthesis window (also called *tapering window*) the circular aspect still remains: the result is the aliased version of an infinite IFFT. A way to counteract this is to choose a zero-padded window for analysis and synthesis.

(3) Shape of the window: one must ensure that a perfect reconstruction is given with a ratio $\frac{R_s}{R_a}$ equal to one (no time stretching). If we use the same window for analysis and synthesis, the sum of the square of the windows, regularly spaced at the resynthesis hope size, should be one.

(4) For a Hanning window without zero-padding the hop size R_s has to be a divisor of $N/4$.

(5) Hamming and Blackman windows provide smaller side lobes in the Fourier transform. However, they have the inconvenience of being non-zero at the edges, so that no tapering is done by using these windows alone. The resynthesis hop size should be a divisor of $N/8$.

(6) Truncated Gaussian windows, which are good candidates, provide a sum that always has oscillations, but which can be below the level of perception.

Phase dispersion An important problem is the difference of phase unwrapping between different bins, which is not solved by the algorithms we presented: the unwrapping algorithm of the analysis gives a phase that is equal to the measured phase modulo 2π. So the unwrapped phase is equal to the measured phase plus a term that is a multiple of 2π. This second term is not the same for every bin. Because of the multiplication by the time-stretching ratio, there is a dispersion of the phases. One cannot even ensure that two identical successive sounds will be treated in the same way. This is in fact the main drawback of the phase vocoder and its removal is treated in several publications [QM98, Fer99, LD99a, Lar03, Röb03, Röb10].

Phase-locked vocoder

One of the most successful approaches to reduce the phase dispersion was proposed in [LD99a]. If we consider the processed sound to be mostly composed of quasi-sinusoidal components, then we can approximate its spectrum as the sum of the complex convolution of each of those components by the analysis window transform (this will be further explained in the spectral processing chapter). When we transform the sound, for instance time stretching it, the phase of those quasi-sinusoidal components has to propagate accordingly. What is really interesting here is that for each sinusoid the effect of the phase propagation on the spectrum is nothing more than a constant phase rotation of all the spectral bins affected by it. This method is referred to as phase-locked vocoder, since the phase of each spectral bin is locked to the phase of one spectral peak.

Starting from the previous M-file, we need to add the following steps to the processing loop:

(1) Find spectral peaks. A good tradeoff is to find local maxima in a predefined frequency range, for instance considering two bins around each candidate. This helps to minimize spurious peaks as well as to reduce the likelihood of identifying analysis window transform side-lobes as spectral peaks.

(2) Connect current peaks to previous frame peaks. The simplest approach is to choose the closest peak in frequency.

(3) Propagate peaks. A simple but usually effective strategy is to consider that peaks evolve linearly in frequency.

(4) Rotate equally all bins assigned to each spectral peak. The assignment can be performed by segmenting the spectrum into frequency regions delimited by the middle bin between consecutive peaks. An alternative is to set the segment boundaries to the bin with minimum amplitude between consecutive peaks.

Furthermore, the code can be optimized taking into account that the spectrum of a real signal is hermitic, so that we do only need to process the first half of the spectrum. A complete **MATLAB** program for time stretching using the phase-locked vocoder is given by M-file 7.10.

M-file 7.10 (VX_tstretch_real_pv_phaselocked.m)

```
% VX_tstretch_real_pv_phaselocked.m   [DAFXbook, 2nd ed., chapter 7]
%===== this program performs real ratio time stretching using the
%===== FFT-IFFT approach, applying spectral peak phase-locking
clear; clf

%----- user data -----
n1          = 256;    % analysis step [samples]
n2          = 300;    % synthesis step ([samples]
s_win       = 2048;   % analysis window length [samples]
[DAFx_in,FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
tstretch_ratio = n2/n1
hs_win     = s_win/2;
w1         = hanning(s_win, 'periodic'); % analysis window
```

```
w2         = w1;    % synthesis window
L          = length(DAFx_in);
DAFx_in    = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out   = zeros(s_win+ceil(length(DAFx_in)*tstretch_ratio),1);
omega      = 2*pi*n1*[0:hs_win]'/s_win;
phi0       = zeros(hs_win+1,1);
psi        = zeros(hs_win+1,1);
psi2       = psi;
nprevpeaks = 0;

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;
while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
  %===========================================
  f     = fft(fftshift(grain));
  %---- optimization: only process the first 1/2 of the spectrum
  f     = f(1:hs_win+1);
  r     = abs(f);
  phi   = angle(f);
  %---- find spectral peaks (local maxima) ----
  peak_loc = zeros(hs_win+1,1);
  npeaks = 0;
  for b=3:hs_win-1
    if ( r(b)>r(b-1) && r(b)>r(b-2) && r(b)>r(b+1) && r(b)>r(b+2) )
      npeaks = npeaks+1;
      peak_loc(npeaks) = b;
      b = b + 3;
    end
  end
  %---- propagate peak phases and compute spectral bin phases
  if (pin==0) % init
    psi = phi;
  elseif (npeaks>0 && nprevpeaks>0)
    prev_p = 1;
    for p=1:npeaks
      p2 = peak_loc(p);
      %---- connect current peak to the previous closest peak
      while (prev_p < nprevpeaks && abs(p2-prev_peak_loc(prev_p+1)) ...
          < abs(p2-prev_peak_loc(prev_p)))
        prev_p = prev_p+1;
      end
      p1 = prev_peak_loc(prev_p);
      %---- propagate peak's phase assuming linear frequency
      %---- variation between connected peaks p1 and p2
      avg_p  = (p1 + p2)*.5;
      pomega = 2*pi*n1*(avg_p-1.)/s_win;
      % N.B.: avg_p is a 1-based indexing spectral bin
      peak_delta_phi = pomega + princarg(phi(p2)-phi0(p1)-pomega);
      peak_target_phase = princarg(psi(p1) + peak_delta_phi*tstretch_ratio);
      peak_phase_rotation = princarg(peak_target_phase-phi(p2));
      %---- rotate phases of all bins around the current peak
      if (npeaks==1)
```

```
        bin1 = 1; bin2 = hs_win+1;
      elseif (p==1)
        bin1 = 1; bin2 = hs_win+1;
      elseif (p==npeaks)
        bin1 = round((peak_loc(p-1)+p2)*.5);
        bin2 = hs_win+1;
      else
        bin1 = round((peak_loc(p-1)+p2)*.5)+1;
        bin2 = round((peak_loc(p+1)+p2)*.5);
      end
      psi2(bin1:bin2) = princarg(phi(bin1:bin2) + peak_phase_rotation);
    end
    psi = psi2;
  else
    delta_phi = omega + princarg(phi-phi0-omega);
    psi       = princarg(psi+delta_phi*tstretch_ratio);
  end

  ft    = (r.* exp(i*psi));
  %---- reconstruct whole spectrum (it is hermitic!)
  ft    = [ ft(1:hs_win+1) ; conj(ft(hs_win:-1:2)) ];
  grain = fftshift(real(ifft(ft))).*w2;
  % plot(grain);drawnow;
  % ========================================
  DAFx_out(pout+1:pout+s_win) = ...
    DAFx_out(pout+1:pout+s_win) + grain;
  %---- store values for next frame ----
  phi0  = phi;
  prev_peak_loc = peak_loc;
  nprevpeaks    = npeaks;
  pin   = pin + n1;
  pout  = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
%DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:length(DAFx_out))/max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_tstretch_noint_pv_phaselocked.wav');
```

Integer ratio time stretching

When the time-stretching ratio is an integer (e.g., time stretching by 200%, 300%), the unwrapping is no longer necessary in the algorithm, because the 2π modulo relation is still preserved when the phase is multiplied by an integer. The key point here is that we can make a direct multiplication of the analysis phase to get the phase for synthesis. So in this case it is more obvious and elegant to use the following algorithm, given by M-file 7.11.

M-file 7.11 (VX_tstretch_int_pv.m)

```
% VX_tstretch_int_pv.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs integer ratio time stretching
```

```
%===== using the FFT-IFFT approach
clear; clf

%----- user data -----
n1           = 64;     % analysis step [samples]
n2           = 512;    % synthesis step ([samples]
s_win        = 2048;   % analysis window length [samples]
[DAFx_in,FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
tstretch_ratio = n2/n1
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(s_win+ceil(length(DAFx_in)*tstretch_ratio),1);
omega    = 2*pi*n1*[0:s_win-1]'/s_win;
phi0     = zeros(s_win,1);
psi      = zeros(s_win,1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in)-s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%============================================
  f     = fft(fftshift(grain));
  r     = abs(f);
  phi   = angle(f);
  ft    = (r.* exp(i*tstretch_ratio*phi));
  grain = fftshift(real(ifft(ft))).*w2;
% ===========================================
  DAFx_out(pout+1:pout+s_win) = ...
    DAFx_out(pout+1:pout+s_win) + grain;
  pin   = pin + n1;
  pout  = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:length(DAFx_out))/max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_stretch_int_pv.wav');
```

7.4.4 Pitch shifting

Pitch shifting is different from frequency shifting: a frequency shift is an addition to every frequency (i.e., the magnitude spectrum is shifted), while pitch shifting is the multiplication of every frequency by a transposition factor (i.e., the magnitude spectrum is scaled). Pitch shifting can be directly linked

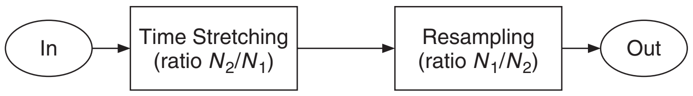

In
Time Stretching (ratio N_2/N_1)
Resampling (ratio N_1/N_2)
Out

Figure 7.25 Resampling of a time-stretching algorithm.

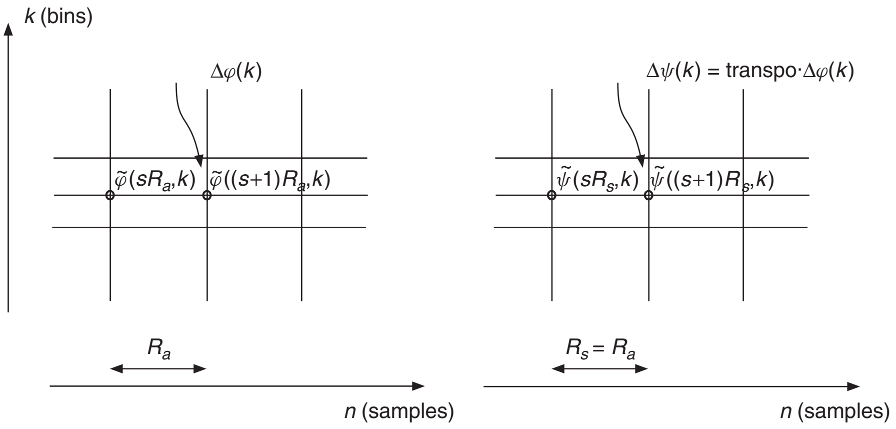

k (bins)
$\Delta\varphi(k)$
$\tilde{\varphi}(sR_a,k)$
$\tilde{\varphi}((s+1)R_a,k)$
R_a
n (samples)
$\Delta\psi(k)$ = transpo·$\Delta\varphi(k)$
$\tilde{\psi}(sR_s,k)$
$\tilde{\psi}((s+1)R_s,k)$
$R_s = R_a$
n (samples)

Figure 7.26 Pitch shifting with the filter-bank approach: the analysis gives the time-frequency grid with analysis hop size R_a. For the synthesis the hop size is set to $R_s = R_a$ and the phase difference is calculated according to $\Delta\psi(k) =$ `transpo` $\Delta\varphi(k)$.

to time stretching. Resampling a time-stretched signal with the inverse of the time-stretching ratio performs pitch shifting and going back to the initial duration of the signal (see Figure 7.25). There are, however, alternative solutions which allow the direct calculation of a pitch-shifted version of a sound.

Filter-bank approach (sum of sinusoids)

In the time-stretching algorithm using the sum of sinusoids (see Section 7.3) we have an evaluation of instantaneous frequencies. As a matter of fact transposing all the instantaneous frequencies can lead to an efficient pitch-shifting algorithm. Therefore the following steps have to be performed (see Figure 7.26):

(1) Calculate the phase increment per sample by $d\varphi(k) = \Delta\varphi(k)/R_a$.

(2) Multiply the phase increment by the transposition factor `transpo` and integrate the modified phase increment according to $\tilde{\psi}(n+1,k) = \tilde{\psi}(n,k)+$ `transpo` $\cdot\Delta\varphi(k)/R_a$.

(3) Calculate the sum of sinusoids: when the transposition factor is greater than one, keep only frequencies under the Nyquist frequency bin $N/2$. This can be done by taking only the `N/(2*transpo)` frequency bins.

The following M-file 7.12 is similar to the program given by M-file 7.8 with the exception of a few lines: the definition of the hop size and the resynthesis phase increment have been changed.

M-file 7.12 (VX_pitch_bank.m)

```
% VX_pitch_bank.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs pitch shifting
%===== using the oscillator bank approach
clear; clf

%----- user data -----
n1           = 512;    % analysis step [samples]
pit_ratio    = 1.2     % pitch-shifting ratio
s_win        = 2048;   % analysis window length [samples]
[DAFx_in,FS] = wavread('la.wav');

%----- initialize windows, arrays, etc -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1);
grain    = zeros(s_win,1);
hs_win   = s_win/2;
omega    = 2*pi*n1*[0:hs_win-1]'/s_win;
phi0     = zeros(hs_win,1);
r0       = zeros(hs_win,1);
psi      = phi0;
res      = zeros(n1,1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in)-s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%============================================
  fc  = fft(fftshift(grain));
  f   = fc(1:hs_win);
  r   = abs(f);
  phi = angle(f);
  %---- compute phase & mangitude increments ----
  delta_phi = omega + princarg(phi-phi0-omega);
  delta_r   = (r-r0)/n1;
  delta_psi = pit_ratio*delta_phi/n1;
  %---- compute output buffer ----
  for k=1:n1
    r0      = r0 + delta_r;
    psi     = psi + delta_psi;
    res(k)  = r0' * cos(psi);
  end
  %---- store for next block ----
  phi0 = phi;
  r0   = r;
  psi  = princarg(psi);
%  plot(res);pause;
% ============================================
  DAFx_out(pout+1:pout+n1) = DAFx_out(pout+1:pout+n1) + res;
  pin  = pin + n1;
  pout = pout + n1;
  end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(hs_win+n1+1:hs_win+n1+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'la_pitch_bank.wav');
```

The program is derived from the time-stretching program using the oscillator-bank approach in a straightforward way: this time the hop size for analysis and synthesis are the same, and a pitch transpose argument `pit` must be defined. This argument will be multiplied by the phase increment `delta_phi/n1` derived from the analysis to get the phase increment `d_psi` in the calculation loop. This means of course that we consider the pitch transposition as fixed in this program, but easy changes may be done to make it vary with time.

Block-by-block approach (FFT/IFFT)

The regular way to deal with pitch shifting using this technique is first to resample the whole output once computed, but this can alternatively be done by resampling the result of every IFFT and overlapping with a hop size equal to the analysis one (see Figure 7.27). Providing that R_s is a divider of N (FFT length), which is quite a natural way for time stretching (to ensure that the sum of the square of windows is equal to one), one can resample each IFFT result to a length of $N\frac{R_a}{R_s}$ and overlap with a hop size of R_a. Another method of resampling is to use the property of the inverse FFT: if $R_a < R_s$, we can take an IFFT of length $N\frac{R_a}{R_s}$ by taking only the first bins of the initial FFT. If $R_a > R_s$, we can zero pad the FFT, before the IFFT is performed. In each of these cases the result is a resampled grain of length $N\frac{R_a}{R_s}$.

The following M-file 7.13 implements pitch shifting with integrated resampling according to Figure 7.27. The M-file is similar to the program given by M-file 7.11, except for the definition of the hop sizes and the calculation for the interpolation.

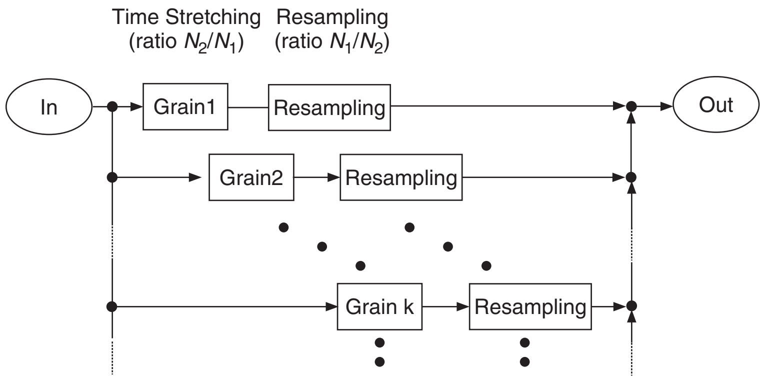

Time Stretching (ratio N_2/N_1)
Resampling (ratio N_1/N_2)
In
Grain1
Resampling
Out
Grain2
Resampling
Grain k
Resampling

Figure 7.27 Pitch shifting with integrated resampling: for each grain time stretching and resampling are performed. An overlap-add procedure delivers the output signal.

M-file 7.13 (VX_pitch_pv.m)

```
% VX_pitch_pv.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program performs pitch shifting
%===== using the FFT/IFFT approach
clear; clf

%----- user data -----
n2           = 512;    % synthesis step [samples]
pit_ratio    = 1.2     % pitch-shifting ratio
s_win        = 2048;   % analysis window length [samples]
[DAFx_in,FS] = wavread('flute2');

%----- initialize windows, arrays, etc -----
n1       = round(n2 / pit_ratio);      % analysis step [samples]
tstretch_ratio = n2/n1;
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1);
omega    = 2*pi*n1*[0:hs_win-1]'/s_win;
phi0     = zeros(s_win,1);
psi      = zeros(s_win,1);

%----- for linear interpolation of a grain of length s_win -----
lx   = floor(s_win*n1/n2);
x    = 1 + (0:lx-1)'*s_win/lx;
ix   = floor(x);
ix1  = ix + 1;
dx   = x - ix;
dx1  = 1 - dx;


tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in)-s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
  f     = fft(fftshift(grain));
  r     = abs(f);
  phi   = angle(f);
  %---- computing phase increment ----
  delta_phi = omega + princarg(phi-phi0-omega);
  phi0  = phi;
  psi   = princarg(psi+delta_phi*tstretch_ratio);
  %---- synthesizing time scaled grain ----
  ft    = (r.* exp(i*psi));
  grain = fftshift(real(ifft(ft))).*w2;
  %----- interpolating grain -----
  grain2 = [grain;0];
  grain3 = grain2(ix).*dx1+grain2(ix1).*dx;
```

```
%  plot(grain);drawnow;
% ===========================================
  DAFx_out(pout+1:pout+lx) = DAFx_out(pout+1:pout+lx) + grain3;
  pin    = pin + n1;
  pout   = pout + n1;
  end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'flute2_pitch_pv.wav');
```

This program is adapted from the time-stretching program using the FFT/IFFT approach. Here the grain is linearly interpolated before the reconstruction. The length of the interpolated grain is now `lx` and will be overlapped and added with a hop size of `n1` identical to the analysis hop size. In order to speed up the calculation of the interpolation, four vectors of length `lx` are precalculated outside the main loop, which give the necessary parameters for the interpolation (`ix`, `ix1`, `dx` and `dx1`). As stated previously, the linear interpolation is not necessarily the best one, and will surely produce some foldover when the pitch-shifting factor is greater than one. Other interpolation schemes can be inserted instead. Further pitch-shifting techniques can be found in [QM98, Lar98, LD99b].

7.4.5 Stable/transient components separation

This effect extracts "stable components" from a signal by selecting only points of the time-frequency representation that are considered as "stable in frequency" and eliminating all the other grains. Basic ideas can be found in [SL94]. From a musical point of view, one would think about getting only sine waves, and leave aside all the transient signals. However, this is not so: even with pure noise, the time-frequency analysis reveals some zones where we can have stable components. A pulse will also give an analysis where the instantaneous frequencies are the ones of the analyzing system and are very stable. Nevertheless this idea of separating a sound into two complementary sounds is indeed a musically good one. The result can be thought of as an "etherization" of the sound for the stable one, and a "fractalization" for the transient one.

The algorithm for components separation is based on instantaneous frequency computation. The increment of the phase per sample for frequency bin k can be derived as

$$d\varphi(sR_a, k) = [\tilde{\varphi}(sR_a, k) - \tilde{\varphi}((s-1)R_a, k)]/R_a. \tag{7.45}$$

We will now sort out those points of a given FFT that give

$$d\varphi(sR_a, k) - d\varphi((s-1)R_a, k) < df, \tag{7.46}$$

where df is a preset value. From (7.45) and (7.46) we can derive the condition

$$\tilde{\varphi}(sR_a, k) - 2\tilde{\varphi}((s-1)R_a, k) + \tilde{\varphi}((s-2)R_a, k) < df\,R_a. \tag{7.47}$$

From a geometrical point of view we can say that the value $\tilde{\varphi}(sR_a, k)$ should be in an angle $df\,R_a$ around the expected target value $\tilde{\varphi}_{\mathrm{t}}(sR_a, k)$, as shown in Figure 7.28.

It is important to note that the instantaneous frequencies may be out of the range of frequencies of the bin itself. The reconstruction performed by the inverse FFT takes only bins that follow this

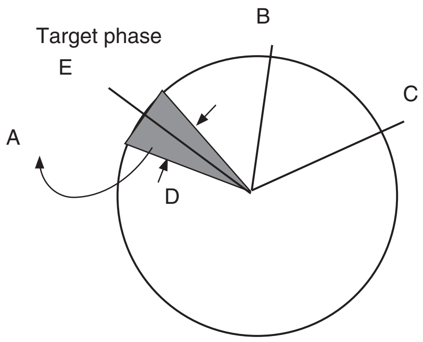


Figure 7.28 Evaluation of stable/unstable grains.

condition. In other words, only gaborets that follow the "frequency stability over time" condition are kept during the reconstruction. The following M-file 7.14 follows this guideline.

M-file 7.14 (VX_stable.m)

```
% VX_stable.m   [DAFXbook, 2nd ed., chapter 7]
%===== this program extracts the stable components of a signal
clear; clf

%----- user data -----
test         = 0.4
n1           = 256;    % analysis step [samples]
n2           = n1;     % synthesis step [samples]
s_win        = 2048;   % analysis window length [samples]
[DAFx_in,FS] = wavread('redwheel.wav');

%----- initialize windows, arrays, etc -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1);
devcent  = 2*pi*n1/s_win;
vtest    = test * devcent
grain    = zeros(s_win,1);
theta1   = zeros(s_win,1);
theta2   = zeros(s_win,1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in)-s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%============================================
  f      = fft(fftshift(grain));
  theta  = angle(f);
  dev    = princarg(theta - 2*theta1 + theta2);
%  plot(dev);drawnow;
  %---- set to 0 magnitude values below 'test' threshold
  ft     = f.*(abs(dev) < vtest);
  grain  = fftshift(real(ifft(ft))).*w2;
  theta2 = theta1;
  theta1 = theta;
% ============================================
  DAFx_out(pout+1:pout+s_win) = ...
    DAFx_out(pout+1:pout+s_win) + grain;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'redwheel_stable.wav');
```

So the algorithm for extraction of stable components performs the following steps:

(1) Calculate the instantaneous frequency by making the derivative of the phase along the time axis.

(2) Check if this frequency is within its "stable range."

(3) Use the frequency bin or not for the reconstruction.

The value of `vtest` is particularly important because it determines the level of the selection between stable and unstable bins.

The algorithm for transient components extraction is the same, except that we keep only bins where the condition (7.47) is not satisfied. So only two lines have to be changed according to

```
test = 2                 % new value for test threshold
...
ft = f*(abs(dev)>vtest); % new condition
```

In order to enhance the unstable grains the value `vtest` is usually higher for the transient extraction.

7.4.6 Mutation between two sounds

The idea is to calculate an arbitrary time-frequency representation from two original sounds and to reconstruct a sound from it. Some of these spectral mutations (see Figure 7.29) give a flavor of cross-synthesis and morphing, a subject that will be discussed later, but are different from it, because here the effect is only incidental, while in cross-synthesis hybridization of sounds is the primary objective. Further ideas can be found in [PE96]. There are different ways to calculate a new combined magnitude and phase diagram from the values of the original ones. As stated in Section 7.3, an arbitrary image is not valid in the sense that it is not the time-frequency representation of a sound, which means that the result will be musically biased by the resynthesis scheme that we must use. Usually phases and magnitudes are calculated in an independent way,

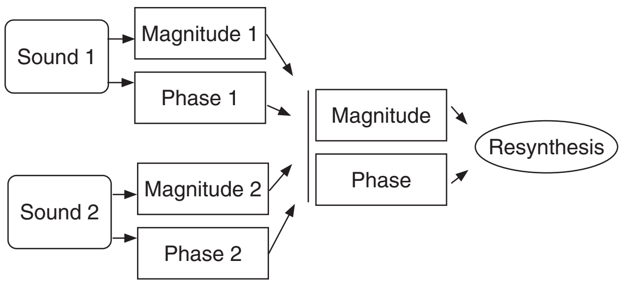


Figure 7.29 Basic principle of spectral mutations.

so that many combinations are possible. Not all of them are musically relevant, and the result also depends upon the nature of the sounds that are combined.

The following M-file 7.15 performs a mutation between two sounds where the magnitude is coming from one sound and the phase from the other. Then only a few lines need to be changed to give different variations.

M-file 7.15 (VX_mutation.m)

```
% VX_mutation.m  [DAFXbook, 2nd ed., chapter 7]
%===== this program performs a mutation between two sounds,
%===== taking the phase of the first one and the modulus
%===== of the second one, and using:
%===== w1 and w2 windows (analysis and synthesis)
%===== WLen is the length of the windows
%===== n1 and n2: steps (in samples) for the analysis and synthesis

clear; clf

%----- user data -----
n1            = 512;
n2            = n1;
WLen          = 2048;
w1            = hanningz(WLen);
w2            = w1;
[DAFx_in1,FS] = wavread('x1.wav');
DAFx_in2      = wavread('x2.wav');

%----- initializations -----
L             = min(length(DAFx_in1),length(DAFx_in2));
DAFx_in1      = [zeros(WLen, 1); DAFx_in1; ...
   zeros(WLen-mod(L,n1),1)] / max(abs(DAFx_in1));
DAFx_in2      = [zeros(WLen, 1); DAFx_in2; ...
   zeros(WLen-mod(L,n1),1)] / max(abs(DAFx_in2));
DAFx_out      = zeros(length(DAFx_in1),1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in1) - WLen;

while pin<pend
```

```
  grain1 = DAFx_in1(pin+1:pin+WLen).* w1;
  grain2 = DAFx_in2(pin+1:pin+WLen).* w1;
%===========================================
  f1     = fft(fftshift(grain1));
  r1     = abs(f1);
  theta1 = angle(f1);
  f2     = fft(fftshift(grain2));
  r2     = abs(f2);
  theta2 = angle(f2);
  %----- the next two lines can be changed according to the effect
  r      = r1;
  theta  = theta2;
  ft     = (r.* exp(i*theta));
  grain  = fftshift(real(ifft(ft))).*w2;
% ===========================================
  DAFx_out(pout+1:pout+WLen) = ...
    DAFx_out(pout+1:pout+WLen) + grain;
  pin    = pin + n1;
  pout   = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
%DAFx_in = DAFx_in(WLen+1:WLen+L);
DAFx_out = DAFx_out(WLen+1:WLen+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'r1p2.wav');
```

Possible operations on the magnitude are:

(1) Multiplication of the magnitudes `r=r1.*r2` (so it is an addition in the dB scale). This corresponds to a logical “AND” operation, because one keeps all zones where energy is located.

(2) Addition of the magnitude: the equivalent of a logical “OR” operation. However, this is different from mixing, because one only operates on the magnitude according to `r=r1+r2`.

(3) Masking of one sound by the other is performed by keeping the magnitude of one sound if the other magnitude is under a fixed or relative threshold.

Operations on phase are really important for combinations of two sounds. Phase information is very important to ensure the validity (or quasivalidity) of time-frequency representations, and has an influence on the quality:

(1) One can keep the phase from only one sound while changing the magnitude. This is a strong cue for the pitch of the resulting sound (`theta=theta2`).

(2) One can add the two phases. In this case we strongly alter the validity of the image (the phase turns with a mean double speed). We can also double the resynthesis hop size `n2=2*n1`.

(3) One can take an arbitrary combination of the two phases, but one should remember that phases are given modulo 2π (except if they have been unwrapped).

(4) Design of an arbitrary variation of the phases.

As a matter of fact, these mutations are very experimental, and are very near to the construction of a true arbitrary time-frequency representation, but with some cues coming from the analysis of different sounds.

7.4.7 Robotization

This technique puts zero phase values on every FFT before reconstruction. The effect applies a fixed pitch onto a sound. Moreover, as it forces the sound to be periodic, many erratic and random variations are converted into robotic sounds. The sliding FFT of pulses, where the analysis is taken at the time of these pulses will give a zero phase value for the phase of the FFT. This is a clear indication that putting a zero phase before an IFFT resynthesis will give a fixed pitch sound. This is reminiscent of the PSOLA technique, but here we do not make any assumption on the frequency of the analyzed sound and no marker has to be found. So zeroing the phase can be viewed from two points of view:

(1) The result of an IFFT is a pulse-like sound and summing such grains at regular intervals gives a fixed pitch.

(2) This can also be viewed as an effect of the reproducing kernel on the time-frequency representation: due to fact that the time-frequency representation now shows a succession of vertical lines with zero values in between, this will lead to a comb-filter effect during resynthesis.

The following M-file 7.16 demonstrates the *robotization effect*.

M-file 7.16 (VX_robot.m)

```
% VX_robot.m   [DAFXbook, 2nd ed., chapter 7]
%===== this program performs a robotization of a sound
clear; clf

%----- user data -----
n1            = 441;    % analysis step [samples]
n2            = n1;     % synthesis step [samples]
s_win         = 1024;   % analysis window length [samples]
[DAFx_in,FS] = wavread('redwheel.wav');

%----- initialize windows, arrays, etc -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in)-s_win;
while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
  f     = fft(grain);
  r     = abs(f);
  grain = fftshift(real(ifft(r))).*w2;
% ============================================
  DAFx_out(pout+1:pout+s_win) = ...
    DAFx_out(pout+1:pout+s_win) + grain;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'redwheel_robot.wav');
```

This is one of the shortest programs we can have, however its effect is very strong. The only drawback is that the `n1` value in this program has to be an integer. The frequency of the robot is `Fs/n1`, where `Fs` is the sampling frequency. If the hop size is not an integer value, it is possible to use an interpolation scheme in order to dispatch the grain of two samples. This may happen if the hop size is calculated directly from a fundamental frequency value. An example is shown in Figure 7.30.

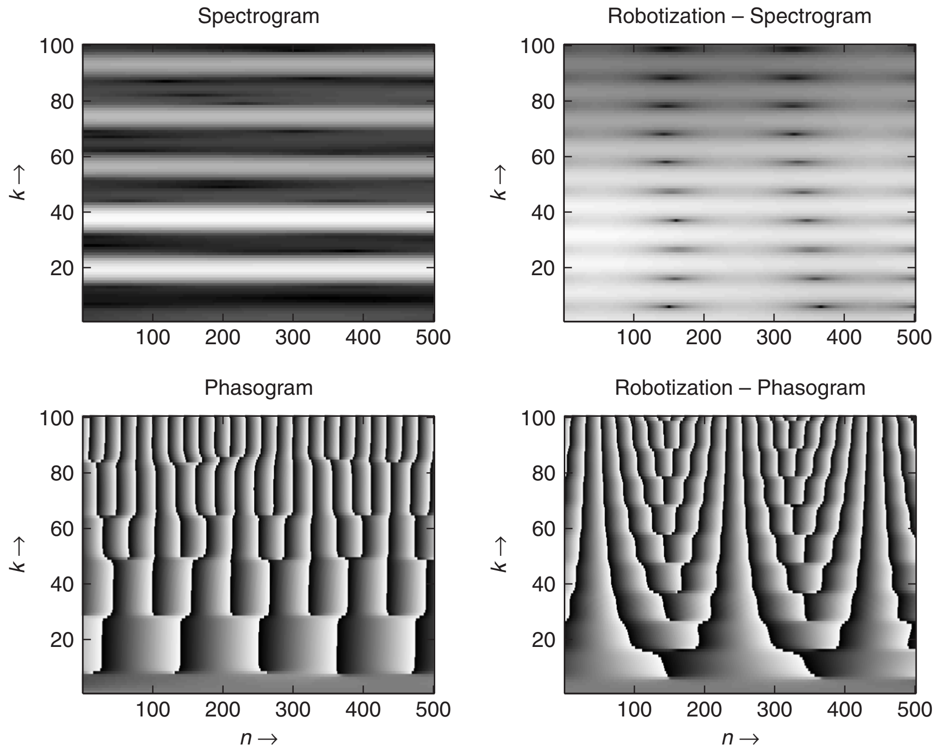


Figure 7.30 Example of robotization with a flute signal.

7.4.8 Whisperization

If we deliberately impose a random phase on a time-frequency representation, we can have a different behavior depending on the length of the window: if the window is quite large (for example, 2048 for a sampling rate of 44100 Hz), the magnitude will represent the behavior of the partials quite well and changes in phase will produce an uncertainty over the frequency. But if the window is small (e.g., 64 points), the spectral envelope will be enhanced and this will lead to a whispering effect. The M-file 7.17 implements the *whisperization effect*.

M-file 7.17 (VX_whisper.m)

```
% VX_whisper.m   [DAFXbook, 2nd ed., chapter 7]
%===== This program makes the whisperization of a sound,
%===== by randomizing the phase
clear; clf

%----- user data -----
s_win        = 512;     % analysis window length [samples]
n1           = s_win/8; % analysis step [samples]
n2           = n1;      % synthesis step [samples]
[DAFx_in,FS] = wavread('redwheel.wav');

%----- initialize windows, arrays, etc -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;
while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%============================================
  f     = fft(fftshift(grain));
  r     = abs(f);
  phi   = 2*pi*rand(s_win,1);
  ft    = (r.* exp(i*phi));
  grain = fftshift(real(ifft(ft))).*w2;
% ===========================================
  DAFx_out(pout+1:pout+s_win) = ...
    DAFx_out(pout+1:pout+s_win) + grain;
  pin   = pin + n1;
  pout  = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
```

```
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'whisper2.wav');
```

It is also possible to make a random variation of the magnitude and keep the phase. An example is shown in Figure 7.31. This gives another way to implement whisperization, which can be achieved by the following **MATLAB** kernel:

```
%===========================================
    f     = fft(fftshift(grain));
    r     = abs(f).*randn(lfen,1);
    phi   = angle(f);
    ft    = (r.* exp(i*phi));
    grain = fftshift(real(ifft(ft))).*w2;
%===========================================
```

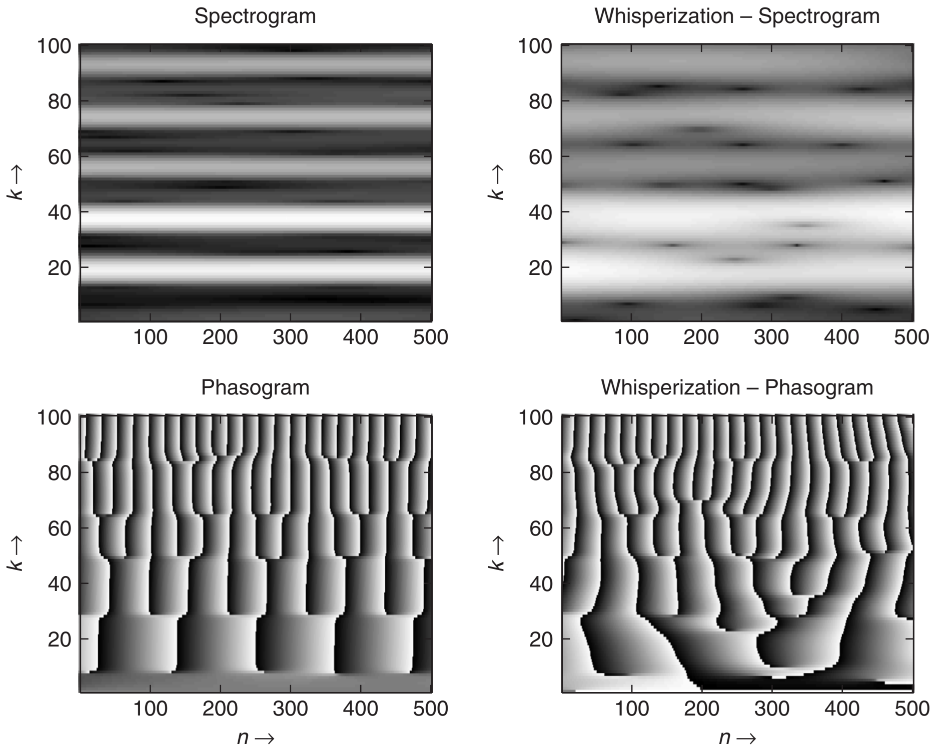


Figure 7.31 Example of whisperization with a flute signal.

7.4.9 Denoising

A musician may want to emphasize some specific areas of a spectrum and lower the noise within a sound. Though this is achieved more perfectly by the use of a sinusoidal model (see Chapter 10), another approach is the use of denoising algorithms. The algorithm we describe uses a non-linear spectral subtraction technique [Vas96]. Further techniques can be found in [Cap94]. A time-frequency analysis and resynthesis are performed, with an extraction of the magnitude and phase information. The phase is kept as it is, while the magnitude is processed in such a way

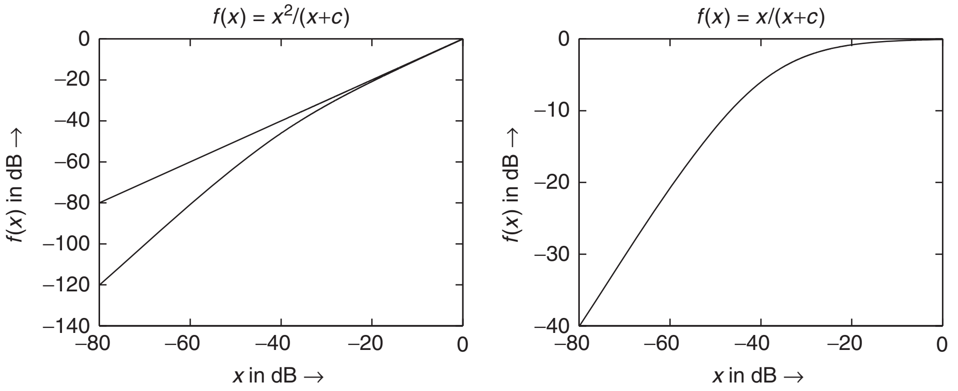


Figure 7.32 Non-linear function for a noise gate.

that it keeps the high-level values while attenuating the lower ones, in such a way as to attenuate the noise. This can also be seen as a bank of noise gates on different channels, because on each bin we perform a non-linear operation. The denoised magnitude vector $X_d(n, k) = f(X(n, k))$ of the denoised signal is then the output of a noise gate with a non-linear function $f(x)$. A basic example of such a function is $f(x) = x^2/(x + c)$, which is shown in Figure 7.32. It can also be seen as the multiplication of the magnitude vector by a correction factor $x/(x + c)$. The result of such a waveshaping function on the magnitude spectrum keeps the high values of the magnitude and lowers the small ones. Then the phase of the initial signal is reintroduced and the sound is reconstructed by overlapping grains with the help of an IFFT. The following M-file 7.18 follows this guideline.

M-file 7.18 (VX_denoise.m)

```
% VX_denoise.m  [DAFXbook, 2nd ed., chapter 7]
%===== This program makes a denoising of a sound
clear; clf

%----- user data -----
n1          = 512;   % analysis step [samples]
n2          = n1;    % synthesis step [samples]
s_win       = 2048;  % analysis window length [samples]
[DAFx_in,FS] = wavread('redwheel.wav');

%----- initialize windows, arrays, etc -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;    % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),1);
hs_win   = s_win/2;
coef     = 0.01;
freq     = (0:1:299)/s_win*44100;

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

```
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;

while pin<pend
  grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
  f     = fft(grain);
  r     = abs(f)/hs_win;
  ft    = f.*r. / (r+coef);
  grain = (real(ifft(ft))).*w2;
% ===========================================
  DAFx_out(pout+1:pout+s_win) = ...
    DAFx_out(pout+1:pout+s_win) + grain;
  pin  = pin + n1;
  pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc
%----- listening and saving the output -----
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L);
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'x1_denoise.wav');
```

An example is shown in Figure 7.33. It is of course possible to introduce different noise-gate functions instead of the simple ones we have chosen. For instance, the formulation

```
  ft = f.*r. / (r+coef);
```

can be replaced by the following one:

```
  ft = f.*r. / max(r,coef);
```

With the first formulation, a bias is introduced to the magnitude of each frequency bin, whereas in the second formulaiton, the bias is only introduced for frequency bins that are actually denoised.

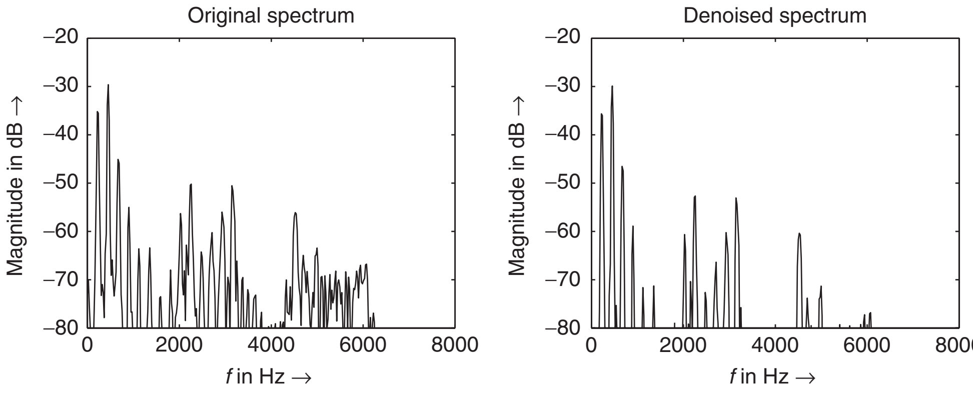


Figure 7.33 The left plot shows the windowed FFT of a flute sound. The right plot shows the same FFT after noise gating each bin using the r/(r+coef) gating function with $c = 0.01$.

Denoising in itself has many variations depending on the application:

(1) Denoising from a tape recorder usually starts from the analysis of a noisy sound coming from a recording of silence. This gives a gaboret for the noise shape, so that the non-linear function will be different for each bin, and can be zero under this threshold.

(2) The noise level can be estimated in a varying manner. For example, one can estimate a noise threshold which can be spectrum dependent. This usually involves spectral estimation techniques (with the help of LPC or cepstrum), which will be seen later.

(3) One can also try to evaluate a level of noise on successive time instances in order to decrease pumping effects.

(4) In any case, these algorithms involve non-linear operations and as such can produce artifacts. One of them is the existence of small grains that remain outside the silence unlike the previous noise (spurious components). The other artifact is that noise can sometimes be a useful component of a sound and will be suppressed as undesirable noise.

7.4.10 Spectral panning

Another sound transformation is to place sound sources in various physical places, which can be simulated in stereophonic sound by making use of azimuth panning. But what happens when trying to split the sound components? Then, each bin of the spectrum can be separately positioned using panning. Even better, by making use of the information from a phase-locked vocoder, one may put all frequency bins of the same component in the same angular position, avoiding the positioning artefacts that result from two adjacent bins that are coding for the same component and that would be otherwise placed in two different positions (the auditory image then being in the two places).

The Blumlein law [Bla83] for each sound sample is adapted to each frequency bin as

$$\sin\theta(n,k) = \frac{g_L(n,k) - g_R(n,k)}{g_L(n,k) + g_R(n,k)} \sin\theta_l \ , \tag{7.48}$$

where $g_L(n,k)$ and $g_L(n,k)$ are the time- and frequency-bin-varying gains to be applied to the left and right stereo channels, $\theta(n,k)$ is the angle of the virtual source position, and θ_l is the angle formed by each loudspeaker with the frontal direction. In the simple case where $\theta_l = 45^\circ$ the output spectra are given by

$$X_L(n,k) = \frac{\sqrt{2}}{2}\left(\cos\theta(n,k) + \sin\theta(n,k)\right) \cdot X(n,k) \tag{7.49}$$

$$X_R(n,k) = \frac{\sqrt{2}}{2}\left(\cos\theta(n,k) - \sin\theta(n,k)\right) \cdot X(n,k). \tag{7.50}$$

A basic example is shown in Figure 7.34. Only the magnitude spectrum is modified: the phase of the initial signal is used unmodified, because this version of azimuth panning only modifies the gains, but not the fundamental frequency. This means that such effect does not simulate the Doppler effect. The stereophinc sound is then reconstructed by overlapping grains with the help of an IFFT. The following M-file 7.19 follows this guideline.

M-file 7.19 (VX_specpan.m)

```
% VX_specpan.m  [DAFXbook, 2nd ed., chapter 7]
%===== This program makes a spectral panning of a sound
clear; clf
```

```
%----- user data -----
fig_plot     = 1;    % use any value except 0 or [] to plot figures
n1           = 512;  % analysis step [samples]
n2           = n1;   % synthesis step [samples]
s_win        = 2048; % analysis window length [samples]
[DAFx_in,FS] = wavread('redwheel.wav');

%----- initialize windows, arrays, etc -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;       % synthesis window
L        = length(DAFx_in);
DAFx_in  = [zeros(s_win, 1); DAFx_in; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in));
DAFx_out = zeros(length(DAFx_in),2);
hs_win   = s_win/2;
coef     = sqrt(2)/2;
%---- control: clipped sine wave with a few periods; in [-pi/4;pi/4]
theta    = min(1,max(-1,2*sin((0:hs_win)/s_win*200))).' * pi/4;
% %---- control: rough left/right split at Fs/30 ~ 1470 Hz
% theta    = (((0:hs_win).'/2 < hs_win/30)) * pi/2 - pi/4;
%---- preserving phase symmetry ----
theta    = [theta(1:hs_win+1); flipud(theta(1:hs_win-1))];

%---- drawing panning function ----
if (fig_plot)
  figure;
  plot((0:hs_win)/s_win*FS/1000, theta(1:hs_win+1));
  axis tight; xlabel('f / kHz \rightarrow');
  ylabel('\theta / rad \rightarrow');
  title('Spectral panning angle as a function of frequency')
end
tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = length(DAFx_in) - s_win;

while pin<pend
    grain = DAFx_in(pin+1:pin+s_win).* w1;
%===========================================
    f      = fft(grain);
    %---- compute left and right spectrum with Blumlein law at 45°
    ftL    = coef * f .* (cos(theta) + sin(theta));
    ftR    = coef * f .* (cos(theta) - sin(theta));
    grainL = (real(ifft(ftL))).*w2;
    grainR = (real(ifft(ftR))).*w2;
% ============================================
    DAFx_out(pout+1:pout+s_win,1) = ...
        DAFx_out(pout+1:pout+s_win,1) + grainL;
    DAFx_out(pout+1:pout+s_win,2) = ...
        DAFx_out(pout+1:pout+s_win,2) + grainR;
    pin  = pin + n1;
    pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc
%----- listening and saving the output -----
```

```
% DAFx_in  = DAFx_in(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L,:) / max(max(abs(DAFx_out)));
soundsc(DAFx_out, FS);
wavwrite(DAFx_out, FS, 'x1_specpan.wav');
```

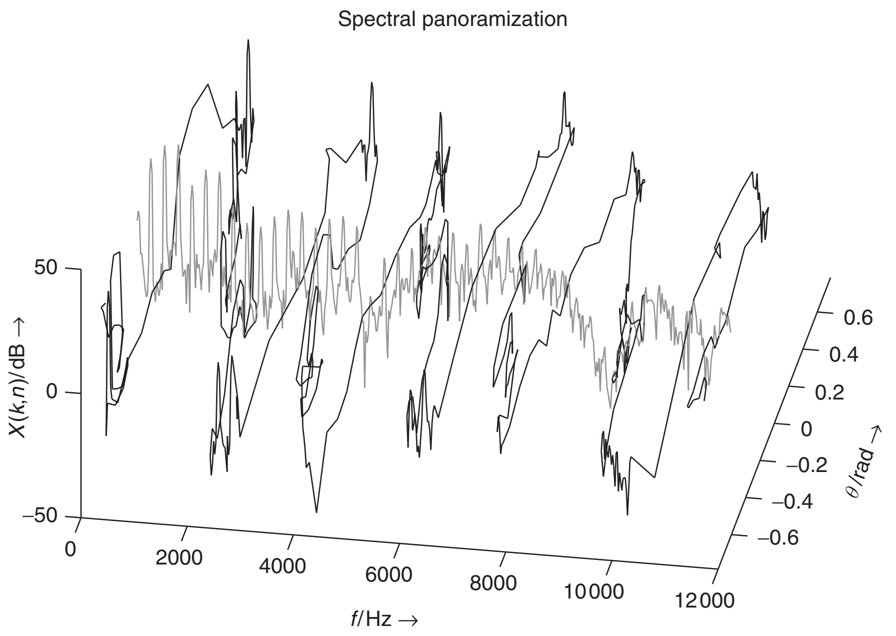


Figure 7.34 Spectral panning of the magnitude spectrum of a single frame, using a wave form as the panning angle. Each frequency bin of the original STFT $X(n, k)$ (centered with $\theta = 0$, in gray) is panoramized with constant power.

Spectral panning offers many applications, which are listed to show the variety of possibilities:

(1) Pseudo-source separation when combined with a multi-pitch detector.

(2) Arbitrary panning when the $\theta(n, k)$ value is not related to the spectral content of the sound.

(3) Vomito effect, when the azimuth angles are refreshed too often (i.e., every frame at a 50 Hz rate), the fast motions of sound also implies timbre modulations (as the amplitude modulation has a frequency higher than 20 Hz), both of which result in some unpleasant effects for the listener.

7.5 Conclusion

The starting point of this chapter was the computation of a time-frequency representation of a sound, to manipulate this representation and reproduce a sound. At first sight this may appear as an easy task, but we have seen that the basis for this time-frequency processing needs a careful description of the fundamentals, because the term 'vocoder' can cover different implementations. We also explained that the arbitrary manipulation of time-frequency representations renders images in a way that they are no longer time-frequency representations of "real" sounds. This phenomenon leads to artifacts, which cannot be avoided.

Digital audio effects described in this chapter only perform manipulations of these time-frequency representations. These effects exclude the extraction of resonances, which will be the subject of the next chapter, and high-level processing such as the extraction of sinusoids and noise. For example, the mentioned bank of filters does not assume any parametric model of the sound. Nevertheless such effects are numerous and diverse. Some of them have brought new solutions to well-known techniques such as filtering. Pitch shifting and time stretching have shown their central place in the phase vocoder approach, which is another implementation possibility independent of the time-segment processing approach from the previous chapter. Their was a clear need for a clarification of the phase vocoder approach in this domain. Though it has been known for years, we have provided a general framework and simple implementations upon which more complex effects may be built. Some of them can reduce the phasiness of the process or perform special high-level processing on transients. Other digital audio effects have been described that fit well under the name "mutations." They are based on modifying the magnitude and phase of one or two time-frequency representations. They put a special flavor on sounds, which musicians characterize as granulation, robotization, homogenization, purification, metallization and so on. Once again, the goal of this chapter is to give a general framework and unveil some of the basic implementations of these alterations of sound, which can be extended to more complex modifications at will.

As a final remark, one can say that no digital audio effect, and time-frequency processing in particular, would exist without a sound. Only a good adaptation of the sound with the effect can give rise to musical creativity. This is the reason why some of the basic algorithms presented put in the hands of creative musicians and artists can give better results than much more complex algorithms in the hands of conventional persons.

References

[AD93] D. Arfib and N. Delprat. Musical transformations using the modification of time-frequency images. *Comp. Music J.*, 17(2): 66–72, 1993.

[Bla83] J. Blauert. *Spatial Hearing: the Psychophysics of Human Sound Localization*. MIT Press, 1983.

[Cap94] O. Cappé. Elimination of the musical noise phenomenon with the ephraim and malah noise suppressor. *IEEE Trans. Acoust. Speech Signal Process.*, 2: 345–349, 1994.

[CGT89] J. M. Combes, A. Grossmann, and Ph. Tchamitchan (eds). *Wavelets. Time Frequency Methods and Phase Space*, 2nd edition. Springer-Verlag, 1989.

[Chu92] C. H. Chui. *An Introduction to Wavelets*. Academic Press, 1992.

[CR83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice-Hall, 1983.

[Cro80] R. E. Crochiere. A weighted overlap-add method of short-time fourier analysis/synthesis. *IEEE Trans. Acoust. Speech Signal Process.*, 281(1): 99–102, 1980.

[Fer99] A. J. S. Ferreira. An odd-DFT based approach to time-scale expansion of audio signals. *IEEE Trans. Speech Audio Process.*, 7(4): 441–453, 1999.

[GBA00] A. De Götzen, N. Bernadini, and D. Arfib. Traditional (?) implementations of a phase vocoder: The tricks of the trade. In *Proc. DAFX-00 Conference on Digital Audio Effects*, pp. 37–43, 2000.

[Lar98] J. Laroche. Time and pitch scale modifications of audio signals. In M. Kahrs and K.-H. Brandenburg (eds), *Applications of Digital Signal Processing to Audio and Acoustics*, pp. 279–309. Kluwer, 1998.

[Lar03] J. Laroche. Frequency-domain techniques for high quality voice modification. In *Proc. 6th DAFX-03 Conf. Digital Audio Effects*, pp. 328–332, 2003.

[LD99a] J. Laroche and M. Dolson. Improved phase vocoder time-scale modification of audio. *IEEE Trans. Speech Audio Process.*, 7(3): 323–332, 1999.

[LD99b] J. Laroche and M. Dolson. New phase-vocoder techniques for real-time pitch shifting, chorusing, harmonizing, and other exotic audio modifications. *J. Audio Eng. Soc.*, 47(11): 928–936, 1999.

[PE96] L. Polansky and T. Erbe. Spectral mutation in soundhack. *Comp. Music J.*, 20(1): 92–101, Spring 1996.

[Por76] M. R. Portnoff. Implementation of the digital phase vocoder using the fast fourier transform. *IEEE Trans. Acoust. Speech Signal Process*., 24(3): 243–248, 1976.

[QC93] S. Quian and D. Chen. Discrete gabor transform. *IEEE Trans. Signal Process*., 41(7): 2429–2438, 1993.

[QM98] T. F. Quatieri and R. J. McAulay. Audio signal processing based on sinusoidal analysis/synthesis. In M. Kahrs and K.-H. Brandenburg (eds), *Applications of Digital Signal Processing to Audio and Acoustics*, pp. 343–416. Kluwer, 1998.

[Röb03] A. Röbel. A new approach to transient processing in the phase vocoder. In *Proc. 6th DAFX-03 Conf. Digital Audio Effects*, pp. 344–349, 2003.

[Röb10] A. Röbel. A shape-invariant phase vocoder for speech transformation. In *Proc. 13th DAFX-10 Conf. Digital Audio Effects*, pp. 298–305, 2010.

[SL94] Z. Settel and C. Lippe. Real-time musical applications using the FFT-based resynthesis. In *Proc. Int. Comp. Music Conf*., 1994.

[Vas96] S. V. Vaseghi. *Advanced Signal Processing and Digital Noise Reduction*. Wiley & Teubner, 1996.

[WR90] J. Wexler and S. Raz. Discrete gabor expansions. *Signal Process*., 21(3): 207–220, 1990.

[Zöl05] U. Zölzer. *Digital Audio Signal Processing*, 2nd edition. John Wiley & Sons, Ltd, 2005.