# 8

# Source-filter processing

**D. Arfib, F. Keiler, U. Zölzer and V. Verfaille**

## 8.1 Introduction

Time – frequency representations give the evolution over time of a spectrum calculated from temporal frames. The notion of the spectral envelope extracted from such representations mostly comes from the voice production and recognition system: the voice production uses vocal chords as an excitation and the mouth and nose as a resonator system or anti-resonator. Voiced signals (vowels) produce a harmonic spectrum on which a spectral envelope is superimposed. This fact about voice strongly influences our way of recognizing other sounds, whether because of the ear or the brain; we are looking for such a spectral envelope as a cue to the identification or classification of sounds. This excitation-resonance model is also called source-filter model in the literature. Thus we can understand why the *vocoding* effect, which is the cross-synthesis of a musical instrument with voice, is so attractive for the ear and so resistant to approximations. We will make use of a source-filter model for an audio signal and modify this model in order to achieve different digital audio effects.

However, the signal-processing problem of extracting a spectral envelope from a spectrum is generally badly conditioned. If the sound is purely harmonic we could say that the spectral envelope is the curve that passes through the points related to these harmonics. This leaves two open questions: how to retrieve these exact values of these harmonics, and what kind of interpolation scheme should we use for the completion of the curve in-between these points? But, more generally, if the sound contains inharmonic partials or a noisy part, this definition no longer holds and the notion of a spectral envelope is then completely dependent on the definition of what belongs to the excitation and what belongs to the resonance. In a way it is more a "envelope-recognition" problem than a "signal-processing" one.

With this in mind we will state that a spectral envelope is a smoothing of a spectrum, which tends to leave aside the spectral line structure while preserving the general form of the spectrum.

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

To provide source-filter sound transformation, two different steps are performed:

(1) Estimate the spectral envelope

(2) Perform the source-filter separation, and sound-filter combination after transformation of one or the other.

There are three techniques with many variants which can be used for both steps:

(1) The **channel vocoder** uses frequency bands and performs estimations of the amplitude of the signal inside these bands and thus the spectral envelope.

(2) **Linear prediction** estimates an all-pole filter that matches the spectral content of a sound. When the order of this filter is low, only the formants are taken, hence the spectral envelope.

(3) **Cepstrum** techniques perform smoothing of the logarithm of the FFT spectrum (in decibels) in order to separate this curve into its slowly varying part (the spectral envelope) and its quickly varying part (the source signal).

For each of these techniques, we will describe the fundamental algorithms in Section 8.2 which allow the calculation of the spectral envelope and the source signal in a frame-oriented approach, as shown in Figure 8.1. Then transformations are applied to the spectral envelope and/or the source signal and a synthesis procedure reconstructs the output sound. Some basic transformations are introduced in Section 8.3. The separation of a source and a filter is only one of the features we can extract from a sound, or more precisely from a time-frequency representation.

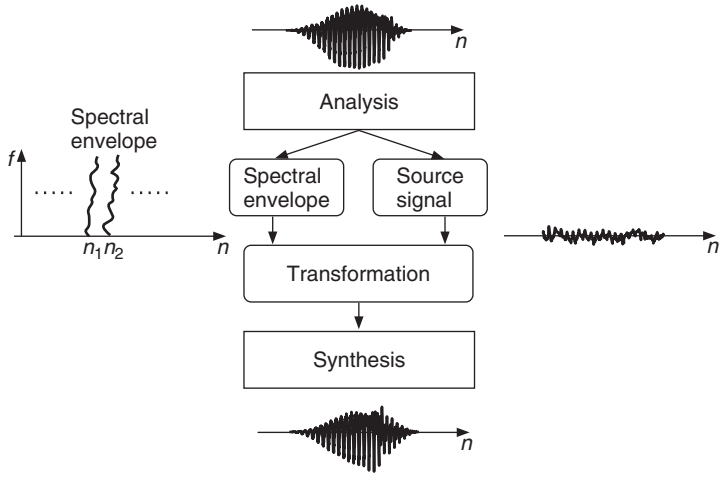


**Figure 8.1** Spectral processing based on time-varying spectral envelopes and source signals. The analysis performs a source and filter separation.

## 8.2 Source-filter separation

Digital audio effects based on source-filter processing extract the spectral envelope and the source (excitation) signal from an input signal, as shown in Figure 8.2. The input signal is whitened by the filter $1/H_1(z)$, which is derived from the spectral envelope of the input signal. In signal-processing terms, the spectral envelope is given by the magnitude response $|H_1(f)|$ or its logarithm $\log|H_1(f)|$ in dB. This leads to extraction of the source signal $e_1(n)$ which can be further processed, for example, by time-stretching or pitch-shifting algorithms. The processed source signal is then

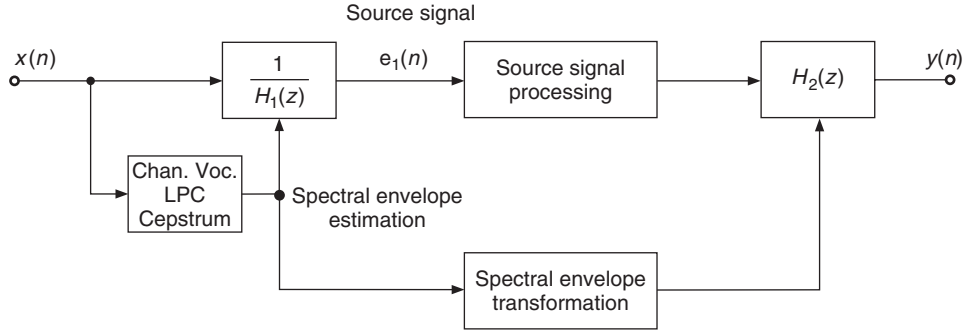


**Figure 8.2** Spectrum estimation (Channel vocoder, Linear Predictive Coding or Cepstrum) and source-signal extraction for individual processing.

finally filtered by $H_2(z)$. This filter is derived from the modified spectral envelope of the input signal or another source signal.

### 8.2.1 Channel vocoder

If we filter a sound with a bank of bandpass filters and calculate the RMS value for each bandpass signal, we can obtain an estimation of the spectral envelope (see Figure 8.3). The parameters of

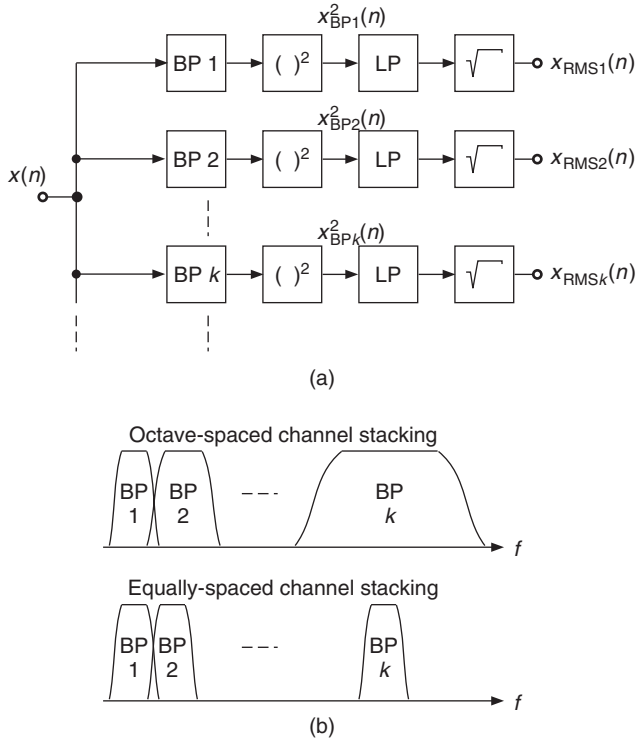


**Figure 8.3** (a) Channel vocoder and (b) frequency stacking.

the filters for each channel will of course affect the precision of the measurement, as well as the delay between the sound input and the spectral calculation. The RMS calculation parameters are also a compromise between good definition and an acceptable delay and trail effect. The spectral estimation is valid around the center frequency of the filters. Thus the more channels there are, the more frequency points of the spectral envelope are estimated. The filter bank can be defined on a linear scale, in which case every filter of the filter bank can be equivalent in terms of bandwidth. It can also be defined on a logarithmic scale. In this case, this approach is more like an "equalizer system" and the filters, if given in the time domain, are scaled versions of a mother filter.

The channel-vocoder algorithm shown in Figure 8.3 works in the time domain. There is, however, a possible derivation where it is possible to calculate the spectral envelope from the FFT spectrum, thus directly from the time-frequency representations. A channel can be represented in the frequency domain, and the energy of an effective channel filter can be seen as the sum of the elementary energies of each bin weighted by this channel filter envelope. The amplitude coming out of this filter is then the square root of these energies.

In the case of filters with equally spaced channel stacking (see Figure 8.3b), it is even possible to use a short-cut for the calculation of this spectral envelope: the spectral envelope is the square root of the filtered version of the squared amplitudes. This computation can be performed by a circular convolution $Y(k) = \sqrt{|X(k)|^2 * w(k)}$ in the frequency domain, where $w(k)$ may be a Hanning window function. The circular convolution is accomplished by another FFT/IFFT filtering algorithm. The result is a spectral envelope, which is a smoothing of the FFT values. An example is shown in Figure 8.4.

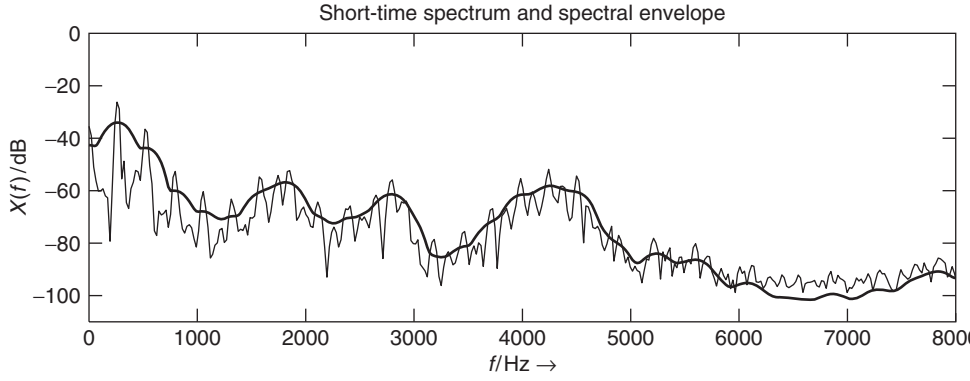


**Figure 8.4** Spectral envelope computation with a channel vocoder.

The following M-file 8.1 defines channels in the frequency domain and calculates the energy in dB inside successive channels of that envelope.

**M-file 8.1** (UX_specenv.m)

```
s_win = 2048;
w     = hanning(s_win, 'periodic');
buf   = y(offset:offset+s_win-1).*w;
f     = fft(buf)/(s_win/2);
freq  = (0:1:s_win-1)/s_win*44100;
flog  = 20*log10(0.00001+abs(f));
%---- frequency window ----
nob   = input('number of bins must be even = ');
w     = hanning(nob, 'periodic');
```

```
w1    = hanning(nob, 'periodic');
w1    = w1./sum(w1);
f_channel = [zeros((s_win-nob)/2,1); w1; zeros((s_win-nob)/2,1)];
%---- FFT of frequency window ----
fft_channel = fft(fftshift(f_channel));
f2    = f.*conj(f); % Squared FFT values
%---- circular convolution by FFT-Multiplication-IFFT ----
energy   = real(ifft(fft(f2).*fft_channel));
flog_rms = 10*log10(abs(energy)); % 10 => combination with sqrt operation
%---- plot result ----
subplot(2,1,1); plot(freq,flog,freq,flog_rms);
ylabel('X(f)/dB');
xlabel('f/Hz \rightarrow'); axis([0 8000 -110 0]);
title('Short-time spectrum and spectral envelope');
```

The program starts with the calculation of the FFT of a windowed frame, where `w` is a Hanning window in this case. The vector `y` contains the sound and a buffer `buf` contains a windowed segment. In the second part of this program `fchannel` represents the envelope of the channel with a FFT representation. Here it is a Hanning window of width `nob`, which is the number of frequency bins. The calculation of the weighted sum of the energies inside a channel is performed by a convolution calculation of the energy pattern and the channel envelope. Here, we use a circular convolution with an FFT-IFFT algorithm to easily retrieve the result for all channels. In a way it can be seen as a smoothing of the energy pattern. The only parameter is the envelope of the channel filter, hence the value of `nob` in this program. The fact that it is given in bins and that it should be even is only for the simplification of the code. The bandwidth is given by $\texttt{nob} \cdot \frac{f_S}{N}$ ($N$ is the length of the FFT).

## 8.2.2 Linear predictive coding (LPC)

One way to estimate the spectral envelope of a sound is directly based on a simple sound-production model. In this model, the sound is produced by passing an excitation source (source signal) through a synthesis filter, as shown in Figure 8.5. The filter models the resonances and has therefore only poles. Thus, this all-pole filter represents the spectral envelope of the sound. This model works well for speech, where the synthesis filter models the human vocal tract, while the excitation source consists of pulses plus noise [Mak75]. For voiced sounds the periodicity of the pulses determines the pitch of the sound, while for unvoiced sounds the excitation is noise-like.

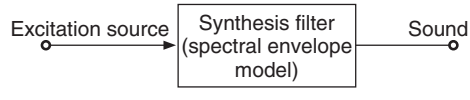


**Figure 8.5** Sound production model: the synthesis filter represents the spectral envelope.

The retrieval of the spectral envelope from a given sound at a given time is based on the estimation of the all-pole synthesis filter mentioned previously. This approach is widely used for speech coding and is called linear predictive coding (LPC) [Mak75, MG76].

### Analysis/Synthesis structure

In LPC the current input sample $x(n)$ is approximated by a linear combination of past samples of the input signal. The prediction of $x(n)$ is computed using an FIR filter by

$$\hat{x}(n) = \sum_{k=1}^{p} a_k x(n-k), \tag{8.1}$$

where $p$ is the *prediction order* and $a_k$ are the prediction coefficients. The difference between the original input signal $x(n)$ and its prediction $\hat{x}(n)$ is evaluated by

$$e(n) = x(n) - \hat{x}(n) = x(n) - \sum_{k=1}^{p} a_k x(n-k). \tag{8.2}$$

The difference signal $e(n)$ is called *residual* or *prediction error* and its calculation is depicted in Figure 8.6 where the transversal (direct) FIR filter structure is used.

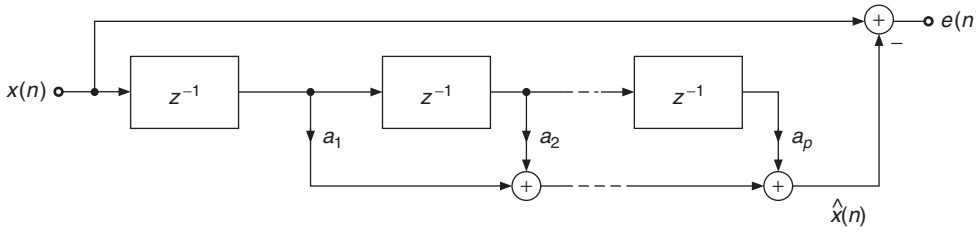


**Figure 8.6** Transversal FIR filter structure for the prediction-error calculation.

With the $z$-transform of the *prediction filter*

$$P(z) = \sum_{k=1}^{p} a_k z^{-k}, \tag{8.3}$$

Equation (8.2) can be written in the $z$-domain as

$$E(z) = X(z) - \hat{X}(z) = X(z)[1 - P(z)]. \tag{8.4}$$

Figure 8.7(a) illustrates the last equation. The illustrated structure is called *feed-forward prediction* where the prediction is calculated in the forward direction from the input signal.

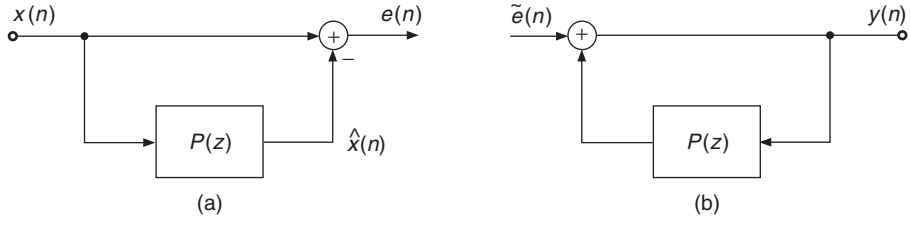


**Figure 8.7** LPC structure with feed-forward prediction: (a) analysis, (b) synthesis.

Defining the *prediction error filter* or *inverse filter*

$$A(z) = 1 - P(z) = 1 - \sum_{k=1}^{p} a_k z^{-k}, \tag{8.5}$$

the prediction error is obtained as

$$E(z) = X(z)A(z). \tag{8.6}$$

The sound signal is recovered by using the *excitation signal* $\tilde{e}(n)$ as input to the all-pole filter

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 - P(z)}. \tag{8.7}$$

This yields the output signal

$$Y(z) = \tilde{E}(z) \cdot H(z), \tag{8.8}$$

where $H(z)$ can be realized with the FIR filter $P(z)$ in a feedback loop, as shown in Figure 8.7(b). If the residual $e(n)$, which is calculated in the analysis stage, is fed directly into the synthesis filter, the input signal $x(n)$ will be ideally recovered.

The IIR filter $H(z)$ is termed *synthesis filter* or *LPC filter* and represents the spectral model – except for a gain factor – of the input signal $x(n)$. As mentioned previously, this filter models the time-varying vocal tract in the case of speech signals.

With optimal filter coefficients, the residual energy is minimized. This can be exploited for efficient coding of the input signal where the quantized residual $\tilde{e}(n) = Q\{e(n)\}$ is used as excitation to the LPC filter.

Figure 8.8 shows an example where for a short block of a speech signal an LPC filter of order $p = 50$ is computed. In the left plot the time signal is shown, while the right plot shows both the spectra of the input signal and of the LPC filter $H(z)$. In this example the autocorrelation method is used to calculate the LPC coefficients. The MATLAB® code for this example is given by M-file 8.2 (the used function `calc_lpc` will be explained later).

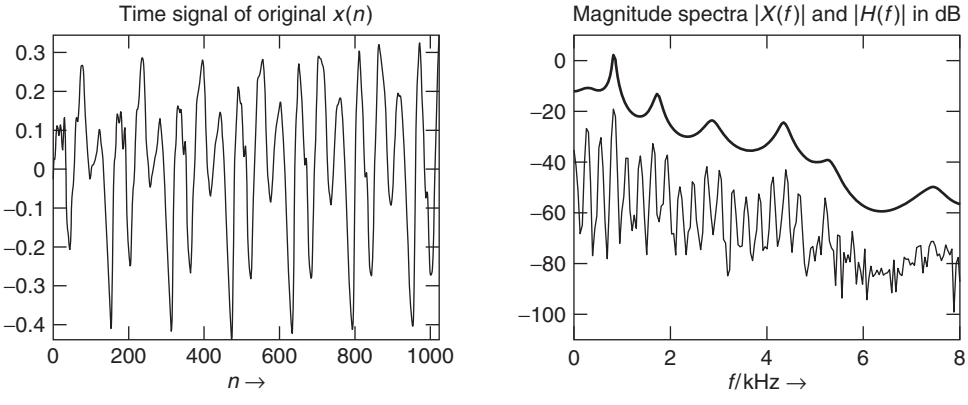


**Figure 8.8** LPC example for the female utterance "la" with prediction order $p = 50$, original signal and LPC filter.

**M-file 8.2** (`figure8_08.m`)

```
N     = 1024;   % block length
Nfft  = 1024;   % FFT length
p     = 50;     % prediction order
n1    = n0+N-1; % end index

[xin,Fs] = wavread(fname,[n0 n1]);
x     = xin(:,1)';   % row vecor of left channel
win   = hamming(N)'; % window for input block
```

```
a = calc_lpc(x.*win,p); % calculate LPC coeffs
% a = [1, -a_1, -a_2,..., -a_p]

Omega  = (0:Nfft-1)/Nfft*Fs/1000; % frequencies in kHz
offset = 20*log10(2/Nfft);        % spectrum offset in dB
A = 20*log10(abs(fft(a,Nfft)));
H = -A+offset;
X = 20*log10(abs(fft(x.*win,Nfft)));
X = X+offset;

n = 0:N-1;
```

### Calculation of the filter coefficients

To find an all-pole filter which models a considered sound well, different approaches may be taken. Some common methods compute the filter coefficients from a block of the input signal $x(n)$. These methods are namely the autocorrelation method [Mak75, Orf90], the covariance method [Mak75, MG76], and the Burg algorithm [Mak77, Orf90]. Since both the autocorrelation method and the Burg algorithm compute the lattice coefficients, they are guaranteed to produce stable synthesis filters, while the covariance method may yield unstable filters.

Now we briefly describe the autocorrelation method, which minimizes the energy of the prediction error $e(n)$. With the prediction error $e(n)$ defined in (8.2), the prediction error energy is[1]

$$E_p = E\left\{e^2(n)\right\}. \tag{8.9}$$

Setting the partial derivatives of $E_p$ with respect to the filter coefficients $a_i$ $(i = 1, \ldots, p)$ to zero leads to

$$\frac{\partial E_p}{\partial a_i} = 2E\left\{e(n) \cdot \frac{\partial e(n)}{\partial a_i}\right\} \tag{8.10}$$

$$= -2E\left\{e(n)x(n-i)\right\} \tag{8.11}$$

$$= -2E\left\{\left[x(n) - \sum_{k=1}^{p} a_k x(n-k)\right] x(n-i)\right\} = 0 \tag{8.12}$$

$$\Leftrightarrow \sum_{k=1}^{p} a_k E\left\{x(n-k)x(n-i)\right\} = E\left\{x(n)x(n-i)\right\}. \tag{8.13}$$

Equation (8.13) is a formulation of the so-called *normal equations* [Mak75]. The autocorrelation sequence for a block of length $N$ is defined by

$$r_{xx}(i) = \sum_{n=i}^{N-1} u(n)u(n-i) \tag{8.14}$$

where $u(n) = x(n) \cdot w(n)$ is a windowed version of the considered block $x(n)$, $n = 0, \ldots, N-1$. Normally a Hamming window is used [O'S00]. The expectation values in (8.13) can be replaced

[1] With the expectation value $E\{\cdot\}$.

by their approximations using the autocorrelation sequence, which gives the normal equations[2]

$$\sum_{k=1}^{p} a_k r_{xx}(i-k) = r_{xx}(i) \quad , i = 1, \ldots, p. \tag{8.15}$$

The filter coefficients $a_k$ ($k = 1, \ldots, p$), which model the spectral envelope of the used segment of $x(n)$ are obtained by solving the normal equations. An efficient solution of the normal equations is performed by the Levinson – Durbin recursion [Mak75].

As explained in [Mak75], minimizing the residual energy is equivalent to finding a best spectral fit in the frequency domain, if the gain factor is ignored. Thus the input signal $x(n)$ is modeled by the filter

$$H_g(z) = G \cdot H(z) = \frac{G}{1 - \sum_{k=1}^{p} a_k z^{-k}}, \tag{8.16}$$

where $G$ denotes the gain factor. With this modified synthesis filter the original signal is modeled using a white noise excitation with unit variance. For the autocorrelation method the gain factor is defined by [Mak75]

$$G^2 = r_{xx}(0) - \sum_{k=1}^{p} a_k r_{xx}(k), \tag{8.17}$$

with the autocorrelation sequence given in (8.14). Hence the gain factor depends on the energy of the prediction error. If $|H_g(e^{j\Omega})|^2$ models the power spectrum $|X(e^{j\Omega})|^2$, the prediction-error power spectrum is a flat spectrum with $|E(e^{j\Omega})|^2 = G^2$. The inverse filter $A(z)$ to calculate the prediction error is therefore also called the "whitening filter" [Mak75]. The **MATLAB** code of the function `calc_lpc` for the calculation of the prediction coefficients and the gain factor using the autocorrelation method is given by M-file 8.3.

**M-file 8.3** (`calc_lpc.m`)

```
% function [a,g] = calc_lpc(x,p) [DAFXbook, 2nd ed., chapter 8]
% ===== This function computes LPC coeffs via autocorrelation method
%       Similar to MATLAB function "lpc"
% !!! IMPORTANT: function "lpc" does not work correctly with MATLAB 6!
% Inputs:
%   x: input signal
%   p: prediction order
% Outputs:
%   a: LPC coefficients
%   g: gain factor
% (c) 2002 Florian Keiler
function [a,g] = calc_lpc(x,p)

R = xcorr(x,p);       % autocorrelation sequence R(k) with k=-p,..,p
R(1:p) = [];          % delete entries for k=-p,..,-1
if norm(R)~=0
   a = levinson(R,p);    % Levinson-Durbin recursion
%   a = [1, -a_1, -a_2,..., -a_p]
else
```

[2] The multiplication of the expectation values by the block length $N$ does not have any effect on the normal equations.

```
    a = [1, zeros(1,p)];
end
R = R(:)'; a = a(:)';   % row vectors
g = sqrt(sum(a.*R));  % gain factor
```

Notice that normally the **MATLAB** function `lpc` can be used, but it does not show the expected behavior for an input signal with zeros. M-file 8.3 fixes this problem and returns coefficients $a_k = 0$ in this case.

Figure 8.9 shows the prediction error and the estimated spectral envelope for the input signal shown in Figure 8.8. It can clearly be noticed that the prediction error has strong peaks occurring with the period of the fundamental frequency of the input signal. We can make use of this property of the prediction error signal for computing the fundamental frequency. The fundamental frequency and its pitch period can deliver pitch marks for PSOLA time-stretching or pitch-shifting algorithms, or other applications. The corresponding **MATLAB** code is given by M-file 8.4.

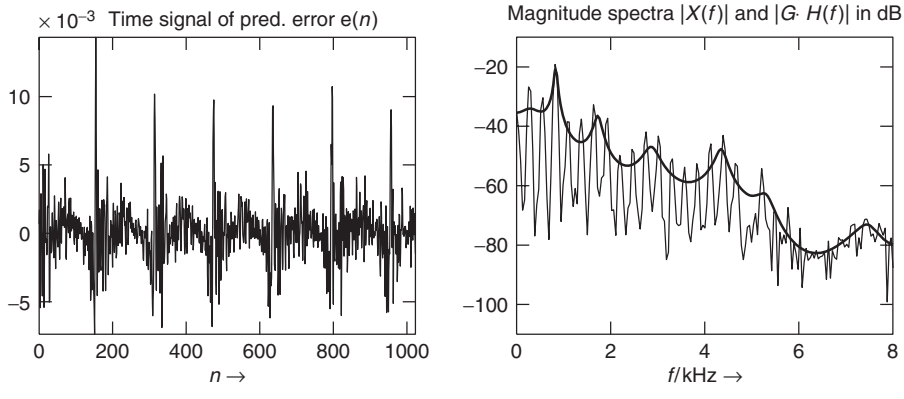


**Figure 8.9** LPC example for the female utterance "la" with prediction order $p = 50$, prediction error and spectral envelope.

**M-file 8.4** (`figure8_09.m`)

```
Nfft = 1024; % FFT length
p    = 50; %prediction order
n1   = n0+N-1; %end index
pre  = p; %filter order= no. of samples required before n0

[xin,Fs] = wavread(fname,[n0-pre n1]);
xin = xin(:,1)';
win = hamming(N)';
x   = xin((1:N)+pre); % block without pre-samples

[a,g] = calc_lpc(x.*win,p); % calculate LPC coeffs and gain
% a = [1, -a_1, -a_2,..., -a_p]
g_db = 20*log10(g) % gain in dB

ein = filter(a,1,xin); % pred. error
e   = ein((1:N)+pre); % without pre-samples
Gp  = 10*log10(sum(x.^2)/sum(e.^2)) % prediction gain

Omega  = (0:Nfft-1)/Nfft*Fs/1000; % frequencies in kHz
offset = 20*log10(2/Nfft); % offset of spectrum in dB
```

```
A   = 20*log10(abs(fft(a,Nfft)));
H_g = -A+offset+g_db; % spectral envelope
X   = 20*log10(abs(fft(x.*win,Nfft)));
X   = X+offset;

n   =  0:N-1;
figure(1)
clf
subplot(221)
plot(n,e)
title('time signal of pred. error e(n)')
xlabel('n \rightarrow')
axis([0 N-1 -inf inf])

subplot(222)
plot(Omega,X); hold on
plot(Omega,H_g,'r','Linewidth',1.5); hold off
title('magnitude spectra |X(f)| and |G\cdot H(f)| in dB')
xlabel('f/kHz \rightarrow')
axis([0 8 -inf inf])
```

Thus for the computation of the prediction error over the complete block length, additional samples of the input signal are required. The calculated prediction error signal $e(n)$ is equal to the source or excitation which has to be used as input to the synthesis filter $H(z)$ to recover the original signal $x(n)$. For this example the prediction gain, defined as

$$G_p = \frac{\sum_{n=0}^{N-1} x^2(n)}{\sum_{n=0}^{N-1} e^2(n)}, \tag{8.18}$$

has the value 38 dB, and the gain factor is $G = -23$ dB.

Figure 8.10 shows spectra of LPC filters at different filter orders for the same signal block as in Figure 8.8. The bottom line shows the spectrum of the signal segment where only frequencies below 8 kHz are depicted. The other spectra in this plot show the results using the autocorrelation method with different prediction orders. For clarity reasons these spectra are plotted with different

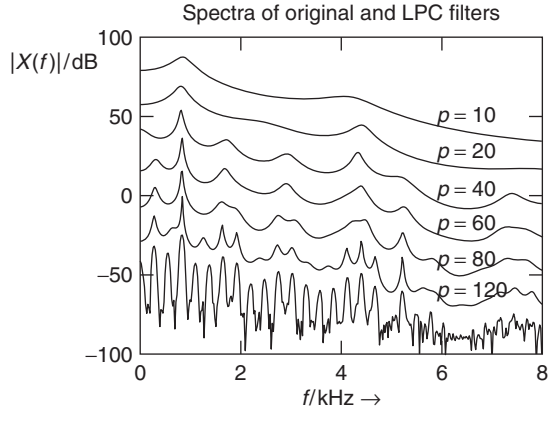


**Figure 8.10** LPC filter spectra for different prediction orders for the female utterance "la."

offsets. It is obvious that for an increasing prediction order the spectral model gets better although the prediction gain only increases from 36.6 dB ($p = 10$) to 38.9 dB ($p = 120$).

In summary, the LPC method delivers a source-filter model and allows the determination of pitch marks or the fundamental frequency of the input signal.

### 8.2.3 Cepstrum

The cepstrum (backward spelling of "spec") method allows the estimation of a spectral envelope starting from the FFT values $X(k)$ of a windowed frame $x(n)$. Zero padding and Hanning, Hamming or Gaussian windows can be used, depending on the number of points used for the spectral envelope estimation. An introduction to the basics of cepstrum-based signal processing can be found in [OS75]. The cepstrum is calculated from the discrete Fourier transform

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} = |X(k)| \mathrm{e}^{j\varphi_x(k)}, \quad k = 0, 1, \ldots, N-1 \tag{8.19}$$

by taking the logarithm

$$\hat{X}(k) = \log X(k) = \log |X(k)| + j\varphi_x(k) \tag{8.20}$$

and performing an IFFT of $\hat{X}(k)$, which yields the *complex cepstrum*

$$\hat{x}(n) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{X}(k) W_N^{-kn}. \tag{8.21}$$

The *real cepstrum* is derived from the real part of (8.20) given by

$$\hat{X}_R(k) = \log |X(k)| \tag{8.22}$$

and performing an IFFT of $\hat{X}_R(k)$, which leads to the *real cepstrum*

$$c(n) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{X}_R(k) W_N^{-kn}. \tag{8.23}$$

Since $\hat{X}_R(k)$ is an even function, the inverse discrete Fourier transform of $\hat{X}_R(k)$ gives an even function $c(n)$, which is related to the complex cepstrum $\hat{x}(n)$ by $c(n) = \frac{\hat{x}(n)+\hat{x}(-n)}{2}$.

Figure 8.11 illustrates the computational steps for the computation of the spectral envelope from the real cepstrum. The real cepstrum $c(n)$ is the IFFT of the logarithm of the magnitude of FFT of the windowed sequence $x(n)$. The lowpass window for weighting the cepstrum $c(n)$ is

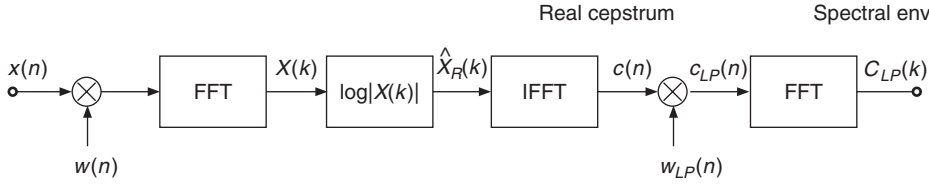


**Figure 8.11** Spectral-envelope computation by cepstrum analysis.

derived in [OS75] and is given by

$$w_{LP}(n) = \begin{cases} 1 & n = 0, N_1 \\ 2 & 1 \leq n < N_1 \\ 0 & N_1 < n \leq N - 1. \end{cases} \quad \text{with } N_1 \leq N/2. \tag{8.24}$$

The FFT of the windowed cepstrum $c_{LP}(n)$ yields the spectral envelope

$$C_{LP}(k) = \text{FFT}\left[c_{LP}(n)\right], \tag{8.25}$$

which is a smoothed version of the spectrum $X(k)$ in dB. An illustrative example is shown in Figure 8.12. Notice that the first part of the cepstrum $c(n)$ $(0 \leq n \leq 150)$ is weighted by the "lowpass window," yielding $c_{LP}(n)$. The IFFT of $c_{LP}(n)$ results in the spectral envelope $C(f)$ in dB, as shown in the lower right plot. The "highpass part" of the cepstrum $c(n)$ $(150 < n \leq 1024)$ represents the source signal, where the first peak at $n = 160$ represents the pitch period $T_0$ (in samples) of the fundamental frequency $f_0 = 44100 \text{ Hz}/160 = 275,625 \text{ Hz}$. Notice also, although the third harmonic is higher than the fundamental frequency, as can be seen in the spectrum of the segment, the cepstrum method allows the estimation of the pitch period of the fundamental frequency by searching for the time index of the first highly significant peak value

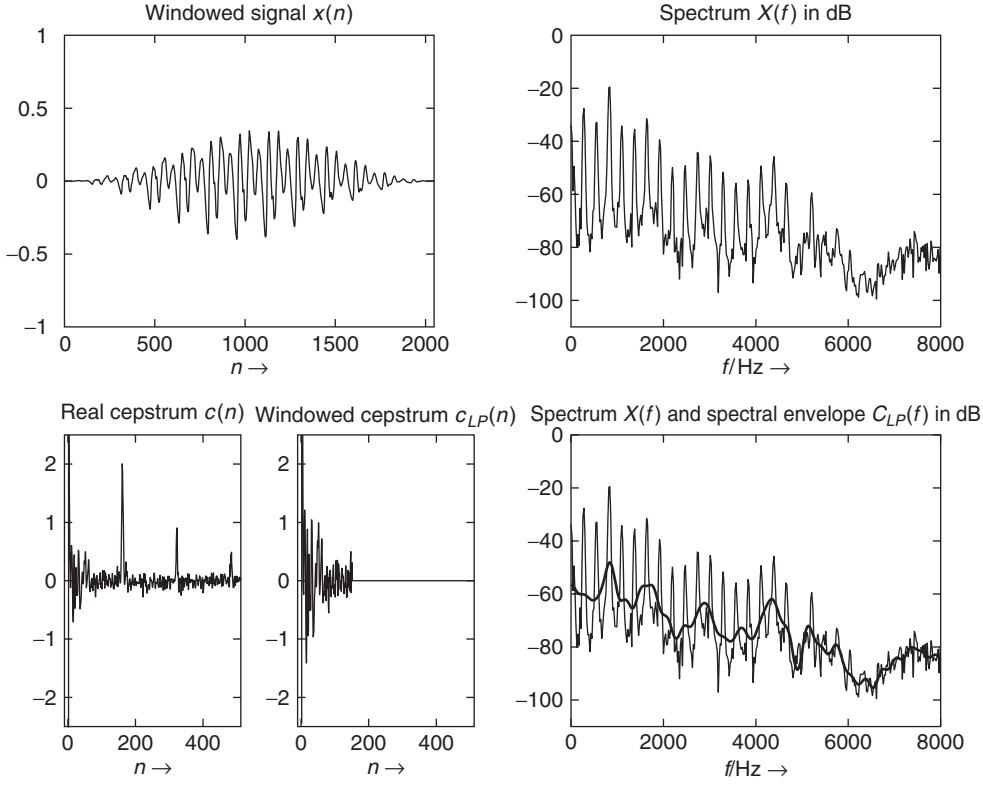


**Figure 8.12** Windowed signal segment, spectrum (FFT length $N = 2048$), cepstrum, windowed cepstrum ($N_1 = 150$) and spectral envelope.

in the cepstrum $c(n)$ after the "lowpass" part can be considered to have vanished. The following M-file 8.5 demonstrates briefly the way a spectral envelope can be calculated via the real cepstrum.

**M-file 8.5** (UX_specenvceps.m)

```
% N1: cut quefrency
WLen = 2048;
w    = hanning(WLen, 'periodic');
buf  = y(offset:offset+WLen-1).*w;
f    = fft(buf)/(WLen/2);
flog = 20*log10(0.00001+abs(f));
subplot(2,1,1); plot(flog(1:WLen/2));

N1   = input('cut value for cepstrum');
cep  = ifft(flog);
cep_cut  = [cep(1); 2*cep(2:N1); cep(N1+1); zeros(WLen-N1-1,1)];
flog_cut = real(fft(cep_cut));
subplot(2,1,2); plot(flog_cut(1:WLen/2));
```

In this program `cep` represents the cepstrum (hence the IFFT of the log magnitude of the FFT). The vector `cep_cut` is the version of the cepstrum with all values over the cut index set to zero. Here, we use a programming short-cut: we also remove the negative time values (hence the second part of the FFT) and use only the real part of the inverse FFT. The time indices $n$ of the cepstrum $c(n)$ are also denoted as "quefrencies." The vector `flog_cut` is a smoothed version of `flog` and represents the spectral envelope derived by the cepstrum method. The only input value for the spectral envelope computation is the `cut` variable. This variable `cut` is homogeneous to a time in samples, and should be less than the period of the analyzed sound.

### Source-filter separation

The cepstrum method allows the separation of a signal $y(n) = x(n) * h(n)$, which is based on a source and filter model, into its source signal $x(n)$ and its impulse response $h(n)$. The discrete-time Fourier transform $Y(\mathrm{e}^{j\omega}) = X(\mathrm{e}^{j\omega}) \cdot H(\mathrm{e}^{j\omega})$ is the product of two spectra: one representing the filter frequency response $H(\mathrm{e}^{j\omega})$ and the other one the source spectrum $X(\mathrm{e}^{j\omega})$. Decomposing the complex values in terms of the magnitude and phase representation, one can make the strong assumption that the filter frequency response will be real-valued and the phase will be assigned to the source signal.

The key point here is to use the mathematical property of the logarithm $\log(a \cdot b) = \log(a) + \log(b)$. The real cepstrum method will perform a spectral-envelope estimation based on the magnitude according to

$$|Y(\mathrm{e}^{j\omega})| = |X(\mathrm{e}^{j\omega})| \cdot |H(\mathrm{e}^{j\omega})|$$

$$\log|Y(\mathrm{e}^{j\omega})| = \log|X(\mathrm{e}^{j\omega})| + \log|H(\mathrm{e}^{j\omega})|.$$

In musical terms separating $\log|X(\mathrm{e}^{j\omega})|$ from $\log|H(\mathrm{e}^{j\omega})|$ is to keep the slow variation of $\log|Y(\mathrm{e}^{j\omega})|$ as a filter and the rapid ones as a source. In terms of signal processing we would like to separate the low frequencies of the signal $\log|Y(\mathrm{e}^{j\omega})|$ from the high frequencies (see Figure 8.13).

The separation of source and filter can be achieved by weighting the cepstrum $c(n) = c_x(n) + c_h(n)$ with two window functions, namely the "lowpass window" $w_{LP}(n)$ and the complementary "highpass window" $w_{HP}(n)$. This weighting yields $c_x(n) = c(n) \cdot w_{HP}(n)$ and $c_h(n) = c(n) \cdot w_{LP}(n)$. The low time values (low "quefrencies") for lowpass filtering $\log|Y(\mathrm{e}^{j\omega})|$ give $\log|H(\mathrm{e}^{j\omega})|$ (spectral envelope in dB) and the high time values (higher

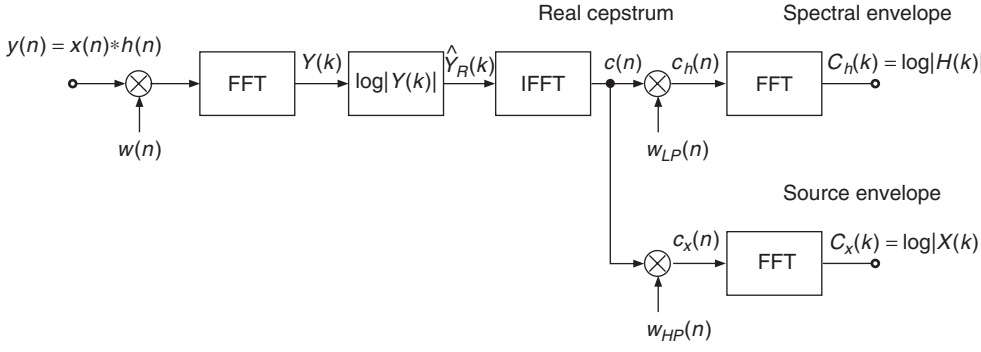


**Figure 8.13** Separating source and filter.

"quefrencies") for highpass filtering yield $\log|X(\mathrm{e}^{j\omega})|$ (source estimation). The calculation of $\exp(\log|H(\mathrm{e}^{j\omega})|)$ gives the magnitude response $|H(\mathrm{e}^{j\omega})|$. From this magnitude transfer function we can compute a zero-phase filter impulse response according to

$$h(n) = \mathrm{IFFT}\left[|H(k)|\right]. \tag{8.26}$$

The cepstrum method has a very good by-product. The combination of the highpass filtered version of the cepstrum and the initial phases from the FFT gives a spectrum that can be considered as the source spectrum in a source-filter model. This helps in designing audio effects based on such a source-filter model. The source signal $x(n)$ can be derived from the calculation of $\exp(C_x(k)) = |X(k)|$ and the initial phase taken from $Y(k) = |Y(k)|\mathrm{e}^{j\varphi_y(k)}$ by performing the IFFT of $|X(k)|\mathrm{e}^{j\varphi_y(k)}$ according to

$$x(n) = \mathrm{IFFT}\left[X(k)e^{j\varphi_y(k)}\right]. \tag{8.27}$$

For finding a good threshold between low and high quefrencies, one can make use of the fact that quefrencies are time variables, and whenever the sound is periodic, the cepstrum shows a periodicity corresponding to the pitch. Hence this value is the upper quefrency or upper time limit for the spectral envelope. A low value will smoothen the spectral envelope, while a higher value will include some of the harmonic or partial peaks in the spectral envelope.

### Hints and drawbacks

- "Lowpass filtering" is performed by windowing (zeroing values over a "cut quefrency"). This operation corresponds to filtering in the frequency domain with a $\frac{\sin(f)}{f}$ behavior. An alternative version is to use a smooth transition instead of an abrupt cut in the cepstrum domain.
- The cepstrum method will give a spectral estimation that smoothes the instantaneous spectrum. However, log values can go to $-\infty$ for a zero value in the FFT. Though this rarely happens with sounds coming from the real world, where the noise level prevents such values, a good prevention is the limitation of the log value. In our implementation the addition of a small value 0.00001 to the FFT values limits the lower log limit to $-100$ dB. Once again, an alternative to this bias applied to all values consists in truncating instead. Then, the following line from **MATLAB** code 8.5

```
flog=20*log10(0.00001+abs(f));
```

becomes

```
flog=20*log10(max(0.00001,abs(f)));
```

The artifact is not the same, as the spectrum shape is no more shifted, but clipped when below the threshold value.

- Though the *real cepstrum* is widely used, it is also possible to use the *complex cepstrum* to perform an estimation of the spectral envelope. In this case the spectral envelope will be defined by a complex FFT.

The spectral envelope estimation with the cepstrum has at least the three following limitations:

- The spectral envelope does not exactly link the spectral peaks (approximation): the spectral envelope estimated being a smoothing of the spectrum, nothing implies that the spectral envelope obtained should have the exact same values as the frequency peaks when it crosses them. And in fact, this does not happen.
- The spectral envelope has a behavior which depends on the choice of the cut-off quefrency (and number of cepstral coefficients): for too small values of the cut-off quefrency, the envelope becomes too smooth, whereas for too high values, the envelope follows the spectrum too much, showing anti-resonances between spectral peaks. Another situation where the cepstrum is not efficient is for high-pitched sounds: the number of spectral peaks is too small for the algorithm to perform well, and the resulting envelope may show erratic oscillations. In fact, the cut-off quefrency should be related to the signal's fundamental frequency, which is not necessarily known at that point in the algorithm.
- The spectral envelope is poorly estimated for high-pitched sounds, since the number of partials is small and widely spaced and the algorithm performs smoothing.

For those reasons, alternatives are needed to enhance the computation of spectral envelopes with the cepstrum, such as the iterative cepstrum and the discrete cepstrum. These two alternative techniques also provide cepstral coefficients that can then be used in the same way as the traditional cepstrum in order to perform source-filter separation and transformation.

### Iterative cepstrum

In order to enhance the computation of the spectral envelope, it is possible to use an iterative algorithm which calculates only the positive difference between the instantaneous spectrum and the estimated spectral envelope in each step. The principle is explained by Laroche in [Lar95, Lar98]: We first compute $E_m(n,k)$ the initial spectral envelope of $X(n,k)$ by the standard cepstrum technique (step $m = 0$). Then, we apply an iterative procedure, which for step $m$ does the following:

(1) Compute the positive difference between spectrum and spectral envelope $X_m(n,k) = \max(0, X(n,k) - E_m(n,k))$

(2) Compute the spectral envelope $E_{m+1}(n,k)$ by cepstrum using the $X_m(n,k)$ spectrum

(3) Stop when $|X_m(n,k) - E_{m+1}(n,k)| < \Delta$ (with $\Delta$ a given threshold).

The spectral envelope is finally given by $E_{m+1}(n,k)$.

While other refinements of the real cepstrum technique for estimating the spectral envelope (such as the discrete cepstrum) require the sound to be harmonic, the iterative cepstrum is similar to the real cepstrum technique: it works for any type of sound. Second, it improves it by providing a smooth envelope that goes through spectral peaks, provided that the threshold $\Delta$ is small and the maximum number of iterations is high. Because of the iterative procedure, the computation

complexity is much higher, and increases exponentially as $\Delta$ decreases. The following **MATLAB** code 8.6 implements this iterative technique.

**M-file 8.6** (UX_iterative_cepstrum.m)

```
% function [env,source] = iterative_cepstrum(FT,NF,order,eps,niter,Delta)
% [DAFXbook, 2nd ed., chapter 8]
% ==== This function computes the spectral enveloppe using the iterative
%      cepstrum method
% Inputs:
%   - FT [NF*1 | complex]     Fourier transform X(NF,k)
%   - NF [1*1 | int]          number of frequency bins
%   - order [1*1 | float]     cepstrum truncation order
%   - eps [1*1 | float]       bias
%   - niter [1*1 | int]       maximum number of iterations
%   - Delta [1*1 | float]     spectral envelope difference threshold
% Outputs:
%   - env [NFx1 | float]      magnitude of spectral enveloppe
%   - source [NFx1 | complex] complex source
function [env,source] = iterative_cepstrum(FT,NF,order,eps,niter,Delta)

%---- use data ----
fig_plot = 0;

%---- drawing ----
if(fig_plot), SR = 44100; freq = (0:NF-1)/NF*SR; figure(3); clf; end

%---- initializing ----
Ep = FT;

%---- computing iterative cepstrum ----
for k=1:niter
  flog     = log(max(eps,abs(Ep)));
  cep      = ifft(flog);      % computes the cepstrum
  cep_cut  = [cep(1)/2; cep(2:order); zeros(NF-order,1)];
  flog_cut = 2*real(fft(cep_cut));
  env      = exp(flog_cut);   % extracts the spectral shape
  Ep       = max(env, Ep);    % get new spectrum for next iteration
  %---- drawing ----
  if(fig_plot)
    figure(3); % clf  %uncomment to not superimpose curves
    subplot(2,1,1); hold on
    plot(freq, 20*log10(abs(FT)), 'k')
    h = plot(freq, 20*log10(abs(env)), 'k');
    set(h, 'Linewidth', 1, 'Color', 0.5*[1 1 1])
    xlabel('f / Hz \rightarrow'); ylabel('\rho(f) / d \rightarrow')
    title('Original spectrum and its enveloppe')
    axis tight; ax = axis; axis([0 SR/5 ax(3:4)])
    drawnow
  end
  %---- convergence criterion ----
  if(max(abs(Ep)) <= Delta), break; end
end

%---- computing source from enveloppe ----
source = FT ./ env;
```

**Discrete cepstrum**

The discrete-cepstrum technique by Galas and Rodet [GR90] can only be used for harmonic sounds. It requires a peak-picking algorithm first (best suited after additive analysis, see Spectral Processing Chapter 10), from which it computes an exact spectral envelope which links the maximum peaks. From the spectrum $X(n,k)$ of the current frame $L$ amplitudes $a(n,l)$ and corresponding frequencies $f(n,l)$ at the peaks in the spectrum are estimated. The cepstral coefficients are then derived so that the frequency response $\overline{F}(f)$ fits those $\{a(n,l),\ f(n,l)\}$ values for $l = 1, \ldots, L$. From a cepstral viewpoint, we look for the coefficients $c(n), n = 0, ..., N$ of a real cepstrum, which corresponding frequency response $\overline{F}(f)$ minimizes the criterion

$$\epsilon(n) = \sum_{l=1}^{L} w(l) || \log a(n,l) - \log |\overline{F}(f)| ||^2 \tag{8.28}$$

with $w(l), l = 1, \cdots, L$ a series of weights related to each partial harmonic. Note that a logarithmic scale is used (it can either be $20\log_{10}$, $\log_{10}$ or log) to ensure the constraint is related to the magnitude spectrum in dB.

Since the values of $\overline{F}(f_l)$ are from a symmetric spectrum (real signal), its Fourier transform can be reduced to a zero-phase cosine sum given by

$$\overline{F}(f) = c(0) + 2\sum_{i=1}^{N} c(i) \cdot \cos(2\pi f i). \tag{8.29}$$

The quantity $\epsilon$ we need to minimize can then be reformulated in matrix form according to

$$\epsilon = ||\mathbf{a} - \mathbf{Mc}||_W^2 = (\mathbf{a} - \mathbf{Mx})^T \mathbf{W} (\mathbf{a} - \mathbf{Mc}) \tag{8.30}$$

with the following notations:

$$\mathbf{a} = \begin{bmatrix} \log a(n,1) \\ log a(n,2) \\ \vdots \\ \log a(n,L) \end{bmatrix} \tag{8.31}$$

$$\mathbf{M} = \begin{bmatrix} 1 & 2\cos(2\pi f_1) & 2\cos(4\pi f_1) & \ldots & 2\cos(2N\pi f_1) \\ 1 & 2\cos(2\pi f_2) & 2\cos(4\pi f_2) & \ldots & 2\cos(2N\pi f_2) \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2\cos(2\pi f_L) & 2\cos(4\pi f_L) & \ldots & 2\cos(2N\pi f_L) \end{bmatrix} \tag{8.32}$$

$$\mathbf{c} = \begin{bmatrix} c(0) \\ c(1) \\ \vdots \\ c(L) \end{bmatrix} \tag{8.33}$$

$$\mathbf{W} = \begin{bmatrix} w(1) & 0 & \cdots & 0 \\ 0 & \log w(2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \log w(L) \end{bmatrix}. \tag{8.34}$$

The cepstral coefficient vector **c** can be estimated by

$$\mathbf{c} = \left(\mathbf{M}^T \mathbf{W}\, \mathbf{M}\right)^{-1} \mathbf{M}^T \mathbf{W}\, \mathbf{a}. \tag{8.35}$$

The following **MATLAB** code 8.7 implements the discrete cepstrum computation.

**M-file 8.7** (UX_discrete_cepstrum_basic.m)

```
% function cep = discrete_cepstrum_basic(F, A, order)
% [DAFXbook, 2nd ed., chapter 8]
% ==== This function computes the discrete spectrum regardless of
%      matrix conditionning and singularity.
% Inputs:
%   - A [1*L | float]        harmonic partial amplitudes
%   - F [1*L | float]        harmonic partial frequencies
%   - order [1*1 | int]      number of cepstral coefficients
% Outputs:
%   - cep [1xorder | float]  cepstral coefficients
function cep = discrete_cepstrum_basic(F, A, order)

%---- initialize matrices and vectors
L = length(A);
M = zeros(L,order+1);
M = zeros(L,L);
R = zeros(order+1,1);
W = zeros(L,L);

for i=1:L
  M(i,1) = 0.5;
  for k=2:order+1
    M(i,k) = cos(2*pi*(k-1)*F(i));
  end
  W(i,i) = 1; % weights = 1 by default
end
M = 2.*M;

%---- compute the solution, regardless of matric conditioning
Mt = transpose(M);
MtWMR = Mt * W * M;
cep = inv(MtWMR) * Mt * W * log(A);
```

Generally, the $\mathbf{M}^T\mathbf{WM}$ matrix is poorly conditioned or even singular, when there are a smaller number of constraints than parameters $L$. Since $L$ corresponds to the number of harmonic peaks in the spectrum, it can be quite small for high-pitched sounds, resulting in such problems. Two improvements will be introduced.

### Regularized discrete cepstrum

The criterion error $\epsilon$, as given in (8.28), may be replaced by a composite that balances between the constraint equation $\epsilon(n)$ and a regularization criterion, which computes the energy of the spectral envelope derivative [CLM95]. This regularization criterion is expressed as $\int_0^1 \left[\frac{d \log |F(n,f)|}{df}\right]^2 \mathrm{d}f$.

The new criterion is then given as

$$\epsilon_{reg}(n) = (1-\lambda)\epsilon(n) + \lambda \left( \int_0^1 \left[ \frac{\mathrm{d}\log|F(n,f)|}{\mathrm{d}f} \right]^2 \mathrm{d}f \right) \tag{8.36}$$

(NB: in [CLM95], there is no $(1-\lambda)$ coefficient for the first term.) From there, it is then easy to show that

$$\left( \int_0^1 \left[ \frac{\mathrm{d}\log|F(n,f)|}{\mathrm{d}f} \right]^2 \mathrm{d}f \right) = \mathbf{c}^T \mathbf{R} \mathbf{c} \tag{8.37}$$

with $\mathbf{R}$ the diagonal matrix which elements are $8\pi^2[0, 1, 2^2, 3^3, ..., L^2]$. From this, we can derive the solution given by

$$\mathbf{c} = \left( \mathbf{M}^T \mathbf{W} \mathbf{M} + \frac{\lambda}{1-\lambda} \mathbf{R} \right)^{-1} \mathbf{M}^T \mathbf{W}\, \mathbf{a}. \tag{8.38}$$

This formulation only differs from [CLM95] by the ratio $\frac{\lambda}{1-\lambda}$, because we prefer to use a value of $\lambda \in [0, 1[$. The corresponding M-file 8.8 shows the implementation.

**M-file 8.8** (UX_discrete_cepstrum_reg.m)

```
% function cep = discrete_cepstrum_reg(F, A, order, lambda)
% [DAFXbook, 2nd ed., chapter 8]
% ==== This function computes the discrete spectrum using a
%      regularization function
% Inputs:
%   - A [1*L | float]       harmonic partial amplitudes
%   - F [1*L | float]       harmonic partial frequencies
%   - order [1*1 | int]     number of cepstral coefficients
%   - lambda [1*2 | float]  weighting of the perturbation, in [0,1[
% Output:
%   - cep [1*order | float] cepstral coefficients
function cep = discrete_cepstrum_reg(F, A, order, lambda)

%---- reject incorrect lambda values
if lambda>=1 | lambda <0
  disp('Error: lambda must be in [0,1[')
  cep = [];
  return;
end

%---- initialize matrices and vectors
L = length(A);
M = zeros(L,order+1);
R = zeros(order+1,1);
for i=1:L
  M(i,1) = 1;
  for k=2:order+1
    M(i,k) = 2 * cos(2*pi*(k-1)*F(i));
  end
end

%---- initialize the R vector values
```

```
coef = 8*(pi^2);
for k=1:order+1
  R(k,1) = coef * (k-1)^2;
end

%---- compute the solution
Mt = transpose(M);
MtMR = Mt*M + (lambda/(1.-lambda))*diag(R);
cep = inv(MtMR) * Mt*log(A);
```

### Less strict and jittered envelope

A second improvement to relax the constraint consists in adding a number of random points around the peaks, and increasing the smoothing criterion. Then, the spectral envelope is more attracted to the region of the spectral peaks, but does not have to go through them exactly. The following **MATLAB** code 8.9 implements this random discrete cepstrum computation.

**M-file 8.9** (UX_discrete_cepstrum_random.m)

```
% function cep = discrete_cepstrum_random(F, A, order, lambda, Nrand, dev)
% [DAFXbook, 2nd ed., chapter 8]
% ==== This function computes the discrete spectrum using multiples of each
%      peak to which a small random amount is added, for better smoothing.
% Inputs:
%   - A [1*L | float]        harmonic partial amplitudes
%   - F [1*L | float]        harmonic partial frequencies
%   - order [1*1 | int]      number of cepstral coefficients
%   - lambda [1v2 | float]   weighting of the perturbation, in [0,1[
%   - Nrand [1*1 | int]      nb of random points generated per (Ai,Fi) pair
%   - dev  1*2 | float]     deviation of random points, with Gaussian
% Outputs:
%   - cep [1*order |â float] cepstral coefficients
function cep = discrete_cepstrum_random(F, A, order, lambda, Nrand, dev)

if lambda>=1 | lambda <0 % reject incorrect lambda values
  disp('Error: lambda must be in [0,1[')
  cep = [];
  return;
end

%---- generate random points ----
L = length(A);
new_A = zeros(L*Nrand,1);
new_F = zeros(L*Nrand,1);
for k=1:L
  sigA = dev * A(k);
  sigF = dev * F(k);
  for l=1:L
    new_A((l-1)*Nrand+1) = A(l);
    new_F((l-1)*Nrand+1) = F(l);
    for n=1:Nrand
      new_A((l-1)*Nrand+n+1) = random('norm', A(l), sigA);
      new_F((l-1)*Nrand+n+1) = random('norm', F(l), sigF);
    end
  end
end
```

```
%---- initialize matrices and vectors
L = length(new_A);
M = zeros(L,order+1);
R = zeros(order+1,1);
for i=1:L
  M(i,1) = 1;
  for k=2:order+1
    M(i,k) = 2 * cos(2*pi*(k-1)*new_F(i));
  end
end
%---- initialize the R vector values
coef = 8*(pi^2);
for k=1:order+1,
  R(k,1) = coef * (k-1)^2;
end
%---- compute the solution
Mt = transpose(M);
MtMR = Mt*M + (lambda/(1.-lambda))*diag(R);
cep = inv(MtMR) * Mt*log(new_A);
```

In conclusion, the cepstrum method allows both the separation of the audio signal into a source signal and a filter (spectral envelope) and, as a by-product, the estimation of the fundamental frequency, which has been published in [Nol64] and later reported in [Sch99]. Various refinements exist, such as the iterative cepstrum and the discrete cepstrum with an increasing computational complexity.

## 8.3 Source-filter transformations

### 8.3.1 Vocoding or cross-synthesis

The term vocoder has different meanings. One is "voice-coding" and refers directly to speech synthesis. Another meaning for this term is the phase vocoder, which refers to the short-time Fourier transform, as discussed in Section 7.2. The last meaning is the one of the musical instrument named the *Vocoder* and this is what this paragraph is about: vocoding or cross-synthesis.

This effect takes two sound inputs and generates a third one which is a combination of the two input sounds. The general idea is to combine two sounds by "spectrally shaping" the first sound by the second one and preserving the pitch of the first sound. A variant and improvement are the removal of the spectral envelope of the initial sound (also called whitening) before filtering with the spectral envelope of the second one. This implies the ability to extract a spectral envelope evolving with time and to apply it to a signal.

Although spectral estimation is well represented by its amplitude versus frequency representation, most often it is the filter representation that can be a help in the application of this spectral envelope: the channel vocoder uses the weighted sum of filtered bandpass signals, the LPC calculates an IIR filter, and even the cepstrum method can be seen as a circular convolution with an FIR filter. As this vocoding effect is very important and can give different results depending on the technique used, we will introduce these three techniques applied to the vocoding effect.

#### Channel vocoder

This technique uses two banks of filters provided by the channel vocoder (see Figure 8.14), as well as the RMS (root mean square) values associated with these channels. For each channel the

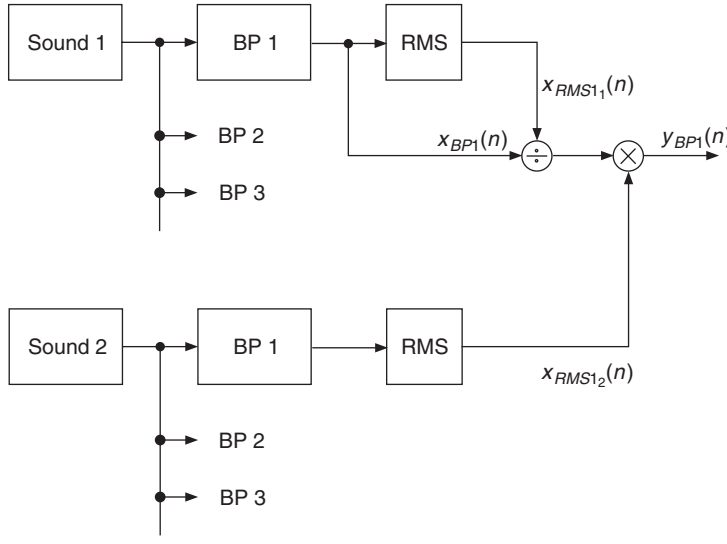


**Figure 8.14** Basic principle of spectral mutations.

bandpass signal is divided by the RMS value of this channel, and then multiplied by the RMS value of the other sound. The mathematical operation is given by

$$y_{BP_i}(n) = x_{BP_i}(n) \cdot \frac{x_{RMSi_2}(n)}{x_{RMSi_1}(n)}, \tag{8.39}$$

where $x_{RMSi_1}(n)$ and $x_{RMSi_2}(n)$ represent the RMS values in channel $i$ for the two sounds. One should be careful with the division. Of course divisions by zero should be avoided, but there should also be a threshold for avoiding the amplification of noise. This works well when sound 2 has a strong spectral envelope, for example, a voice. The division by $x_{RMSi_1}(n)$ can be omitted or replaced by just modifying the amplitude of each band. Sound 1 can also be a synthetic sound (pulse, sawtooth, square).

The following M-file 8.10 demonstrates a cross-synthesis between two sounds based on the channel vocoder implemented by IIR filters.

**M-file 8.10** (UX_cross_synthesis_CV.m)

```
% UX_cross_synthesis_CV.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a cross-synthesis with channel vocoder
clear

%----- setting user data -----
[DAFx_in_sou,FS] = wavread('moore_guitar'); % signal for source extraction
DAFx_in_env      = wavread('toms_diner');   % signal for spec. env. extraction
ly       = min(length(DAFx_in_sou), length(DAFx_in_env));  % min signal length
DAFx_out = zeros(ly,1);      % result signal
r        = 0.99;             % sound output normalizing ratio
lp       = [1, -2*r, +r*r];  % low-pass filter used
epsi     = 0.00001;

%----- init bandpass frequencies
f0 = 10;              % start freq in Hz
f0 = f0/FS *2;        % normalized freq
```

```
fac_third = 2^(1/3);  % freq factor for third octave
K  = floor(log(1/f0) / log(fac_third)); % number of bands

%----- performing the vocoding or cross synthesis effect -----
fprintf(1, 'band number (max. %i):\n', K);
tic
for k=1:K
  fprintf(1, '%i ', k);
  f1 = f0 * fac_third;            % upper freq of bandpass
  [b, a] = cheby1(2, 3, [f0 f1]); % Chebyshev-type 1 filter design
  f0 = f1;                        % start freq for next band
  %-- filtering the two signals --
  z_sou    = filter(b, a, DAFx_in_sou);
  z_env    = filter(b, a, DAFx_in_env);
  rms_env  = sqrt(filter(1, lp, z_env.*z_env)); % RMS value of sound 2
  rms_sou  = sqrt(epsi+filter(1, lp, z_sou.*z_sou));  % with whitening
%  rms_sou = 1.;                              % without whitening
  DAFx_out = DAFx_out + z_sou.*rms_env./rms_sou; % add result to output buffer
end
fprintf(1, '\n');
toc

%----- playing and saving output sound -----
soundsc(DAFx_out, FS)
DAFx_out_norm = r * DAFx_out/max(abs(DAFx_out)); % scale for wav output
wavwrite(DAFx_out_norm, FS, 'CrossCV')
```

This program performs bandpass filtering inside a loop. Precisely, Chebychev type 1 filters are used, which are IIR filters with a ripple of 3 dB in the passband. The bandwidth is chosen as one-third of an octave, hence the 0.005 to 0.0063 window relative to half of the sampling rate in **MATLAB**'s definition. Then sound 1 and sound 2 are filtered, and the RMS value of the filtered sound 2 is extracted: `z2` is squared, filtered by a two pole filter on the x axis, and its square root is taken. This RMS2 value serves as a magnitude amplifier for the `z1` signal, which is the filtered version of sound 1. This operation is repeated every one-third of an octave by multiplying the frequency window, which is used for the definition of the filter, by 1.26 (3rd root of 2). A whitening process can be introduced by replacing line `rms1 = 1.;` with `rms1 = epsi + norm(filter(1, lp, z1.*z1), 2);`. A small value `epsi` (0.0001) is added to RMS1 to avoid division by zero. If `epsi` is greater, the whitening process is attenuated. Thus this value can be used as a control for whitening.

### Linear prediction

Cross-synthesis between two sounds can also be performed using the LPC method [Moo79, KAZ00]. One filter removes the spectral envelope of the first sound and the spectral envelope of the second sound is used to filter the excitation signal of the first sound, as shown in Figure 8.15.

The following M-file 8.11 performs cross-synthesis based on the LPC method. The prediction coefficients of sound 1 are used for an FIR filter to whiten the original sound. The prediction coefficients of sound 2 are used in the feedback path of a synthesis filter, which performs filtering of the excitation signal of sound 1 with the spectral envelope derived from sound 2.

**M-file 8.11** (`UX_cross_synthesis_LPC.m`)

```
% UX_cross_synthesis_LPC.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a cross-synthesis with LPC
clear;
```

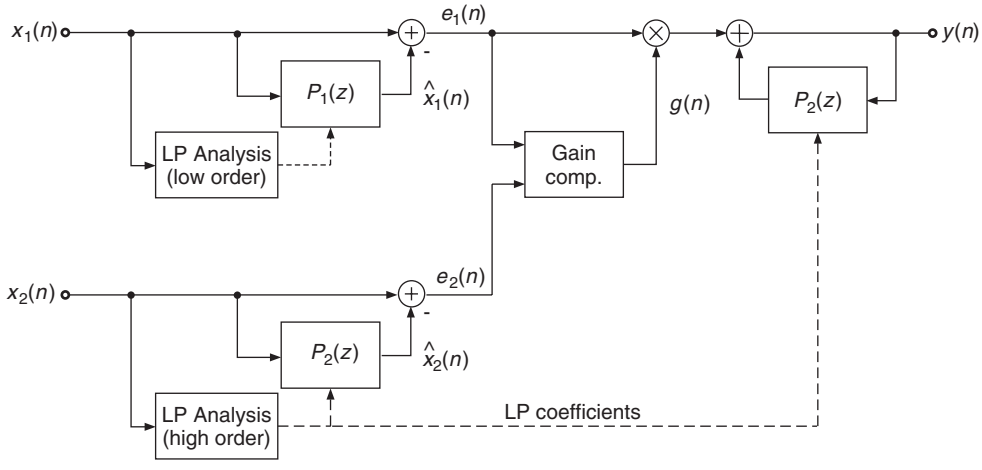


**Figure 8.15** Cross-synthesis with LPC.

```
%----- user data -----
[DAFx_in_sou, FS] = wavread('moore_guitar.wav');  % sound 1: source/excitation
DAFx_in_env   = wavread('toms_diner.wav');        % sound 2: spectral env.
long          = 400;        % block length for calculation of coefficients
hopsize       = 160;        % hop size (is 160)
env_order     = 20          % order of the LPC for source signal
source_order  = 6           % order of the LPC for excitation signal
r             = 0.99;       % sound output normalizing ratio

%----- initializations -----
ly = min(length(DAFx_in_sou), length(DAFx_in_env));
DAFx_in_sou = [zeros(env_order, 1); DAFx_in_sou; ...
  zeros(env_order-mod(ly,hopsize),1)] / max(abs(DAFx_in_sou));
DAFx_in_env = [zeros(env_order, 1); DAFx_in_env; ...
  zeros(env_order-mod(ly,hopsize),1)] / max(abs(DAFx_in_env));
DAFx_out = zeros(ly,1);     % result sound
exc      = zeros(ly,1);     % excitation sound
w        = hanning(long, 'periodic');   % window
N_frames = floor((ly-env_order-long)/hopsize); % number of frames

%----- Perform ross-synthesis -----
tic
for j=1:N_frames
  k        = env_order + hopsize*(j-1);     % offset of the buffer
  %!!! IMPORTANT: function "lpc" does not give correct results for MATLAB 6 !!!
  [A_env, g_env]  = calc_lpc(DAFx_in_env(k+1:k+long).*w, env_order);
  [A_sou, g_sou]  = calc_lpc(DAFx_in_sou(k+1:k+long).*w, source_order);
  gain(j) = g_env;
  ae      = - A_env(2:env_order+1); % LPC coeff. of excitation
  for n=1:hopsize
    excitation1   = (A_sou/g_sou) * DAFx_in_sou(k+n:-1:k+n-source_order);
    exc(k+n)      = excitation1;
    DAFx_out(k+n) = ae * DAFx_out(k+n-1:-1:k+n-env_order)+g_env*excitation1;
  end
end
toc

%----- playing and saving output signal -----
DAFx_out      = DAFx_out(env_order+1:length(DAFx_out)) / max(abs(DAFx_out));
```

```
soundsc(DAFx_out, FS)
DAFx_out_norm = r * DAFx_out/max(abs(DAFx_out)); % scale for wav output
wavwrite(DAFx_out_norm, FS, 'CrossLPC')
```

### Cepstrum

Signal processing based on cepstrum analysis is also called homomorphic signal processing [OS75, PM96]. We have seen that we can derive the spectral envelope (in dB) with the cepstrum technique. Reshaping a sound is achieved by whitening (filtering) a sound with the inverse spectral envelope $1/|H_1(f)|$ and then filtering with the spectral envelope $|H_2(f)|$ of the second sound (see Figure 8.16). The series connection of both filters leads to a transfer function $H_2(f)/H_1(f)$. By taking the logarithm according to $\log|H_2(f)|/|H_1(f)| = \log|H_2(f)| - \log|H_1(f)|$, the filtering operation is based on the difference of the two spectral envelopes. The first spectral envelope performs the whitening by inverse filtering and the second spectral envelope introduces the formants. The inverse filtering of the input sound 1 and subsequent filtering with spectral envelope of sound 2 can be performed in one step by the fast convolution technique.

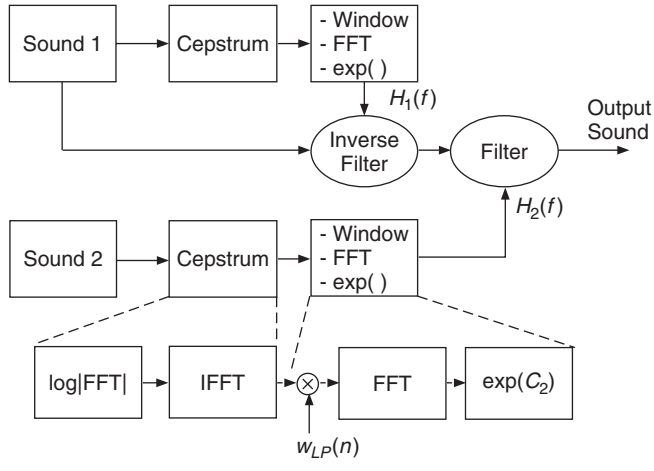


**Figure 8.16** Basic principle of homomorphic cross-synthesis. The spectral envelopes of both sounds are derived by the cepstrum method.

Here we present the core of a program given by M-file 8.12 that uses the spectral envelope of a sound (number 2) superimposed on a sound (number 1). Though musically very effective, this first program does not do any whitening of sound 1.

**M-file 8.12** (UX_cross_synthesis_cepstrum.m)

```
% UX_cross_synthesis_cepstrum.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a cross-synthesis with cepstrum
clear all; close all

%----- user data -----
% [DAFx_sou, SR] = wavread('didge_court.wav');  % sound 1: source/excitation
% DAFx_env       = wavread('la.wav');           % sound 2: spectral enveloppe
[DAFx_sou, SR] = wavread('moore_guitar.wav'); % sound 1: source/excitation
DAFx_env  = wavread('toms_diner.wav');        % sound 2: spectral enveloppe
```

```
s_win     = 1024;   % window size
n1        = 256;    % step increment
order_sou = 30;     % cut quefrency for sound 1
order_env = 30;     % cut quefrency for sound 2
r         = 0.99;   % sound output normalizing ratio

%----- initialisations -----
w1          = hanning(s_win, 'periodic');  % analysis window
w2          = w1;               % synthesis window
hs_win      = s_win/2;          % half window size
grain_sou   = zeros(s_win,1);   % grain for extracting source
grain_env   = zeros(s_win,1);   % grain for extracting spec. enveloppe
pin         = 0;                % start index
L           = min(length(DAFx_sou),length(DAFx_env));
pend        = L - s_win;        % end index
DAFx_sou    = [zeros(s_win, 1); DAFx_sou; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_sou));
DAFx_env    = [zeros(s_win, 1); DAFx_env; ...
  zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_env));
DAFx_out    = zeros(L,1);

%----- cross synthesis -----
while pin<pend
  grain_sou = DAFx_sou(pin+1:pin+s_win).* w1;
  grain_env = DAFx_env(pin+1:pin+s_win).* w1;
  %=======================================
  f_sou     = fft(grain_sou);             % FT of source
  f_env     = fft(grain_env)/hs_win;      % FT of filter
  %---- computing cepstrum ----
  flog      = log(0.00001+abs(f_env));
  cep       = ifft(flog);                 % cepstrum of sound 2
  %---- liftering cepstrum ----
  cep_cut   = zeros(s_win,1);
  cep_cut(1:order_sou) = [cep(1)/2; cep(2:order_sou)];
  flog_cut  = 2*real(fft(cep_cut));
  %---- computing spectral enveloppe ----
  f_env_out = exp(flog_cut);              % spectral shape of sound 2
  grain     = (real(ifft(f_sou.*f_env_out))).*w2; % resynthesis grain
  % =======================================
  DAFx_out(pin+1:pin+s_win) = DAFx_out(pin+1:pin+s_win) + grain;
  pin       = pin + n1;
end

%----- 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, SR);
DAFx_out_norm = r * DAFx_out/max(abs(DAFx_out)); % scale for wav output
wavwrite(DAFx_out_norm, SR, 'CrossCepstrum')
```

In this program `n1` represents the analysis step increment (or hop size), and `grain_sou` and `grain_env` windowed buffers of `DAFx_in_sou` and `DAFx_in_env`. `f_sou` is the FFT of `grain_sou` and `f_env` is the spectral envelope derived from the FFT of `grain_env`. Although this algorithm performs a circular convolution, which theoretically introduces time aliasing, the resulting sound does not have artifacts.

Whitening DAFx_in_sou before processing it with the spectral envelope of DAFx_in_env can be done in a combined step: we calculate the spectral envelope of DAFx_in_sou and subtract it (in dB) from the spectral envelope of DAFx_in_env. The following code lines given by M-file 8.13 perform a whitening of DAFx_in_sou and a cross-synthesis with DAFx_in_env.

**M-file 8.13** (UX_cross_synthesis_cepstrum_whitening.m)

```
%=========================================
f_sou     = fft(grain_sou);               % FT of source
f_env     = fft(grain_env)/hs_win;        % FT of filter
%---- computing cepstra ----
flog_sou      = log(0.00001+abs(f_sou));
cep_sou       = ifft(flog_sou);           % cepstrum of sound 1 / source
flog_env      = log(0.00001+abs(f_env));
cep_env       = ifft(flog_env);           % cepstrum of sound 2 / env.
%---- liftering cepstra ----
cep_cut_env   = zeros(s_win,1);
cep_cut_env(1:order_env) = [cep_env(1)/2; cep_env(2:order_env)];
flog_cut_env  = 2*real(fft(cep_cut_env));
cep_cut_sou   = zeros(s_win,1);
cep_cut_sou(1:order_sou) = [cep_sou(1)/2; cep_sou(2:order_sou)];
flog_cut_sou  = 2*real(fft(cep_cut_sou));
%---- computing spectral enveloppe ----
f_env_out = exp(flog_cut_env - flog_cut_sou);   % whitening with source
grain     = (real(ifft(f_sou.*f_env_out))).*w2; % resynthesis grain
% =========================================
```

In this program flog_cut_sou and flog_cut_env represent (in dB) the spectral envelopes derived from grain_sou and grain_env for a predefined cut quefrency. Recall that this value is given in samples. It should normally be below the pitch period of the sound, and the lower it is, the more smoothed the spectral envelope will be.

### 8.3.2 Formant changing

This effect produces a "Donald Duck" voice without any alteration of the fundamental frequency. It can be used for performing an alteration of a sound whenever there is a formant structure. However, it can also be used in conjunction with pitch-shifted sounds for recovering a natural formant structure (see Section 8.3.4).

The musical goal is to remove the spectral envelope from one sound and to impose another one, which is a warped version of the first one, as shown in Figure 8.17, where the signal processing is also illustrated. This means that we have to use a spectral correction that is a ratio of the two spectral envelopes. In this way the formants, if there are any, are changed according to this warping function. For example, a transposition of the spectral envelope by a factor of two will give a "Donald Duck" effect without time stretching. This effect can also be seen as a particular case of cross-synthesis, where the modifier comes from an interpolated version of the original sound. Though transposition of the spectral envelope is classical, other warping functions can be used.

From a signal-processing point of view the spectral correction for formant changing can be seen in the frequency domain as $H_2(f)/H_1(f)$. First divide by the spectral envelope $H_1(f)$ of the input sound and then multiply by the frequency scaled spectral envelope $H_2(f)$. In the cepstrum domain the operation $H_2(f)/H_1(f)$ leads to the subtraction $C_2(f) - C_1(f)$, where $C(f) = \log|H(f)|$. When using filters for time-domain processing, the transfer function is $H_2(f)/H_1(f)$ (see Figure 8.17). We will shortly describe three different methods for the estimation of the two spectral envelopes.

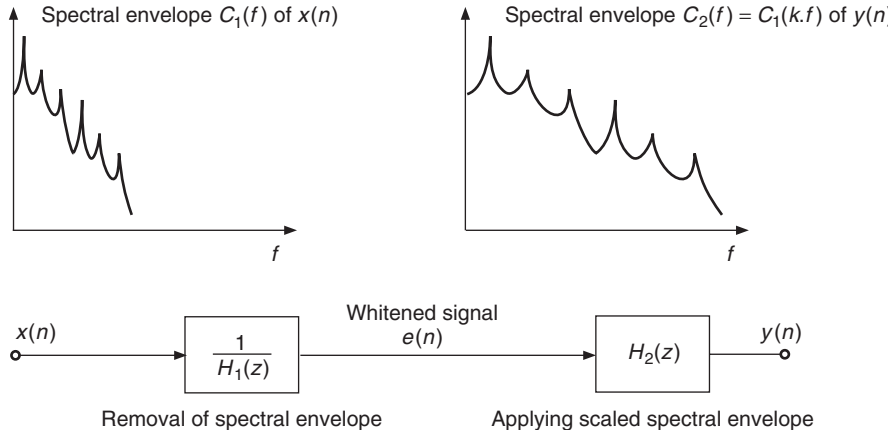


**Figure 8.17** Formant changing by frequency scaling the spectral envelope and time-domain processing.

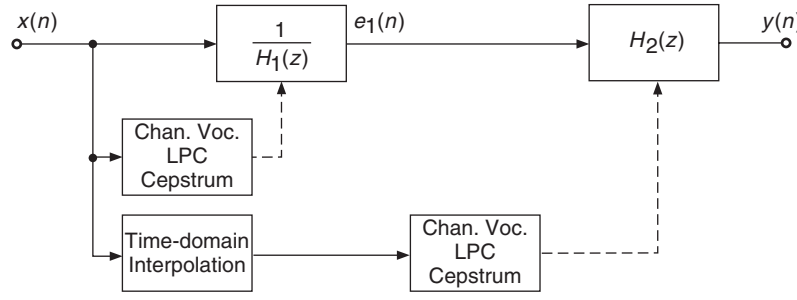


**Figure 8.18** Formant changing by time-domain processing.

### Interpolation of the input signal

The spectral envelopes $C_1(f)$ and $C_2(f)$, or filters $H_1(f)$ and $H_2(f)$ can be obtained by different techniques. If $C_2(f)$ is a frequency-scaled version of $C_1(f)$, one can calculate the spectral envelope $C_2(f)$ from the analysis of a transposed version of the initial signal, as shown in Figure 8.18. The transposed version is obtained by time-domain interpolation of the input signal. The channel vocoder, LPC and the cepstrum method, allow the estimation of either the spectral envelope or the corresponding filter. One must take care to keep synchronicity between the two signals. This can be achieved by changing the hop size according to this ratio. The algorithm works as follows:

- Whitening: filter the input signal with frequency response $\frac{1}{H_1(f)}$ or subtract the input spectral envelope $C_1(f) = \log|H_1(f)|$ from the log of the input magnitude spectrum.
- The filter $H_1(f)$ or the spectral envelope $C_1(f)$ is estimated from the input signal.
- Formant changing: apply the filter with frequency response $H_2(f)$ to the whitened signal or add the spectral envelope $C_2(f) = \log|H_2(f)|$ to the whitened log of the input magnitude spectrum.
- The filter $H_2(f)$ or the spectral envelope $C_2(f)$ is estimated from the interpolated input signal.

Formant changing based on the cepstrum analysis is shown in Figure 8.19. The spectral correction is calculated from the difference of the log values of the FFTs, of both the input signal and the interpolated input signal. This log difference is transformed to the cepstrum domain, lowpass weighted and transformed back by the exponential to the frequency domain. Then, the filtering of the input signal with the spectral correction filter $H_2(z)/H_1(z)$ is performed in the frequency domain. This fast convolution is achieved by multiplication of the corresponding Fourier transforms of the input signal and the spectral correction filter. The result is transformed back to the time domain by an IFFT yielding the output signal. An illustrative example is shown in Figure 8.20. The M-file 8.14 demonstrates this technique.

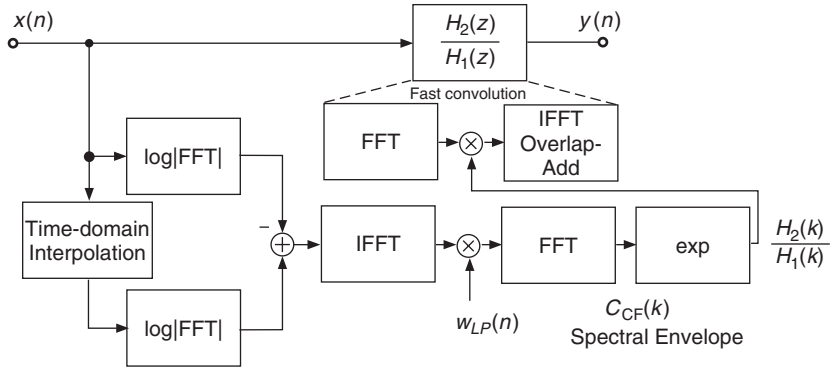


**Figure 8.19** Formant changing by frequency-domain processing: cepstrum analysis, spectral correction filter computation and fast convolution.

**M-file 8.14** (UX_fmove_cepstrum.m)

```
% UX_fmove_cepstrum.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a formant warping with cepstrum
clear; clf;
%----- user data -----
fig_plot = 0;     % use any value except 0 or [] to plot figures
[DAFx_in, SR] = wavread('la.wav'); % sound file
warping_coef  = 2.0;
n1        = 512;    % analysis hop size
n2        = n1;     % synthesis hop size
s_win     = 2048;   % window length
order     = 50;     % cut quefrency
r         = 0.99;   % sound output normalizing ratio
%----- initializations -----
w1        = hanning(s_win,  'periodic'); % analysis window
w2        = w1;              % synthesis window
hs_win    = s_win/2;         % half window size
L         = length(DAFx_in); % signal length
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(L,1);       % output signal
t         = 1 + floor((0:s_win-1)*warping_coef); % apply the warping
lmax      = max(s_win,t(s_win));
```

```
tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = L - s_win;
if(fig_plot)
  pin  = 6500;
  pend = pin + s_win;
end
while pin<pend
   grain  = DAFx_in(pin+1:pin+s_win).* w1;
%=========================================
   f      = fft(grain)/hs_win;         % spectrum of grain
   flogs  = 20*log10(0.00001+abs(f));  % log|X(k)|

   grain1 = DAFx_in(pin+t).* w1;       % linear interpolation of grain
   f1     = fft(grain1)/hs_win;        % spectrum of interpolated grain
   flogs1 = 20*log10(0.00001 + abs(f1)); % log|X1(k)|
   flog   = log(0.00001+abs(f1)) - log(0.00001+abs(f));
   cep    = ifft(flog);                % cepstrum
   cep_cut = [cep(1)/2; cep(2:order); zeros(s_win-order,1)];

   corr   = exp(2*real(fft(cep_cut))); % spectral shape
   grain  = (real(ifft(f.*corr))).*w2;

   fout   = fft(grain);
   flogs2 = 20*log10(0.00001+abs(fout));

   %----- figures for real-time spectral shape up to SR/2 -----
   if(fig_plot)
     range = (1:hs_win/2);
     subplot(3,1,1);  plot((range)*SR/s_win, flogs(range));
     title('a) original spectrum'); drawnow;
     subplot(3,1,2);  plot((range)*SR/s_win, flogs1(range));
     title('b) spectrum of time-scaled signal');
     subplot(3,1,3);  plot((range)*SR/s_win, flogs2(range));
     title('c) formant changed spectrum');
     xlabel('f in Hz \rightarrow');
     drawnow
   end
% =========================================
   DAFx_out(pout+1:pout+s_win) = DAFx_out(pout+1:pout+s_win) + grain;
   pin  = pin + n1;
   pout = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc
%----- listening and saving the output -----
soundsc(DAFx_out, SR);
DAFx_out_norm = r * DAFx_out/max(abs(DAFx_out)); % scale for wav output
wavwrite(DAFx_out_norm, SR, 'la_fmove.wav');
```

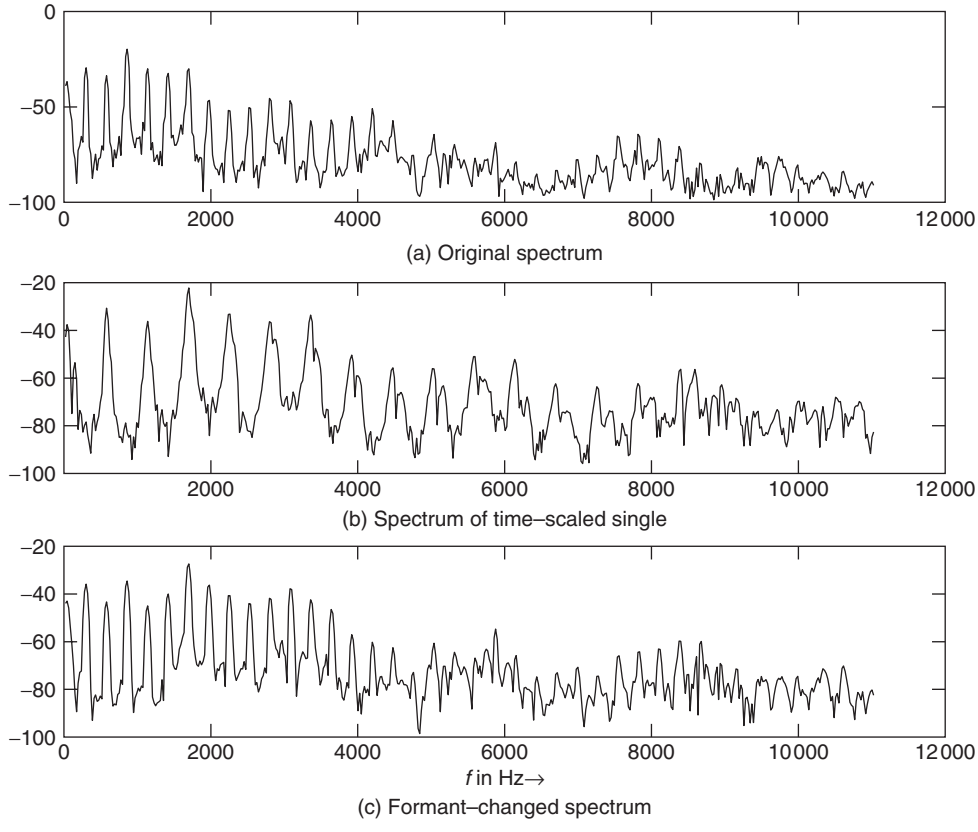


**Figure 8.20** Example of formant changing: the upper plot shows the input spectrum and the middle plot the spectrum of the interpolated signal. The lower plot shows the result of the formant-changing operation, where the spectral envelope of the interpolated spectrum can be noticed.

### Interpolation or scaling of the spectral envelope

The direct warping is also possible, for example, by using the interpolation of the spectral envelope derived from a cepstrum technique: $C_2(f) = C_1(k \cdot f)$ or $C_2(f/k) = C_1(f)$. There are, however, numerical limits: the cepstrum method uses an FFT and frequencies should be below half of the sampling frequency. Thus, if the transposition factor is greater than one, we will get only a part of the initial envelope. If the transposition factor is less than one, we will have to zero-pad the rest of the spectral envelope to go up to half of the sampling frequency. The block diagram for the algorithm using the cepstrum analysis method is shown in Figure 8.21. The following M-file 8.15 demonstrates this method.

**M-file 8.15** (UX_fomove_cepstrum.m)

```
% UX_fomove_cepstrum.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a formant warping with cepstrum
clear; clf;
```

```
%----- user data -----
fig_plot = 0;     % use any value except 0 or [] to plot figures
[DAFx_in, SR] = wavread('la.wav'); % sound file
warping_coef  = 2.0
n1       = 512;    % analysis hop size
n2       = n1;     % synthesis hop size
s_win    = 2048;   % window length
order    = 50;     % cut quefrency
r        = 0.99;   % sound output normalizing ratio

%----- initializations -----
w1       = hanning(s_win,  'periodic'); % analysis window
w2       = w1;              % synthesis window
hs_win   = s_win/2;         % half window size
L        = length(DAFx_in); % signal length
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(L,1);
x0       = floor(min((1+(0:hs_win)/warping_coef), 1+hs_win));
         % apply the warping
x        = [x0, x0(hs_win:-1:2)];% symmetric extension

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = L - s_win;

while pin<pend
   grain = DAFx_in(pin+1:pin+s_win).* w1;
%=========================================
   f     = fft(grain)/hs_win;
   flog  = log(0.00001+abs(f));
   cep   = ifft(flog);
   cep_cut   = [cep(1)/2; cep(2:order); zeros(s_win-order,1)];
   %---- flog_cut1|2 =  spectral shapes before/after formant move
   flog_cut1 = 2*real(fft(cep_cut));
   flog_cut2 = flog_cut1(x);
   corr  = exp(flog_cut2-flog_cut1);
   grain = (real(ifft(f.*corr))).*w2;
% =========================================
   DAFx_out(pout+1:pout+s_win) = DAFx_out(pout+1:pout+s_win) + grain;
   pin   = pin + n1;
   pout  = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
soundsc(DAFx_out, SR);
DAFx_out_norm = r * DAFx_out/max(abs(DAFx_out)); % scale for wav output
wavwrite(DAFx_out_norm, SR, 'la_fomove.wav');
```

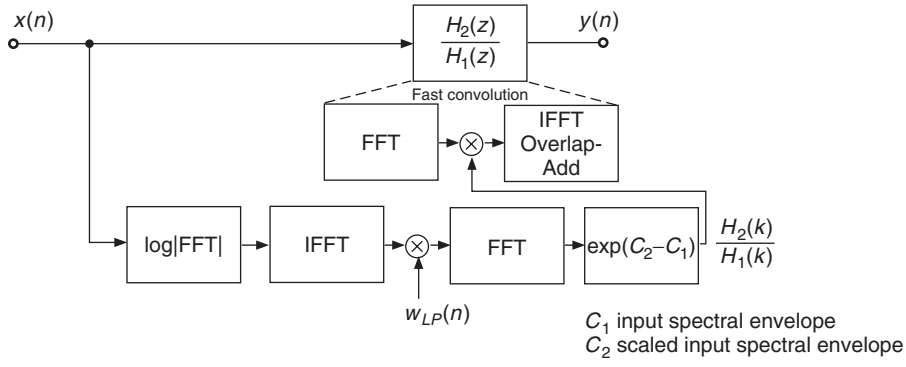


**Figure 8.21** Formant changing by scaling the spectral envelope.

### Direct warping of filters

A direct warping of the spectral envelope filter $H_1(z)$ to $H_2(z)$ is also possible. The warping of a filter transfer function can be performed by the allpass function $z^{-1} \rightarrow \frac{z^{-1}-\alpha}{1-\alpha z^{-1}}$. Substituting $z^{-1}$ in the transfer function $H_1(z)$ by $\frac{z^{-1}-\alpha}{1-\alpha z^{-1}}$ yields the warped transfer function $H_2(z)$. Further details on warping can be found in Chapter 11 and in [Str80, LS81, HKS+00].

## 8.3.3 Spectral interpolation

Spectral interpolation means that instead of mixing two sounds, we mix their excitation signals independently of their spectral envelopes, as shown in Figure 8.22. If we have the decomposition of sound grains in the frequency domain according to $E(f) \cdot H(f)$, where $E(f)$ represents the Fourier transform of the excitation and $H(f)$ is the spectral envelope ($H(f) = \exp[C(f)]$), we

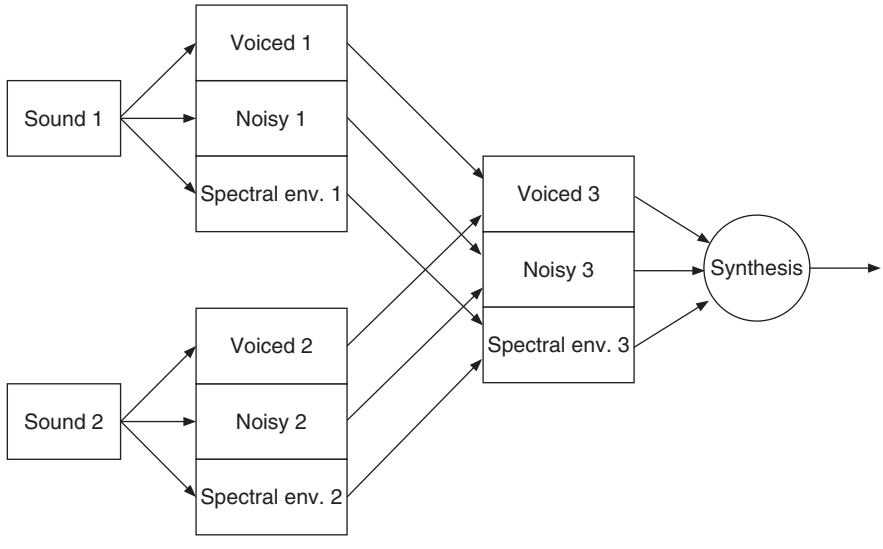


**Figure 8.22** A possible implementation of spectral interpolation between two sounds.

can perform spectral interpolation between two sounds by mixing according to

$$Y(f) = [e_1 E_1(f) + e_2 E_2(f)] \cdot [c_1 H_1(f) + c_2 H_2(f)]. \tag{8.40}$$

The excitation grains and the spectral envelopes are added. This transformed representation is then used for the resynthesis operation. We introduce cross-terms by this method, which musically means that the excitation source of one sound also influences the spectral envelope of the second and conversely. For regular mixing of two sounds the result would be $k_1 E_1(f) \cdot H_1(f) + k_2 E_2(f) \cdot H_2(f)$. The M-file 8.16 performs time-varying spectral interpolation between two sounds. We go from a first sound to another one by independently mixing the sources and resonances of these two sounds.

**M-file 8.16** (UX_spectral_interp.m)

```
% UX_spectral_interp.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a spectral interpolation with cepstrum
%
% k: spectral mix, calculated at every step in this example, as
%    starts with gain=0 for sound 1 and gain=1 for sound 2
%    finishes with gain=1 for sound 1 and gain=0 for sound 2
%    so we move from sound 1 to sound 2
clear;

%----- user data -----
[DAFx_in1,SR] = wavread('claire_oubli_voix.WAV');  % sound 1
DAFx_in2 = wavread('claire_oubli_flute.WAV'); % sound 2
n1       = 512;            % analysis hop size
n2       = n1;             % synthesis hop size
s_win    = 2048;           % window length
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;             % synthesis window
cut      = 50              % cut-off quefrency

%----- initializations -----
L        = min(length(DAFx_in1), length(DAFx_in2));
DAFx_in1 = [zeros(s_win, 1); DAFx_in1; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in1)); % 0-pad + norm
DAFx_in2 = [zeros(s_win, 1); DAFx_in2; ...
   zeros(s_win-mod(L,n1),1)] / max(abs(DAFx_in2)); % 0-pad + norm
DAFx_out = zeros(length(DAFx_in1),1);

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = L - s_win;
while pin<pend
   %---- k factor (spectral mix) wich varies between 0 and 1
   k      = pin / pend;
   kp     = 1 - k;
   %---- extracting input grains
   grain1 = DAFx_in1(pin+1:pin+s_win) .* w1;
   grain2 = DAFx_in2(pin+1:pin+s_win) .* w1;
%=========================================
   %---- computing spectral shape of sound 1
   f1     = fft(fftshift(grain1));
```

```
   flog  = log(0.00001+abs(f1));
   cep   = fft(flog);
   cep_coupe   = [cep(1)/2; cep(2:cut); zeros(s_win-cut,1)];
   flog_coupe1 = 2*real(ifft(cep_coupe));
   spec1 = exp(flog_coupe1);
   %---- computing spectral shape of sound 2
   f2    = fft(fftshift(grain2));
   flog  = log(0.00001+abs(f2));
   cep   = fft(flog);
   cep_coupe   = [cep(1)/2; cep(2:cut); zeros(s_win-cut,1)];
   flog_coupe2 = 2*real(ifft(cep_coupe));
   spec2 = exp(flog_coupe2);
   %----- interpolating the spectral shapes in dBs
   spec  = exp(kp*flog_coupe1+k*flog_coupe2);
   %----- computing the output spectrum and grain
   ft    = (kp*f1./spec1+k*f2./spec2).*spec;
   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
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

%----- listening and saving the output -----
% DAFx_in = DAFx_in1(s_win+1:s_win+L);
DAFx_out = DAFx_out(s_win+1:s_win+L) / max(abs(DAFx_out));
soundsc(DAFx_out, SR);
wavwrite(DAFx_out, SR, 'spec_interp.wav');
```

Spectral interpolation is the first step towards morphing, a term borrowed from the visual domain but with a much more ambiguous meaning in the audio domain. Time synchronization between the two sounds has to be taken into account. The matter of pitch interpolation should be different from the mixing of excitation signals as it is presented here. Morphing usually relies on high-level attributes and spectral interpolation and follows more complicated schemes, such as shown in Figure 8.22. Advanced methods will be discussed in Chapter 10.

### 8.3.4 Pitch shifting with formant preservation

In Chapter 6, we saw some pitch-shifting algorithms which transpose the entire spectrum, and consequently the spectral envelope. This typically alters the voice giving a "Donald Duck" or "barrel" feeling. For pitch shifting a sound without changing its articulation, i.e., its formant structure, one has to keep the spectral envelope of the original sound.

#### Inverse formant move plus pitch shifting

A possible way to remove the artifacts is to perform a formant move in the inverse direction of the pitch shifting. This process can be inserted into a FFT/IFFT-based pitch-shifting algorithm before the reconstruction, as shown in Figure 8.23. For this purpose we have to calculate a correction function for this formant move.

The following algorithm (see M-file 8.17) is based on the pitch-shifting algorithm described in Chapter 7 (see Section 7.4.4). The only modification is a formant move calculation before the

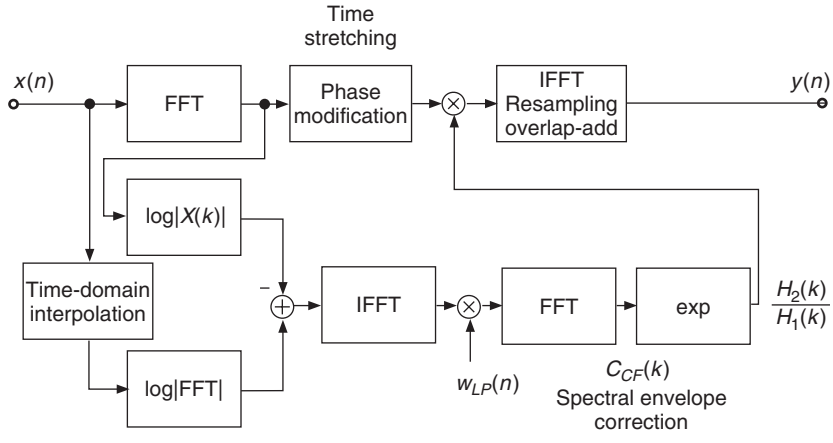


**Figure 8.23** Pitch shifting with formant preservation: the pitch shifting is performed in the frequency domain.

reconstruction of every individual grain, which will be overlapped and added. For the formant move calculation, a crude interpolation of the analysis grain is performed, in order to recover two spectral envelopes: the one of the original grain and the one of its pitch-transposed version. From these two spectral envelopes the correction factor is computed (see previous section) and applied to the magnitude spectrum of the input signal before the reconstruction of the output grain (see Figure 8.23).

**M-file 8.17** (UX_pitch_pv_move.m)

```
% UX_pitch_pv_move.m   [DAFXbook, 2nd ed., chapter 7]
% ==== This function performs a ptch-shifting that preserves
%      the spectral enveloppe
clear;

%----- user data -----
[DAFx_in, SR] = wavread('la.wav'); % sound file
n1     = 512;  % analysis hop size
               % try n1=400 (pitch down) or 150 (pitch up)
n2     = 256;  % synthesis hop size
               % keep it a divisor of s_win (256 is pretty good)
s_win  = 2048; % window length
order  = 50;   % cut quefrency
coef   = 0.99;   % sound output normalizing ratio

%----- initializations -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;   % synthesis window
tscal    = n2/n1;  % time-scaling ratio
s_win2   = 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)); % 0-pad + norm
%-- for phase unwrapping
omega    = 2*pi*n1*[0:s_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);
DAFx_out = zeros(lx+length(DAFx_in),1);
x        = 1 + (0:lx-1)'*s_win/lx;
ix       = floor(x);
ix1      = ix + 1;
dx       = x - ix;
dx1      = 1 - dx;
warp     = n1/n2   % warpinf coefficient, = 1/tscal
t        = 1 + floor((0:s_win-1)*warp);
lmax     = max(s_win,t(s_win))

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin    = 0;
pout   = 0;
pend   = L - lmax;

while pin<pend
   grain   = DAFx_in(pin+1:pin+s_win).* w1;
%=========================================
   f       = fft(fftshift(grain));
   r       = abs(f);
   phi     = angle(f);
   %---- unwrapping the phase ----
   delta_phi = omega + princarg(phi-phi0-omega);
   phi0    = phi;
   psi     = princarg(psi+delta_phi*tscal);
   %---- moving formant ----
   grain1  = DAFx_in(pin+t) .* w1;
   f1      = fft(grain1)/s_win2;
   flog    = log(0.00001+abs(f1))-log(0.00001+abs(f));
   cep     = ifft(flog);
   cep_cut = [cep(1)/2; cep(2:order); zeros(s_win-order,1)];
   corr    = exp(2*real(fft(cep_cut))); % correction enveloppe
   %---- spec env modif.: computing output FT and grain ----
   ft      = (r.* corr.* exp(i*psi));
   grain   = fftshift(real(ifft(ft))).*w2;
   %---- pitch-shifting: interpolating output 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
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

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

An illustrative example of pitch shifting with formant preservation is shown in Figure 8.24. The spectral envelope is preserved and the pitch is increased by a factor of two.

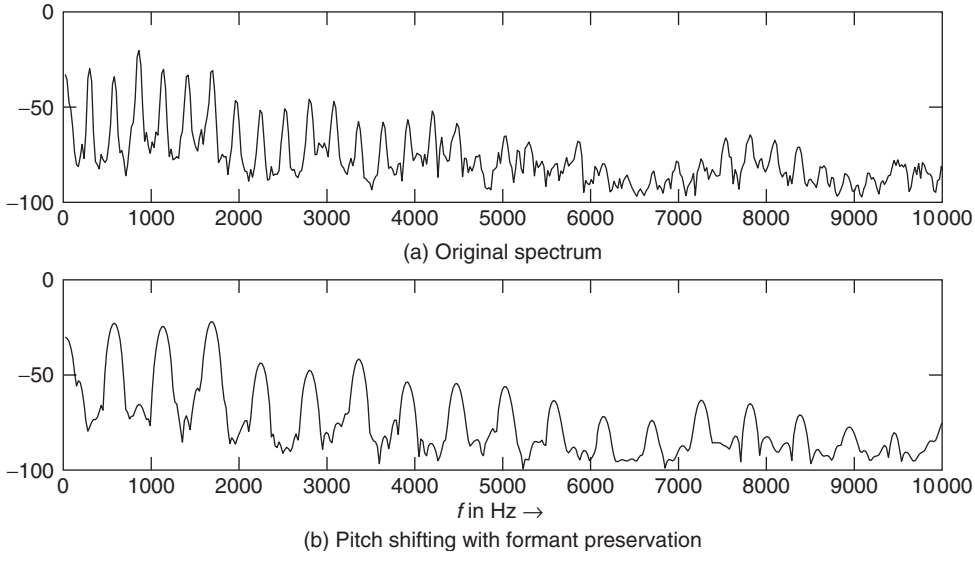


**Figure 8.24** Example of pitch shifting with formant preservation.

### Resampling plus formant move

It is also possible to combine an interpolation scheme with a formant move inside an analysis-synthesis loop. The block diagram in Figure 8.25 demonstrates this approach. The input segments are interpolated from length $N_1$ to length $N_2$. This interpolation or resampling also changes the time duration and thus performs pitch shifting in the time domain. The resampled segment is then applied to an FFT/IFFT-based analysis/synthesis system, where the correction function for the formant move is computed by the cepstrum method. This correction function is based on the input spectrum and the spectrum of the interpolated signal and is computed with the help of the

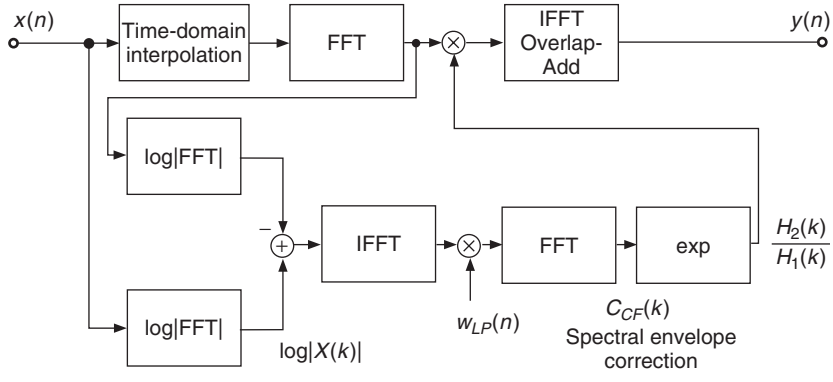


**Figure 8.25** Pitch shifting with formant preservation: the pitch shifting is performed in the time domain.

cepstrum technique. Then the correction function is applied to the interpolated input spectrum by the fast convolution technique.

The following M-file 8.18 performs interpolation of successive grains with a ratio given by the two numbers $R_a =$ `n1` and $R_s =$ `n2` and performs a formant move to recover the original spectral envelope.

**M-file 8.18** (`UX_interp_move.m`)

```
% UX_interp_move.m   [DAFXbook, 2nd ed., chapter 8]
% ==== This function performs a ptch-shifting that preserves
%      the spectral enveloppe
clear;

%----- user data -----
[DAFx_in, SR] = wavread('la.wav'); % sound file
n1     = 400;  % analysis hop size
               % try n1=400 (pitch down) or 150 (pitch up)
n2     = 256;  % synthesis hop size
               % keep it a divisor of s_win (256 is pretty good)
s_win  = 2048; % window length
order  = 50;   % cut quefrency
coef   = 0.99; % sound output normalizing ratio

%----- initializations -----
w1       = hanning(s_win, 'periodic'); % analysis window
w2       = w1;      % synthesis window
tscal    = n2/n1    % time-scaling ratio
hs_win   = s_win/2; % half window size
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 + norm
%-- for linear interpolation of a grain of length s_win
lx       = floor(s_win*n1/n2);
DAFx_out = zeros(ceil(tscal*length(DAFx_in)),1);
x        = 1 + (0:s_win-1)'*lx/s_win;
ix       = floor(x);
ix1      = ix + 1;
dx       = x - ix;
dx1      = 1 - dx;
warp     = n1/n2   % warpinf coefficient, = 1/tscal
lmax     = max(s_win,lx)

tic
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
pin  = 0;
pout = 0;
pend = L - lmax;

while pin<pend
%=========================================
   %----- FT of interpolated grain
   grain1  = (DAFx_in(pin+ix).*dx1 + DAFx_in(pin+ix1).*dx).* w1;
   f1      = fft(grain1)/hs_win;
   %----- FT of reference grain, for formant matching
   grain2  = DAFx_in(pin+1:pin+s_win).* w1;
   f2      = fft(grain2)/hs_win;
```

```
    %----- correction factor for spectral enveloppe
    flog    = log(0.00001+abs(f2))-log(0.00001+abs(f1));
    cep     = ifft(flog);
    cep_cut = [cep(1)/2; cep(2:order); zeros(s_win-order,1)];
    corr    = exp(2*real(fft(cep_cut)));
    %----- so now make the formant move
    grain   = fftshift(real(ifft(f1.*corr))).*w2;
    % plot(grain);drawnow;
%=========================================
    DAFx_out(pout+1:pout+s_win) = DAFx_out(pout+1:pout+s_win) + grain;
    pin     = pin + n1;
    pout    = pout + n2;
end
%UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
toc

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

**Resampling of the excitation signal**

Instead of moving the formants, an alternative technique is to calculate an excitation signal by removing the spectral envelope, to process the excitation signal by a pitch shifting algorithm and to filter the pitch shifted excitation signal with the original spectral envelope. LPC algorithms can be used for this approach. Figure 8.26 shows a block diagram of pitch shifting with formant preservation using the LPC method. First, a predictor is computed and the predictor is used for the inverse filtering of the input signal, which yields the excitation signal. Then the excitation signal is applied to a pitch shifting algorithm and the output signal is filtered with the synthesis filter $H_1(z)$. The processing steps can be performed completely in the time domain. The pitch shifting is achieved by first time stretching and subsequent resampling.

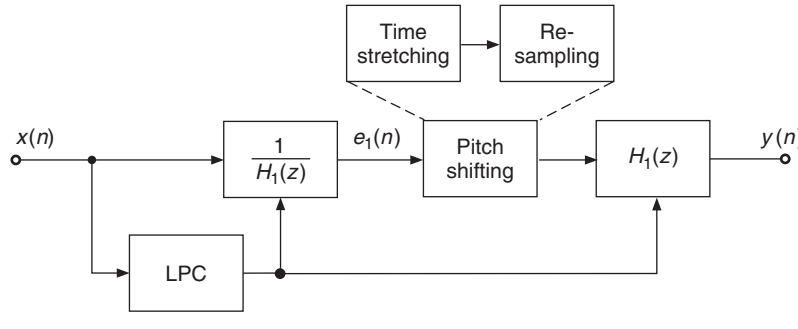


**Figure 8.26** Pitch shifting with formant preservation with the LPC method.

## 8.4 Conclusion

The central topic of this chapter is the division of the audio signal into its source signal and a time-varying filter derived from the spectral envelope of the signal. These two features are individually processed before synthesis of an output signal. The source-filter model of an audio signal, originally

a basic technique for speech processing, allows the implementation of several digital audio effects based on these two global features of an audio signal and opens up new vistas for experimentation and further research. These global features can either be extracted by time-frequency techniques (FFT/IFFT) and the cepstrum method or time-domain techniques based on linear prediction (LPC). Both techniques deliver a source-filter model of the audio input signal. Beyond it, they allow the extraction of further global features such as pitch or fundamental frequency, and this will be further described in Chapter 9. A further alternative to the source-filter processing presented in this chapter, is the separation of the audio signal into individual components such as sinusoids and noise, which is discussed in Chapter 10.

# References

[CLM95] O. Cappé, J. Laroche and E. Moulines. Regularized estimation of cepstrum envelope from discrete frequency points. In *IEEE ASSP Workshop on App. Signal Proces. Audio Acoust*., pp. 213–216, 1995.

[GR90] T. Galas and X. Rodet. An improved cepstral method for deconvolution of source-filter systems with discrete spectra: Application to musical sounds. In *Proc. Int. Comp. Music Conf. (ICMC'90), Glasgow*, pp. 82–8, 1990.

[HKS+00] A. Härma, M. Karjalainen, L. Savioja, V. Välimäki, U. K. Laine and J. Huopaniemi. Frequency-warped signal processing for audio applications. *J. Audio Eng. Soc*., 48(11): 1011–1031, 2000.

[KAZ00] F. Keiler, D. Arfib and U. Zölzer. Efficient linear prediction for digital audio effects. In *Proc. DAFX-00 Conf. Digital Audio Effects*, pp. 19–24, 2000.

[Lar95] J. Laroche. *Traitement des Signaux Audio-Fréquences*. Département TSI, Sup'Télécom Paris, 1995.

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

[LS81] P. Lansky and K. Steiglitz. Synthesis of timbral families by warped linear prediction. *Comp. Music J*., 5(3): 45–47, 1981.

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

[Mak77] J. Makhoul. Stable and efficient lattice methods for linear prediction. *IEEE Trans. Acoust. Speech Signal Proc.* ASSP-25(5): 423–428, 1977.

[MG76] J. D. Markel and A. H. Gray. *Linear Prediction of Speech*. Springer-Verlag, 1976.

[Moo79] J. A. Moorer. The use of linear prediction of speech in computer music applications. *J. Audio Eng. Soc*., 27(3): 134–140, 1979.

[Nol64] A. M. Noll. Short-time spectrum and "cepstrum" techniques for vocal-pitch detection. *J. Acoust. Soc. Am*., 36(2): 296–302, 1964.

[Orf90] S. J. Orfanidis. *Optimum Signal Processing, An Introduction*, 2nd edition. McGraw-Hill, 1990.

[OS75] A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing*. Prentice-Hall, 1975.

[O'S00] D. O'Shaugnessy. *Speech Communication*, 2nd edition. Addison-Wesley, 2000.

[PM96] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing*. Prentice-Hall, 1996.

[Sch99] M. R. Schroeder. *Computer Speech*. Springer-Verlag, 1999.

[Str80] H. W. Strube. Linear prediction on a warped frequency scale. *J. Acoust. Soc. Am*., 68(4): 1071–1076, 1980.