9. Analysis and resynthesis procedures

(This section last updated June 2002)

This section provides tutorial information on three *analysis/resynthesis* techniques available on the ECMC SGI and Linux systems:

• **phase vocoder** programs analyze the changing signal energy within hundreds or thousands of very narrow frequency bands; during resynthesis, the signal is reconstructed, but modifications in pitch, duration or other parameters can be introduced

• **sms** ("**S***pectral* **M***odeling* **S***ynthesis*") is a package of programs originally developed at CCRMA (Stanford University) in which sounds are divided into a periodic (pitched) component, analyzed in a manner similar to phase vocoder procedures, and a "stochastic" (aperiodic) component, analyzed as filtered white noise. *sms* resynthesis techniques provide a broad range of sound modification possibilities, including timbral morphing, but work much better with certain types of sounds than with others.

• **linear predictor coding** (*lpc*) analysis and resynthesis programs do not attempt to capture *every* frequency component within a source sound,but rather to capture time-varying spectral *formants* (resonances, or emphasized narrow frequency bands). Spectral formant analyses created by *lpc* software can be used with Eastman Csound Library instrument algorithm *resyn* to produce varied resynthesis of a sound, or with algorithm *xsyn* to create a hybrid, cross-synthesized "child" sound that has some characteristics of its two "parent" sounds; *lpc* resynthesis and cross-synthesis also can be performed with the *mixviews* application.

All *analysis/resynthesis* procedures (there are others in addition to the three surveyed here) involve two steps:

(1) time varying spectral (timbral) and amplitude analysis, and sometimes pitch analysis as well, of a source sound;

(2) resynthesis operations, in which this analysis data is used to create a new soundfile, generally with modifications such as pitch shifting, timbral modifications, time warping (expansion or contraction of the original duration) or formant shifting (changing the apparent size or physical characteristics of the vibrating object)

In most cases:

• the source soundfile to be analyzed must be monophonic

Exceptions: The *mixviews* application can perform both *phase vocoder* and (often less successfully) *lpc* analysis and resynthesis on *stereo* as well as mono soundfiles. The *PVC* programs can be used with input soundfiles with any number of input channels.

• the analysis and resynthesis operations are performed in succession, rather than simultaneously.

First one performs spectral analysis of a source soundfile. Most often, the resulting analysis data is written into a file, and the data within this analysis file is then used to perform resynthesis. Once one has obtained a usable analysis file, this file can be used any number of times to perform various types of resynthesis.

Exceptions: The *Ceres* phase vocoder application stores the analysis data in RAM for immediate resynthesis, and provides no means to save this analysis data permanently to a file. In most of the *PVC* programs, analysis and resynthesis are performed in a single, continuous operation, and the analysis is not saved. However, the *PVC* program *pvanalysis* does write analyses to disk files, and these analysis files can then be used with a few of the resynthesis programs, such as *twarp*.

Because these analysis files are very large — sometimes larger than the source soundfiles themselves — we store them on the *snd* disks rather than on the smaller system disks of *arcana* and *syrinx*. When you launch any of the analysis or resynthesis programs discussed here, the program automatically moves into current working home soundfile directory (*$SFDIR*), and will write all new files you create to your *$SFDIR* unless you specify an alternative subdirectory or path. In naming your analysis files, be sure that the name you provide will indicate to you in the future that the file is a phase vocoder, *sms* or *lpc* analysis file (which is not playable), rather than a soundfile.

In addition, the *sflib* directories include an **anal** ("analysis file") subdirectory, which in turn includes the following subdirectories:

*/sflib/anal/***pv** : includes phase vocoder analysis files available to all users

We have placed only a few files here, because phase vocoder analysis files tend to be very large, and they can be easily recreated.

*/sflib/anal/***sms** : includes *sms* analysis files available to all users

*/sflib/anal/***lpc** : includes linear prediction analysis files available to all users

All three of the techniques surveyed here share certain common capabilities. For example, *analysis/resynthesis* procedures generally provide independent control of pitch and duration, so that during resynthesis one can change the pitch of a sound without altering its duration, or vice versa. Note that this generally is not possible with most hardware or software resampling techniques, such as those provided by ECMC Csound Library algorithms *samp* and *tsamp*. Some sequencer, signal processing and plug-in programs, such as *Logic Audio, Peak, Hyperprism* and *ProTools*, provide time expansion and contraction by a different technique — by duplicating groups of samples to extend the duration, or eliminating groups of samples to reduce the duration. With some periodic sounds this works fairly well, but artifacts and glitches often result, especially with more complex sounds.

Phase vocoder, sms and *lpc* procedures each have unique capabilities — not available, or not obtained as easily or with as much control, by means of the other two techniques. And each has limitations and pitfalls as well. Here are a few general guidelines:

Most **phase vocoder** procedures are comparatively easy to use, presenting the user with relatively few decisions, but offer a comparatively narrow range of resynthesis modification possibilities, often limited to time warping and/or pitch shifting. However, extensions to these basic procedures, available within the *PVC* programs, provide many additional possibilities for sound modification.

Advantages: ease of use; often works fairly well, and sometimes very well, on a wide range of harmonic (pitched) and inharmonic sound sources; often the best or easiest method to use with low pitched or very high pitched sounds, with acoustically complex percussive sounds, with string tones, and with most types of speech, especially if your resynthesis goals are fairly straightforward or limited.

Limitations and disadvantages: time stretching often introduces artifacts, notably a metallic or artificially "reverberant" or smearing quality, although there are ways to minimize such distortion; pitch shifting also shifts the formants (resonances) of a sound, often resulting in timbral changes or an artificial timbral quality; analyses often need to be custom tailored to anticipated types of resynthesis modifications (an analysis that works well for one type of resynthesis modification may work poorly for other types of modifications)

sms provides some unique possibilities for timbral, pitch and durational modification during resynthesis, but is more complicated to use, and often works much better with comparatively simple harmonic (pitched) sounds than with inharmonic or timbrally complex source material or with speech.

Advantages: considerable flexibility in resynthesis operations; *sms* techniques are particularly well suited for "morphing" between two timbres, and for "detuning" a timbre by changing the frequency ratios between its partials; time warping usually does not introduce temporal smearing

Limitations and disadvantages: does not provide good resynthesis for many percussive or inharmonic timbres, or for most low pitched or very high pitched harmonic timbres, which may sound anemic or "synthetic" after resynthesis; certain types of quasi-harmonic timbres with strong noise components, including low pitched string tones and piano tones, can be difficult or impossible to capture adequately; complicated to use, sometimes requiring several attempts before a good analysis is obtained.

lpc resynthesis works well with certain types of both pitched and inharmonic timbres (e.g. woodwinds, crotales, maracas and, sometimes, vocal tones and speech), but often less well, or poorly, with other, timbrally complex types of timbres, such as piano and string tones. Cross-synthesis is often easier and more successful than timbral resynthesis.

Advantages: lpc is often the best method for the creation of hybrid, cross-synthesized timbres (timbres in between those of two acoustic sources); unlike most *phase vocoder* and *sms* techniques, pitch shifting will not result in formant shifting (unless you want it to); provides a wide range of

modification possibilities.

Limitations and disadvantages: *lpc* techniques are often the most complicated of the three surveyed here, sometimes requiring several retries before the desired result is achieved; works much better with some types of timbres than with others, and sometimes produces a buzzy quality or artifacts; resynthesis does not capture *exact* frequency ratios within a sound, but rather a simplified, overly harmonic approximation of these ratios; silences within a sound source may result in amplitude pops, chirps and hiccups.

In addition to the information provided here, I suggest that you consult the discussions of phase vocoder, *sms* and linear prediction procedures, as well as alternative analysis/resynthesis procedures, within Curtis Road's *Computer Music Tutorial.*

9.1. Phase Vocoder procedures

On the ECMC SGI and Linux systems several phase vocoder programs are available for use. Some of these programs provide a limited range of resynthesis modification possibilities, and are comparatively simple to use, while others provide more extensive capabilities, generally at some sacrifice in ease of use. Some employ a graphical interface, others a Unix command-line syntax. The four programs or packages listed below currently are the best documented and most reliable of our phase vocoder applications. Users are encouraged to try out two or more of these alternatives to determine which best suits their working and musical preferences:

(1) **Ceres** : recommended primarily as an introductory application, or for applying particular modifications to isolated sounds, rather than for systematic use; provides an excellent display of the analysis data (the best graphical display of any program surveyed here), which can be useful in understanding why timbres sound as they do; provides only a single analysis option; analyses and analysis settings cannot be saved, but the program does supply some useful additions to basic resynthesis options; recommended as especially for introductory work with phase vocoding

(2) **PVC** : a very comprehensive and powerful collection of programs that provide many analysis and resynthesis options. As a result of these many analysis and sound modification options, the learning curve can be rather steep. The *PVC* programs are run from a shell window by editing template script files. Highly recommended for those who want to explore the full capabilities of phase vocoding techniques.

(3) The *Csound* utility **pvanal**, used in conjunction with *Csound* unit generator **pvoc** or with Eastman Csound Library instrument **phavoc** ; the analysis options of *pvanal* are limited, but this method provides considerable flexibility, and also is by far the best method to use if you want to construct complete melodies, chordal sonorities or complex textures, rather than isolated sounds, by means of phase vocoder reynthesis; beginning users can employ analyses created with *pvanal* in scores for Library instrument algorithm *phavoc*; advanced users familiar with Csound orchestra file design can create their *own* resynthesis algorithms, which might incorporate some recently developed opcodes that provide additional resynthesis modification possibilities.

(4) *Mammut*, an idiosyncratic application, quite different in approach from all of the others discussed here, that requires an experimental, heuristic, "let's get lucky" attitude, rather than a methodical or systematic approach to sound modification. The resynthesis results are often surprising or unpredictable — you may generate a lot of garbage soundfiles, then suddenly produce a real gem without knowing exactly why. Recommended for those who like to tinker and experiment.

(5) **mixviews** is a powerful application that can be used to perform many types of signal processing operations (phase vocoding is just one of its several modules) within a somewhat non-standard GUI interface; initially this application may seem rather complicated to use; it provides a few unique phase vocoing capabilities, particularly for editing and modifying analysis data, and several of *mixviews*' phase vocoder options can be applied to *lpc* analysis and resynthesis operations as well. However, while the phase vocoder capabilities of *mixviews* are serviceable, I do not find these to be among the stronger features of the application. Compared with the four other applications above, the phase vocoding techniques provided by *mixviews* do not provide any particular special advantages. Thus, we will devote comparatively little time in class and in this *Users' Guide* to phase vocoding techniques with *mixviews*, and you likely will use this application more extensively in your work with LPC procedures.

Unfortunately, phase vocoder analysis files created with one application generally are not readable or usable by other applications. *Csound*, for example, can only read and process phase vocoder analysis files created with *pvanal*, and cannot open analysis files created with *mixviews* or *PVC*. Thus, if you use more than one of these phase vocoder applications, it is important to know which of these applications created each of your phase vocoder analysis files. One way to assure this is to use some consistent naming procedure when creating analysis files, such as using consistent file name extensions or prefixes.

Thus, if we have created two phase vocoder analysis files of the *sflib/string* soundfile *vln.b3*, one with the *PVC* program *pvanalysis* and one with *pvanal* for use with *Csound*, we might call these files

vln.b3.pvc or perhaps *pvcanal.vln.b3* (PVC)

vln.b3.pvanal or vanal.f2vln.b3 (pvanal/Csound)

Alternatively, we might prefer to include a file name extension such as *.pv* when naming all of these files, but to place them in separate subdirectories, one of which contains all of our *PVC* analysis files, another of which includes all analysis files created with the *Csound* utility *pvanal*.

The discussion below focuses primarily — but not exclusively — on the four applications listed above. Phase vocoder operations are the most widely used type of analysis/resynthesis procedures, on *Windows* and Macintosh as well as Linux/Unix systems. For this reason, the discussion that follows includes some generalized information applicable to almost all phase vocoder operations.

Analysis parameters

Various phase vocoder analysis applications differ in the number of required and optional arguments that can or must be provided by the user, and sometimes also in the scale or range of usable values for certain parameters. *Ceres* and Csound's *pvanal* provide relatively few analysis argument options, employing default values that cannot be changed for most analysis parameters. Beginning users may welcome this "lack of complication." One problem with such programs, however, is that if the resulting analysis does not provide successful resynthesis, or works with certain types of resynthesis modifications, but not with others, there is relatively little that the user can do to improve the analysis. Other programs, and in particular the *PVC* programs, initially may confront us with more parameter decisions than we would like; however, these "hooks" can provide a means to improve an unsatisfactory analysis, or to tailor an analysis to particular resynthesis goals.

A majority of phase vocoder analysis programs include at least two analysis parameters:

(1) *Frequency (spectral) resolution :* **Number of FFT filters**:
This value, often abbreviated *N* or *FFT*, determines the number of *Fast Fourier Transform* bandpass filters to be used in the analysis. The argument must be an integer and a power-of-two. Most often, the default value is 1024. A higher value, such as 2048 or even 4096, sometimes is necessary or desirable for adequate frequency resolution of complex timbres. Occasionally, 512 filters works better. The larger the number of filters, the slower the computation time, and the larger the resulting file.

The center frequencies of the filters are even spaced between 0 herz and the Nyquist frequency (1/2 of the sampling rate). The frequency resolution of the filters is

nyquist freq. / (FFT / 2)

[the *nyquist frequency* divided by the *FFT* value divided by 2]

Thus, with a 44100 herz soundfile and an *FFT* size of 1024, the frequency resolution is

22050 / (1024/2) = 22050/512 = 43.066 herz

In other words, the filters are centered about 43 herz apart, at 0 hz., 43 hz., 86 hz., and so on up to 22 kHz. If the source soundfile had a 22k sampling rate, the filters would instead be centered 21.533 herz apart.

The goal in setting the *FFT* size is to set it high enough that no more than one frequency component within the source sound spectrum lies within any individual filter,[1] but also, for reasons discussed below, not to set this value any higher than necessary to achieve this end. Thus, if the source sound is a high pitched flute tone, an *FFT* size of 1024 will probably do just fine (the partials of the flute tone are fairly far apart). By contrast, if the sound you are analyzing is a cymbal, or a very low piano tone (both of which may contain partials that are closer in frequency than 43 herz), a higher *FFT* value of 2048 or even 4096 may

[1] If two frequencies lie within a single filter band, they will "fuse" into a single resynthesis frequency with strong amplitude beating, resulting in timbral distortion. Generally, this is undesirable and will sound bad, but in certain cases it can produce interesting modifications to a source sound.

yield better results. However, some analysis programs, including *pvanal*, do not allow more than 1024 filters, although this limitation soon may be changed.

(2) *Analysis period :* **Window (Frame) Size**:
This parameter, which also must be an integer and a power-of-two, determines the number of samples included in (and thus the duration of) each analysis frame (or "window"). Often, this argument has a default value equal to the *FFT* parameter (usually 1024), or else to twice the FFT value. If the sampling rate of the source soundfile is 44.1k and the frame size is set to 1024, each analysis frame will average the amplitudes and frequencies of all spectral components detected over a period of about 23 milliseconds (1024/44100 = .0232). If we increase the window size to 2048 samples, each analysis frame will encompass about 46 milliseconds of the source sound, which is adequate for many sustained sounds with amplitude, pitch and spectral envelopes that evolve rather slowly. For strongly pitched harmonic source sounds (especially low pitched sources), window sizes 2048, 4096, or else one power-of-two larger than the FFT size, sometimes yields better frequency resolution than 1024 — if the analysis program allows such values.

However, with *phase vocoder* algorithms there generally is a trade-off between frequency resolution and temporal resolution. High *FFT* values and (to a somewhat lesser degree) high and *Frame/Window* size values generally produce better frequency (and thus timbral) resolution, but poorer temporal resolution, which may cause smearing or unwanted "reverberation" or "flanging." The values you choose for *FFT* and *Window* size will be determined in part, therefore, by whether you intend to be transpose the pitch of the sound, or to alter its duration. If we wish to stretch or shrink the duration of a source sound, but not change its pitch, lower *FFT* and *W* values, such as 512, may produce better results. If we will be performing both pitch transposition and temporal modification during resynthesis, we may need to try out different *FFT* and frame size values before finding the best compromise.

(3) *Frame offset*:
Each analysis frame is static, averaging the spectral content and amplitude over a duration determined by the window size. If the analysis frames analyzed successive, non-overlapping segments of the source sound, resynthesis likely would result in discontinuities, glitches and loss in audio quality. To correct for this, phase vocoder (and, also, *sms* and *lpc*) analysis algorithms overlap the frames, so that each frame shares many samples both with the preceding frame and with the following frame. A *frame offset* (sometimes called *decimation*) value determines the number of samples between adjacent frames, and thus, in combination with the *frame size/window* value, the amount by which the frames overlap. Of the phase vocoder applications surveyed here, only *mixviews* requires a *frame offset* value.

Resynthesis parameters

Two resynthesis parameters are included in almost all phase vocoder resynthesis programs:

(1) Some method to specify changes in duration (time warping)
and
(2) Some method by which to specify pitch transpositions

Pitch transpositions often are specified by means of a *Frequency Multiplier* parameter, whose argument is a multiplier for all of the frequency components detected in the analysis. Values of 0 and 1 typically have no effect. A value of .5 will transpose all of the frequency components within the source sound down an octave without affecting the duration. A value of 2 will raise the pitch of the soundfile an octave, and a value of 1.5 will shift the pitch up a perfect fifth. (Consult the online ECMC helpfile *pitchratios* for other pitch interval ratios.) However, pitch shifts also will result in a corresponding shift in formants, which probably will not be noticeable in idiophonic sounds, but will be all too apparent (the so-called "munchkin" or "sick cow" effect) when sounds with strong formants (such as vocal and string tones) are transposed up or down by more than a minor third or so.

Time warping is specified in various ways in different program. The PVC *programs include a time expansion/contraction factor* parameter. By contrast, *mixviews* and *Csound* simply require the user to specify an output duration for the resynthesis soundfile. If the output duration is less than the duration of the analysis, time compression results. (Each analysis frame is used for a shorter resynthesis duration than in the original sound.) When the output duration exceeds that of the analysis, time expansion results. (The reading of the analysis frames is "slowed down.") Although this may seem simple, it can cause problems when significant compression or expansion (by a factor of 4 or greater) are performed, but the *Frame (window) size* and

Frame offset (decimation) analysis parameters were not optimized for this much compression or expansion.

In addition to time warping and pitch shifting, the two most basic and common types of sound modification procedures applied during resynthesis, some phase vocoder programs — notably those in the *PVC* package — provide many other types of resynthesis modification possibilities that are controlled by additional parameters.

Simple soundfile examples

In the *sflib/x* directories of our SGI and Linux systems you can find a soundfile called *shortspeech*, a fragment from the beginning of the beloved soundfile *voicetest*. A phase vocoder analysis of *shortspeech* was used to create eight illustrative resynthesized soundfiles, also located in the *sflib/x* directory, named *phasevoc.1* through *phasevoc.8*. These eight soundfiles originally were created with a (now obsolete) NeXT phase vocoder application *PVC.app*. However, since these are fairly rudimentary examples of typical phase vocoder sound modification procedures, these soundfile examples could be created easily enough with *Ceres, PVC mixviews* or *Csound* as well.

phasevoc1 : straight resynthesis (should sound identical to the original)

Pitch shifting only:

phasevoc2 : pitch is lowered a perfect fourth (5 semitones)
phasevoc3 : pitch is lowered an octave
phasevoc4 : pitch is raised a perfect fourth (5 semitones)
phasevoc5 : pitch is raised an octave

Time warping only :

phasevoc6 : time compression by a factor of 4 (the output duration is 1/4 of the input duration, anbd events happen four times as quickly; each 256 input analysis frames are used to create 64 output resynthesis frames)
phasevoc7 : time expansion by a factor of 8 (the output duration is 8 times the input duration; each 256 input analysis frames are used to create 2048 output resynthesis frames)

Time warping AND pitch shifting :

phasevoc8 : time expansion by a factor of 1.5 (64 analuysis frames are used to create 96 resynthesis frames) and the pitch is raised by one semitone

9.1.1. Using CERES

Ceres is a graphically-based phase vocoder analysis and resynthesis application originally written for SGI systems by Oyvind Hammer (who also wrote the SGI *mix* application) and subsequently ported to Linux by several programmers. Hardcopy documentation is available within the *SGI DOCs* and *Linux DOCs* binders in rooms 52 and 53, and you should consult this documentation before and while using the application. A simplified tutorial summary is presented here. Another usage summary is provided in Dave Phillips' *Linux Music & Sound* text.

Ceres creates an analysis in RAM for immediate resynthesis, and provides no means to save the analysis data to a disk file. In addition, this does limit the size of soundfiles that can be analyzed. However, on all of our SGI and Linux systems (and especially on Linux systems*madking* and *firebird*, which currently have 512 MB of RAM, this rarely is much of a limitation. The application provides only one user-settable analysis parameter: the number of *FFT* filters to be used in all analyses you create after opening the program. The default value is 1024.

To start *Ceres* with this default *FFT* value, simply type *ceres* in a shell window. Unless you append an ampersand to this command line, *Ceres* will tie up this shell window for as long as it runs.
To set the *FFT* value to some other power-of-two number, such as 2048, type

ceres 2048

If you wish to change this value, you must quit the program and reopen it with a new argument. The user cannot adjust the *Frame (window) size* any other analysis parameter values, all of which are fixed.
The program will open with a blank window and menus for *File, Transform, Export* (rarely used) and *settings* near the top, just under the titlebar.

Two resynthesis modes are available: a faster but lower quality mode (the default) and a slower but better quality mode. Almost always, you will want to change immediately to the higher quality mode:

• Under the *Settings* menu, click on *Resynthesis*. Then, in the new window that opens, check the *Additive Synthesis* box and click on OK

You also may wish to change the default gray display to something more colorful:

• Again under the *Settings* menu, select *Display*, then change the default *Grey* (sic) *colors* default either to *Hot* or *Cold* colors.

To select a soundfile for analysis, click on *File*, then on *Load & Analyze*. In the selection window that opens, select or type in a soundfile and then click on OK or tap a carriage return. Stereo input soundfiles can be used, but the resynthesis output will be mono.

The analysis may take some time. When it is completed, a time varying display of the frequencies within the sound will be displayed.

Generally, it is best to perform straight resynthesis (no modifications) first to test the quality of the analysis. To do this, select *Synth & Save* under the *File* menu. In the "dialog box" that opens, type in a name for the output (resynthesized) soundfile and click on OK or tap a carriage return. When resynthesis is completed, play this test soundfile. If it sounds good, you can proceed to create one or more additional resynthesis soundfiles, this time with modifications. If the result is not good, your only recourse (other than quitting the program and reopening it with another *FFT* value) is to reduce the number of frequency components used in resynthesis. To do this, click on the *Transform* menu, then on *Sieve*, then set the *Number of harmonics* value to some (power-of-two) number less than the *FFT* value, then click on OK , and then redo the resynthesis.

Modifications

To perform time warping during resynthesis, click on the *Settings* menu, then on *Resynthesis* and, in the box that opens, set a *Time stretch factor* (greater than 1. for time expansion, less than 1. to compress the duration).

All other available resynthesis modifications, which affect either pitch or timbre, are accessed under the *Transform* menu.

For *pitch transposition*, select *Pitch shift* and, in the box that opens, set a *Transposition factor* ("frequency multiplier") greater than or less than 1. (Consult the helpfile *pitchratios* in a shell window for equal tempered intervallic ratios.)

To introduce a continuous *glissando*, also include a value in the *Multiplication per second* box, again using the *pitchratios* file as a guide if necessary. If we place a value of 1.059 in this box, the pitch will ascend by one semitone (a frequency ratio of 1:1.059) during each second of the output soundfile.

Checking the *Control function* box and setting a *Static frequency* can produce interesting effects, in which all of the frequency components either diverge from, or converge on, this fixed frequency.

After setting the *Pitch shift* parameters, and any other *Transform* options, as you wish, and closing the boxes for these options, again select *Synth & Save* under the *File* menu to perform resynthesis.

Although the *File* menu also includes a *Play* option, the soundfiles available for playing will not include any new, resynthesized soundfiles you create with *Ceres*. You must play your output soundfiles from a shell window or some other application.

After experimenting with the time warping and pitch transposition options provided by *Ceres*, you also may wish to explore some of the timbral modification options available under the *Transform* menu, and discussed (briefly) within the hardcopy *Ceres* documentation:

Sieve (which, as noted earlier, reduces the number of FFT frequencies used in resynthesis) will simplify a timbral spectrum, especially when small values (less than 20) are specified. With a *sieve* value of 10, *ceres* will completely eliminate all but the ten strongest (highest amplitude) frequencies within the spectrum. An additional option enables us to change this value exponentially over time, so that the spectrum either becomes gradually less complex or more complex ("richer").

Spectrum shift adds a fixed positive or negative number to each frequency component. This *frequency shifting* "detunes" the timbre (altering the ratios between its frequencies, and usually making these ratios more complex). Note that the efffect is quite different from pitch transposition, which preserves the *ratios* between the frequency components of the spectrum.

Filter applies a *band reject* filter to the analysis, filtering out a contiguous portion of the original spectrum.

Average smooths out time varying changes in the frequency components, averaging the frequency of each component over the full duration of the output soundfile. In this manner, we could change normal speech inflections into a monotone.

To use *Move to pitch grid* , first click on the *Settings* menu and select *Grid scale*, and set this value to *Major, Minor, Pentatonic* or *Chromatic*. Then, in the *Move to pitch grid*, select a probability between 0 (no change in the original frequencies) and 1 (maximum change). If this parameter is set to *1*, all frequency components within the resynthesized soundfile will be shifted so that the resulting sound outlines a major or minor chord, or a pentatonic or chromatic scale.

☞ Important note: If you create a series of resynthesis soundfiles in succession, *Ceres* generally will retain the previous resynthesis values while adding changes you make. Thus, if you apply a pitch shift, and then, in your next resynthesis, edit the *Spectrum shift* parameter, the preceding *Pitch shift* value likely will be retained. This may or may not be what you intend. When in doubt, check through the various resynthesis parameter settings under the *Transform* menu, and also the current setting of the *Time stretch* factor, before launching a resynthesis.

9.1.2. Using the PVC programs at Eastman

PVC is a collection of excellent phase vocoder programs (currently the most extensive and powerful phase vocoder programs available on the ECMC systems) written by Paul Koonce. Documentation in *html* format is available online through a link in the *onlinedocs* page of the ECMC web site (read the ECMC version rather than Paul Koonce's original version). This document, also included in the hardcopy *SGI DOCS* and *LINUX DOCs* binders in the studios, has been edited by Allan Schindler to reflect ECMC usage, and should be your principal starting point and reference source for work with the *PVC* programs.

To simplify usage of many (but by no means all) of the *PVC* programs described in the html documentation, I have created local scripts, with the routine name followed by the extension *.tp*, ("template") based upon models provided by Koonce, that can be used to run these programs. To obtain a list of currently available ECMC templates for PVC programs, type

pvc.tp

To obtain a summary on how to use one of these template scripts, type the script name with no arguments. For example, typing

plainpv.tp

will display a summary of how to use the *plainpv.tp* template. To see a generic *plainpv* script without providing input and output soundfile arguments, type:

plainpv.tp -

The *PVC* programs most frequently used by ECMC users are *plainpv, pvanalysis* and *twarp*. However, you may discover that one of the other programs is ideally suited to a particular compositional application.

There are some internal differences in how the *PVC* programs work on our SGI and Linux systems. For example, on the SGI systems, the input soundfile must be in AIFF format, and output soundfiles also are ultimately written in AIFF format, but (for rather complicated reasons) internal processing is done in NeXT/Sun format. By contrast, on the ECMC Linux systems, input and output soundfiles can be either in AIFF or in WAVE format. Despite these under-the-hood differences, however, a script that you create to run a PVC program on either an ECMC Linux or SGI system will produce exactly the same result when run on any of our other systems.

In addition to these *.tp* script templates, I have created example script files for many (but, again, not for all) of the *PVC* programs. To obtain a listing of these example files, type

pvcex

To display one or more of these example PVC script files through the paging program "less," type:

pvcex filename(s)

To capture one or more of these files, type:

getpvcex filename(s) > filename

Hardcopy of all of these example files is available in the *ECMC PVC Examples* binder in the studios.

Soundfiles in the *sflib/x* directory exist for all of these examples except for a few that do not create soundfiles, but instead create an analysis file or some other type of file.

To learn how to use *plainpv*, the most basic program in the PVC package, or any other *PVC* program for which an ECMC *.tp* script template exists, I recommend the following steps:

(1) Read a portion of the online or hardcopy HTML documentation on *PVC* and find a program you want to try.
(2) Find out if an ECMC script exists for the program by typing

pvc.tp

If an ECMC script does exist:
(3) Find out whether one or more ECMC example files exist for the program by typing

pvcex

and note the names of example files for the program. In the case of *plainpv*, there are several examples, and we probably would begin with example *plainpv1*
(4) Look at one of the example files. To see example *plainpv1*, for example, type

pvcex plainpv1

or else consult the printout of this file in the *ECMC PVC Examples* binder. While studying this example, listen to the compiled soundfile in the *sflib/x* directory that was created by this example file:

psfl plainpv1

(5) Look at, and listen to, other examples created by the program, such as *plainpv2* and *plainpv3*
(6) When you are ready to use the program yourself, obtain a template file for the program. For a usage summary of how to use an ECMC *.tp* script, type the script name with no arguments:

plainpv.tp

Then type

plainpv.tp insound outsound

and, if everything looks okay,

!! > scriptfile

or else simply

plainpv.tp insound outsound > scriptfile

to create a script file with analysis and resynthesis parameters that you can edit and then use to run *plainpv*.
(7) Next open this script file with *vi* or some other text editor, changing some of the default parameter values within the top half of the file to meet your resynthesis goals. Do not change anything within the bottom half of the file (after the "*OFFICE USE ONLY*" line except at the very end of the script, where you can remove any temporary *gen* function files you have created. The pound sign # serves as the comment symbol for all *PVC* scripts, and all characters on a line that follow this symbol are ignored by the *PVC* program.
(8) When your script file is ready, run the program with the command

sh filename

(Note that *PVC* scripts must be run by a Bourne shell, with the *sh* command. The Bourne shell, the oldest type of Unix shell, differs in some ways from the *cshell* [*csh*] with which you probably are more familiar.)
(9) When the job is completed, play the resulting soundfile. If you aren't completely happy with the result, edit the script file again and run it again.

Phase vocoder jobs sometimes can take a long time to run. You can suspend any *PVC* job in process at any time by typing ^z (*control z*). However, on our SGI systems, the partially compiled output soundfile will be in NeXT, not AIFF format, and will have the temporary name *pvcout*. Type

p pvcout

to play these partially completed soundfile. Resume compilation by typing *%* or *fg*. To kill the job, type *^c* after resumption. On the Linux systems, you can suspend compilation to play a partially compiled soundfile in the same manner; however, the partially compiled soundfile will have the name you have provided (rather than *pvcout*), and will be in WAVE format.

For advanced and adventurous ECMC users: If you want to try to use a program, such as *ringfilter*, for which I have not created an ECMC *.tp* script, you can find Paul Koonce's *S.program_name* scripts in the directory

/usr/local/soundapps/PVC/SCRIPTS.

However, these scripts will not work out-of-the-box. For one thing, they require NeXT format input soundfiles, and for another, they will not place you in your soundfile directory.

Many of the parameter values in the *PVC* programs can be controlled by means of time varying function tables created with *Cmusic gen routines* that are bundled with the *PVC* distribution. These *Cmusic gen routines* are similar in many respects to *Csound gen routines*, but there are some important differences as well, as discussed and illustrated at length within the ECMC version of the HTML *PVC* documentation. Additional information on these *gen routines*, excerpted from F. Richard Moore's text *Elements of Computer Music*, first edition, is included as an appendix within the hardcopy *ECMC PVC Examples* binder.

9.1.3. Performing phase vocoder analysis and resynthesis with CSOUND

The *Csound* distribution includes a standalone phase vocoder analysis program called *pvanal*, and a unit generator called *pvoc* that can be included within an orchestra file instrument to read in analysis files and perform resynthesis. At Eastman, the steps involved in this process include:

(1) Use *pvanal* to create an analysis file.

(2) Use the local utility *pvlink* to create a soft *link* file, in your current working Unix directory, that "points to" this analysis file. See the online or hardcopy *man* page for *pvlink* for details on using this simple script.

(3) Use either an orchestra file that includes Eastman Csound Library algorithm *phavoc*, or else an instrument algorithm of your own design, to create a resynthesized soundfile. (Your own "instrument" might be based upon the generic *phavoc* algorithm, but include modifications or extensions.)

A *man* page for *pvanal* is available online, and in hardcopy both within the *Csound* manual itself and in the *SGI DOCs* and *LINUX DOCs* binders. For a usage summary of the program, type the command name with no arguments. The summary will look like this:

```
Usage: pvanal [-n<frmsiz>] [-w<windfact> | -h<hopsize>] [-g | -G<latch>]
  [-v | -V txtfile] inputSoundfile outputFFTfile
```

This is repulsive, but there is good news: The program is fairly robust. In most cases the flag argument default values (some of which depend upon the sampling rate) work adequately, and the simple command

*pvanal Input***Sound***file Output***Analysis***file*

often is sufficient to produce a usable analysis. If not, you can try tinkering with the *-n* argument and, if still unsatisfied, with either the *-w* or the *-h* argument.

- As discussed earlier, the *-n* ("frame size") argument determines the number of samples analyzed within each analysis frame. With 44.1k soundfiles, an argument of 1024 (the maximum allowed, and also the default) generally works well, although there are occasions when some of us wish we could raise this value.
- The *-h* ("hopsize") argument corresponds to the *Frame offset* parameter discussed earlier.
- Alternatively, in place of a *-h* value, it is more common to specify a *-w* ("window overlap") factor, which has a default value of 4. This specifies that the *-h* ("hopsize," or "frame offset") value equals the *-n* "window size" divided by 4. If *-n* is set to 1024, the "hopsize" value would be 256. The *-w* argument should not exceed 8 (*frame size* divided by 8).

pvanal and the *phavoc* Library algorithm do not provide the abundant modification possibilities of such applications as *Ceres* or *mixviews*. With *pvanal*, one

- cannot adjust the number of *FFT* filters, an unfortunate limitation;
- one must analyze an entire input soundfile (it is not possible to analyze only a portion of the sound); and
- the resulting analysis data currently cannot be edited by simple means

Despite these limitations, if you want to create rhythms, melodies, complex textures or simply a lot of output notes from one or more phase vocoder analysis files, Csound's ability to resynthesize any number of simultaneous or successive output notes in a single pass, with user control over the relative amplitudes and

other phrasing characteristics of these notes, can be invaluable. The score file that produced *sflib/x* example soundfile *phavoc22*, for example, creates 22 output notes. One recoils at the tedium that would be involved in creating each of these resynthesis notes one at a time with one of our other phase vocoder programs, and then mixing them all together.

For usage information on the *phavoc* algorithm, along with a score template and example scores, consult the *Eastman Csound Library* binder. A score template is obtained in the usual fashion:

getsc phavoc > filename

Advanced users will find some recently developed extensions to Csound's basic phase vocoder resources. *pvadd* can be used to synthesize only a single frequency component, or groups of selected frequency components, from an analysis file, "filtering out" all other frequencies. Several *pvadd* opcodes could be employed within an instrument. *pvcross* and *pvinterp* provide some basic cross synthesis possibilities between two source sounds. *pvcross* maps applies the time varying amplitude values for each partial from one analysis file to the frequency values derived from a second analysis file. *pvinterp* can be used to "morph" between two sounds. *vpvoc* enables one to apply time varying alterations to the amplitudes of the frequency components of a sound.

9.1.4. Using MAMMUT

mammut is an offbeat, phase-vocoder based SGI application also written by Oyvind Hammer and subsequently ported to Linux by other programmers. *mammut* performs an FFT analysis of an input soundfile in a single window rather than in successive framesa, and then uses this "averaged" analysis data to perform various types of transformations on the sound. As a result of this unorthodox approach, the output resynthesis often will sound very different from the original sound, and generally will not follow the original spectral evolution. The same parameter values applied to two sound sources may produce startling different outputs.

Some ECMC users have found *mammut* to be a highly intriguing application that produces results that could not easily be obtained with any other DSP program. Other users find the application simply goofy. Procedures for running the application are provided in the ECMC *help* file *mammut*, and in the *SGI DOCs* and *LINUX DOCs* binders.

9.1.5. Using MIXVIEWS

mixviews is a comprehensive SGI and Linux application that allows us to perform various types of soundfile editing, analysis, simple mixing (such as cross fading between two soundfiles) and filtering operations. In addition, *mixviews* provides subroutines for performing phase vocoder analysis and resynthesis, and also *lpc* analysis and resynthesis in separate operations. Like the *dap* and *snd* applications, *mixviews* can be used instead of the bare-boned (but easy-to-use) SGI *soundeditor* application for editing soundfiles, and also to apply various DSP "effects" to these soundfiles (something that can *not* be done with *soundeditor*). Here at the ECMC, however, *mixviews* is used most frequently to create linear prediction analysis files for resynthesis with Csound.

Hardcopy documentation for *mixviews* is available in the *SGI DOCs* document in the studio. However, this documentation is more useful as a reference by users who already understand the basic operation of the program than as a tutorial introduction. No online HTML *Help* is available within the application.

Although powerful, the *mixviews* program can at times be awkward to use. Each major signal processing operation, such as performing a phase vocoder analysis, opens up a new window ("view") that displays the data that results from this operation. This data is stored in RAM rather than written to a file, until one explicitly performs a *Save* operation in this window. The GUI interface does not follow some standard X-windows conventions. For example, you cannot use the right mouse button to open a hidden menu and "pop" this window to the top of the stack. Rather, the window can only be selected by clicking on its title bar. With many windows ("views") open simultaneously, the title bar for the window you want to activate often will be hidden, requiring that you miniaturize other windows (by clicking in the boxed small dot near the right edge of the titlebar) or else move them to the corners of the monitor display, in order to access the desired window.

To open the *mixviews* application, type **mxv** (short for "**M***i***XV***iews*") in a shell window. This will open a blank *tmp* ("temporary") window with various menu choices under the titlebar, and place you in your home soundfile (*$SFDIR*) directory.

To open a soundfile (or some other type of file, such as a phase vocoder or *lpc* analysis file) for editing or use, click on the *File* menu and then select *Open*. In the *Open* selection box that appears, select or type in the name (and, if necessary, the directory) of the soundfile or analysis file to be opened and tap a carriage return, or else click on confirm. A window will open displaying an amplitude waveform (or some other type of data display) for the selected file. If you have opened a soundfile, and wish to play it, select *Play* under the *Sound* menu. Selecting only a portion of a soundfile for playing, analysis or editing cannot be done with the customary mouse dragging technique. Rather:

☞ To set a *beginning* "edit" point for a selection, click in the waveform display with the *left* mouse button;

☞ To select an *end* point for the section, click on the waveform display with the *middle* mouse button;

☞ To select the entire soundfile ("select all"), click anywhere within the waveform display with the *right* mouse button.

Soundfiles created (or else edited and then saved) with *mixviews* are saved in *AIFC* rather than *AIFF* format.

When you are done using *mixviews* you may have several large and miniaturized windows still open. Even if you select *Exit* or *Quit* from one of these windows, each open window must be closed, one by one, and for each window that contains unsaved data, you will be relentlessly interrogated as to whether or not you wish to save this data. If you are absolutely CERTAIN that you have saved everything that you want to keep, you can simplify this repetitive process by returning to the shell window from which you launched *mixviews* and typing a *control c*. This will "pull the plug" on *mixviews*, aborting the program and closing all of its open windows in a single step.

9.1.6. Using mixviews to perform phase vocoder analysis and resynthesis

[Most of you can skip this subsection on using *mixviews* to perform phase vocoder analysis and resynthesis.]

Creating a phase vocoder **analysis** *with mixviews*

To create a phase vocoder analysis of a soundfile:

(1) Make sure that a window for the source soundfile is open, and that you have selected the portion of this soundfile to be analyzed. This region must be highlighted.

(2) Under the *Analysis* menu, select *Phase vocoder analysis*.

In the window that opens, set the desired analysis values for the phase vocoder analysis.

• The *Frame size* (*Window size*) argument, discussed earlier, determines the number of samples to be analyzed per frame. A suggested starting point value is *1024*.

• The *Frame offset* and *Frame rate* arguments generally can be left at *0*. *mixviews* will then use default values for these arguments.

• The *FFT size* argument, as usual, determines the number of bandpass filters to be created. The default value of *256* generally is too low. A value of *1024* often works well.

Reminder: Increasing the *FFT* value to *2048* or higher yields better spectal resolution but poorer temporal resolution. Conversely, a value of 512 or 256 yields better temporal resolution, but poorer spectral resolution.

• If you will be performing time warping in your resynthesis, it usually is best to set a *Time scaling* value in this window, and use a corresponding time scaling value in resynthesis. For example, a *Time scaling* argument of *4* will optimize the analysis for time stretching a resynthesis by a factor of 4, so that the duration of a resynthesized soundfile will be four times as long as the duration of the sound that has been analyzed.

Note, however, that the resulting "custom" analysis file, optimized for a particular time expansion factor, probably will not yield good resynthesis results if used for time *compression*.

(3) After setting these parameters, click on *Confirm* or tap a carriage return. When the analysis is done, a new window will open displaying this analysis data. You may not find this display all that useful. Note that the analysis data is in RAM, and is not saved to a file until you perform a *Save* operation in this window.

At any time, now or in the future when you open this analysis file, you can obtain information on the analysis parameters by selecting *File info* under the *File* menu.

Advanced users should note that unlike many other phase vocoder programs, *mixviews* allows one to edit and alter the phase vocoder analysis data displayed in a window by means of various operations available under the *Edit, Modify* and *Pvoc* menus. These modification procedures sometimes can be useful in transforming sounds.

To perform phase vocoder **resynthesis** *with mixviews:*

(1) If a window for the phase vocoder analysis file to be used is not already open, open this analysis file, which will load the data into RAM. Then select how much of the analysis is to be used in resynthesis. To select (highlight) the entire file, click anywhere in the analysis display with the *right* mouse button; otherwise, use the left and middle mouse buttons to select beginning and end points within the analysis file.

(2) Within the analysis file window, click on *File*, then on *New Type* and then on *Sound File*.

(3) In the "dialog box" that opens, set the soundfile header values and a name for the resynthesis soundfile.

(4) A blank new *soundfile* window will open for the resynthesis soundfile data. In this window, click on *Sound*, then select *Synthesis*, then *Phase vocoder resynthesis*.

(5) Yet another "dialog box" will open, in which you must set the resynthesis parameters, including the desired output *duration*. This argument defaults to *one second*, and must almost always be changed.

- If you do *not* wish to perform time warping, your *Duration* argument should match the duration of the phase vocoder analysis file.
- If you *do* wish to perform time warping, set the *Duration* argument to the desired output duration.

To launch the resynthesis, tap a carriage return, or click on Confirm

(6) When the resynthesis has been completed an amplitude waveform display will appear and you can play the resynthesized sound. If you like it, you can perform a *Save* operation in the usual manner (select *Save* under the *File* menu).

9.2. Using SMS on the ECMC Linux systems

Spectral Modeling Synthesis is a powerful analysis/synthesis system created by Xavier Serra and other programmers that has been under development for the better part of a decade. Originally, *sms* was implemented on SGI systems, but most of the more recent development has been on Linux and *Windows* platforms. However, newer versions of these programs have tended to be buggy, as you will see. Currently, yet another iteration of SMS is in development — an open source, cross platform C++ library that should become available in 2002 or 2203, and some day may supercede the Linux version described in these pages.

In the ECMC studios we are running a five year old (but still fully functional and robust) package of SMS programs and related ECMC utilities on SGI system *arcana*. We also a running a recent version (2.6.3) of *sms*, installed in June of 2002, on our Linux systems. These SGI and Linux versions of SMS are completely incompatible; the programs and their parameters differ substantially, and analysis files created on one of these platforms cannot be used on the other.

Even though this can cause some initial confusion, we are maintaining the older SGI version of SMS on *arcana* for the benefit of old timers already familiar with it, and because it includes some significant resources that do not work correctly, or as well, in the newer Linux version. However, new users should learn the Linux version, and it is highly unlikely that you will want to take the (considerable) time necessary to master the SGI version as well. All of the documentation in this *Users' Guide* section, in the *LINUX DOCs* binder and in the online Linux *man* pages and Linux *ecmchelp* files refers to the Linux version of SMS. Legacy documentation on the SGI version is maintained on *arcana* and in the *SGI DOCs* binder. **Do all of your work with SMS on the Linux systems.** This includes viewing *man* pages, and viewing and listening to examples. The SMS documentation and example files on *arcana* refer to the SGI version of SMS, not to the Linux version.

Overview:

sms is based on a spectral model in which sounds are made up of two components: a *pitched* (also called "*deterministic*") component, and a *noise* (also called "*residual*" or "*stochastic*") component. When analyzing a soundfile, *sms* first performs Fourier transform procedures to identify the harmonic or inharmonic *partials* within the sound. Unlike phase vocoder programs, however, *sms* also attempts to "connects the dots," tracking each partial "trajectory" from frame to frame as a series of sinusoids that change in frequency and amplitude over time. The remaining portion of the sound, called the *residual*, which could not be modeled by the frame-to-frame evolution of these sinusoids, is then analyzed as filtered noise.

sms analysis/resynthesis is basically a two-step process:

1) Create an analysis of a soundfile that models the pitch, amplitude and spectral evolution of the sound over time. This analysis is written to a (large) file stored on the *snd* disk (along with your soundfiles), but it is not playable.

2) Use this analysis file to synthesize a new soundfile.

These two basic steps, however, can be broken down into a series of smaller tasks. SMS provides more than fifty parameters for tailoring an analysis to the particular pitched, spectral and amplitude evolution of the source sound, and even more options for modifying this source sound during resynthesis. Because so many variable options are available during both of these operations, these argument values are stored as parameters within ASCII input files during both the analysis and synthesis stages.

Thus, several files are created when we use *SMS*:
(1) an *ASCII analysis parameter* file, which sets all of the values to be used to create an analysis
(2) a binary data *analysis* file
(3) generally a simple *test soundfile* is then created in order to assess how successfully the analysis has captured (modeled) the key components of the source sound
(4) an *ASCII synthesis parameter* file, which sets all of the values to be used in re-synthesizing the sound with modifications
(5) the *synthesis soundfile*

The executable *sms* program has two modes that can be used to create both analysis files and synthesis soundfiles:

sms analysis analysis_parameter_file (creates an analysis file)
and
sms synthesis synthesis_parameter_file (creates a synthesis soundfile)

However, I recommend that you never run *sms* this way. The *sms* parser is fairly primitive, and the program often aborts when it encounters even trivial syntactical errors. Also, the *sms* binary has some bugs, and sometimes has trouble locating files. For these reasons (and, more mundanely, beause I have trouble spelling both "analysis" and "synthesis"), I have created five local scripts to simplify running *sms* on the ECMC Linux systems:

(1) *smsanaltp* : provides a template for creating an ASCII analysis parameter file
(2) *smsanal* : runs *sms* in analysis mode to create an analysis file
(3) *smstest* : used to test the quality of an analysis file
(4) *smsynthtp* : provides a template for creating an ASCII synthesis parameter file
(5) *smssynth* : runs *sms* in synthesis mode to create a soundfile

To save yourself a lot of aggravation and head scratching, you always should use *smsanal* and *smssynth*, rather than the *sms* command, to create analysis and resynthesis files. The entire procedure is summarized in the following pages. All of the programs associated with *sms* except a GUI called *smsrtsynth*, which you may or may not choose to use, are run from a shell window. Typing any of the commands above with no arguments will display a usage summary, and *man* pages are available for each of these five utilities.

9.2.1. SMS analysis

Preparing analysis parameter files: smsanaltp

smsanaltp provides a template for creating an input analysis parameter file. The template includes analysis parameters filled in with default values. Users then can edit these default values with a text editor, changing them as needed in order to obtain a better analysis. The syntax for obtaining a template is:

smsanaltp [flag option] inputsoundfile [outputanalysisfile] [> filename]

The *inputsoundfile* argument is the name of a **monophonic**, **44.1 k AIFF** or **WAV** format soundfile. The filename extensions *.aif*, *.aiff* and *.wav* can be omitted if you wish. For soundfiles in your *SFDIR* or in any of the *sflib* directories, you need type only the name of the soundfile, not its full path. For input soundfiles in directories that branch from your *$SFDIR*, include the subdirectory name(s).

The optional *outputanalysisfile* argument is the name you wish to give to the output *sms* analysis file. If this argument is omitted, the analysis file will be named *test.sms*. I recommend that you include the file-name extension *.sms* when naming all *sms* analysis files.

The optional *flag option* determines the type of template you will get. There are four variants:
(1) a *short, uncommented* template, the default version, is provided when no flag option is specified immediately after *smsanaltp* on the command line. This template includes only those analysis parameters most frequently changed from default values (such as the name of the input soundfile, the name of the output *sms analysis* file, and pitch detection parameters).

(2) a *"verbose*" (commented) "*short*" template, selected with a *-v* flag. This template includes the same parameters as #1 above, but with usage comments for each parameter.

(3) an *uncommented "long*" template, selected with a *-l* flag, which includes practically **all** *sms analysis* parameters filled in with default values.

(4) a commented ("*verbose*")" *long*" template, selected with either a *-lv* flag or with a *-vl* flag, identical to variant #3 above except that most of the parameters include usage comments.

Beginning users generally will want to select *commented* templates, while advanced users may prefer less cluttered *uncommented* templates. The *short* templates generally will suffice when analyzing comparatively simple sounds such as pitched string, wind and vocal tones. You can always add additional parameters to the template if needed. by typing them in or by cutting-and-pasting them from another file or a shell window display. Long templates can be useful when analyzing acoustically more complex sounds such as idiophones and environmental sounds, where many of the default values may need to be altered. Consult the *smsanaltp* manual page for more detailed information.

Usage examples:

(1) *smsanaltp bssn.a2*

will display a short, uncommented template for analyzing the *sflib/wind* soundfile *bssn.a2*

(2) *smsanaltp -v spinningsound spinningsound.sms > spinningsound.sms*

will create a commented short template for analyzing your soundfile *spinningsound* (or *spinningsound.wav*, *spinningsound.aif* or *spinningsound.aiff*) and will write this template into a file named *spinningsound.sms* in your current working Unix directory. The analysis file also will be named *spinningsound.sms*. Giving a common name to an analysis file, and to the parameter file used to create it, is a common practice that I recommend.

(3) *smsanaltp -l Section2/3voices.wav SMS/3voices.sms > 3voices.sms*

will write a long uncommented template for analyzing the soundfile *3voices.wav* in your soundfile subdirectory *Section2* into a parameter file named *voices2.sms* in your current working Unix directory. The analysis file that is specified in this parameter file will be *3voices.sms* in your soundfile subdirectory *SMS*.

See the *man* page for *smsanaltp* for more details.

Because *sms* analysis files can be very large, we always write them to the *snd* disk, rather than to the smaller Unix system disk. Parameter files, by contrast, are small and are written along with other ASCII files in your */home* directory or (better) in subdirectories that branch from your *home* folder.

Let us follow the practice of creating an SMS analysis file for the */sflib/wind* soundfile *oboe.bf3*. We will use a verbose short template:

smsanaltp oboe.bf3 oboe.bf3.sms > smsoboetry1

When we open the parameter file *smsoboetry1* we will see this:

```
// Include comments only at beginning of lines, NOT after analysis parameters
InputSoundFile /sflib/wind/oboe.bf3
OutputSmsFile /snd/allan/oboetry1
 // ######## GENERAL PARAMETERS ############################
  // SineModel: 0 = no pitch analysis, 1 = harmonic, 2 = inharmonic; default=1
      SineModel 1
  // window type : 0 (least smooth) to 11 (smoothest); default = 8
      WinType 8
 // Frame rate of 344.532 is optimized for most 44.1k soundfiles
      FrameRate 344.532
 // BeginPos = skip time in % into input soundfile: range 0 (beginning) to 1. (end)
      BeginPos 0
```

```
// EndPos = end time in % to stop reading input soundfile: range  0 to 1.
      EndPos 1
// ########   FUNDAMENTAL PITCH (for harmonic sounds) ###################
// PitchDetection: 1 = yes, 0 = do not perform, use DefaultPitch as reference
      PitchDetection 1
 // lowest possible, highest possible & default fundamental:
 // You almost always will want to change the 3 default values below
LowestPitch 40
DefaultPitch 75
HighestPitch 600
// ATTACK REANALYSIS parameters are useful for sounds with sharp attacks, like pizzicati
 // AttackReanalysis : 1 = analyze attacks backwards 0 = analyze attacks forward
AttackReanalysis 0
// Next 3 parameters only used if AttackReanalysis is set to 1
// nAttackReanalysisFrames: numbers of frames to use in reanalyzing attack
    nAttackReanalysisFrames 20
// AttackReanalysisLowPitchMargin and AttackReanalysisHighPitchMargin:
// multipliers for how far fundamental can deviate from DefaultPitch during atack
    AttackReanalysisLowPitchMargin 0.95
    AttackReanalysisHighPitchMargin 1.05
// ######### PARTIALS ##########################################3
 // Number of partials (sines) : 0 to 400 ; default=60
nSines 60
  // Lowest frequency to search for in input sound: 0 to 22050 hertz
LowestFreq  20
  // Highest frequency to search for in input sound: 0 to 22050 hertz
HighestFreq  11025
// ######  RESIDUAL (noise component) : ############
// ResModel : not documented; range 0 to 5; try another value if resynthesized
// noise component is bad
     ResModel 4
```

Lines beginning with a double slash *// are comments, and any line that includes this // symbol ANWHERE on the line will not be seen by SMS. Thus, you cannot include comments after a parameter, like this:*

LowestPitch 210 // **WRONG!**
This line will be deleted, and the *LowestPitch* parameter will be set to its default value of 40.

A discussion of all SMS analysis parameters is available in the document *SMS Analysis Parameters*, available in the *LINUX DOCs* binder and online on the *DOCs* page of the ECMC web site. Within this document I have included some bracketed comments to note errors and omissions within the documentation.

If you wish, you can delete parameters with default values that you are sure you will not want to change. However, do not delete the *FrameRate* value of 344.532, which is necessary for 44.1k soundfiles and which differs from the default used by the SMS binary.

The parameters are grouped into five units in the template: *General, Fundamental Pitch, Attack Reanalysis, Partials* and *residual*:

General analysis parameters

SineModel : generally set this to 1, the default, when analyzing most sounds, even speech. Setting *SineModel* to *2* may be necessary with certain inharmonic sounds, but this will not offer as many synthesis modification possibilities. Do not use a 0 or else there will be no pitch analysis, and there will be relatively few synthesis modification possibilities.
WinType : the default value of *8* usually works well, and this parameter is not frequently changed.
Always keep the *FrameRate 344.532* line, as noted above.
BeginPos and *EndPos* respectively allow us to skip into a soundfile when performing the analysis, and to

analyze only a portion of the soundfile. However, these values must be given as decimal percentages of the total soundfile length, from 0 to 1., and not in seconds. Thus, for a soundfile with a 5 second duration, to analyze only the portion of the soundfile from 1 second to 4 seconds, these values would be set to:

BeginPos .2
EndPos .8

Pitch tracking

If *PitchDetection* is set to *1* the analysis will try to extract the time varing fundamental pitch of the sound. If *PitchDetection* is set to *0* no pitch analysis will be performed and you will not be able to transpose the sound during resynthesis.

It is almost always a good idea to change the *DefaultPitch* to the perceived pitch of the sound. Use the *ecmchelp* file *herz*, or the ECMC *midinote* utility for help. This value also determines the window size, an important parameter, especially for acoustically complex sounds. Usually the *LowestPitch* and *Highest-Pitch* should be set respectively to about 80 % and 120 % of the *DefaultPitch* value to allow for scoops, vibrato and other pitch variations. Obviously a glissando may require a wider variance.

ATTACK REANALYSIS

When the *AttackReanalysis* parameter is changed from its default 0 to *1*, the beginning of the soundfile will be analyzed backwards. This can be very useful for idiophones (including stribg pizzicati and piano tones) that begin with a noise burst before settling into more periodic vibration. By analyzing the noise burst backwards, beginning at a point where the pitch and timbral spectrum are more stable, a better analysis often is obtained.

When the *AttackReanalysis* parameter is changed to 1 the other three indented *AttackReanalysis* parameters come into play. (They are ignored when *AttackReanalysis* is 0.) *nAttackReanalysisFrames* determines the number of opening frames that will be analyzed backwards. With the *FrameRate* of 344.532 frames per second, setting *nAttackReanalysisFrames* to 115 or so will cause the first 1/3 second to be analyzed backwards. The *AttackReanalysisLowPitchMargin* and *AttackReanalysisHighPitchMargin* are multipliers for *DefaultPitch* values during the backwards analysis only, when the pitch deviation from the fundamental often is much higher than during the steady state portion of the sound. You generally will want to increase the default *nAttackReanalysisFrames*, *AttackReanalysisLowPitchMargin* and *AttackReanalysisHighPitch-Margin* values.

Partial frequencies

The *nSines* parameter sets the number of partial frequencies that SMS will try to identify and track. Hwever, these sinusoids include not only steady state partials, but also brief frequency trajectories that may occur, especially during the note attack, and thus you should set this value higher than you expect — approximately double the number of presumed partials.

The goal here is to enable SMS to include all of the pitched components of the source sound within its pitch analysis, so that if the sound subsequently is transposed during re-synthesis, only the bow scrape, breath noise and/or other completely unpitched elements of the source sound will not be transposed. If you set *nSines* too low some of the weaker but still audible pitched components may be included in the residual and when you transpose the sound you might get unwanted "harmnizing" (some of the original pitch will survive). If you set *nSines* too high, the pitched component may include glitches or other artifacts. Often, hwoever, a reasonable ballpark value works just fine.

The *LowestFreq* and *HighestFreq* arguments, which often can be left at defaults, set the lowest possible and highest possible frequencies that may occur in the source sound.

Residual

Often, the *residual* is the weak spot in SMS analyses, and there are relatively few parameters that ou can adjust to try to improve a poor residual analysis. The *ResModel* parameter, which has a range between 0 and 5 and a default value of 4, is not documented, and I have no idea what defines the five "models." Still, if your pitch analysis is good, but the residual is poor, you might try changing this parameter.

Before changing the default analysis parameter values, it often is helpful to consider the overall acoustical properties of the source sound. Some fundamental considerations include:

- Does the sound have a well-defined pitch that we need to capture within the analysis, or is it essentially inharmonic, like a snare drum hit or waterfall?
- If the sound is pitched:

☞ Is the spectrum essentially harmonic, as in most woodwind, brass and arco string tones? Or is the perceived pitch actually a "strike tone" within an essentially inharmonic spectrum, as in gong tones, temple blocks and almost all idiophonic sounds? Sometimes the answer to this question requires some acoustical knowledge. With piano tones, for examples, the "harmonics" become progressively sharper, and the tone begins (and often ends) with noise components produced by the hammer and dampers.

☞ How stable is the pitch? Is there a wide vibrato (as in many sung tones)? Are there glissandi, scoops, random pitch deviations or other types of pitch inflections?

• Does the sound begin with a sharp or "noisy" attack or articulation, like piano and pizzicato tones and most idiophonic sounds? Such complex attacks can be the most difficult portion of a sound to capture successfully in an analysis.

• Does the sound change rapidly over time in amplitude and/or timbre, like most idiophonic sounds, or is there a "steady state" during most of the tone?

The */sflib/anal/sms* directory contains SMS analyses of several *sflib* source sounds that can be used for SMS synthesis. In addition, the analysis parameter files that created these analyses can be consulted for illustrations of parameter settings for various types of sounds. To obtain a listing of these example parameter files type: *lssmsex*
To view one or more of these analysis parameter files, type: *getsmsex filename(s)*

As luck would have it, one of the example analysis files is named *oboe.bf3.sms* (the same soundfile we set out to analyze above), and the example file looks like this:

InputSoundFile /sflib/wind/oboe.bf3
OutputSmsFile /sflib/anal/x/oboe.bf3.sms
FrameRate 344.532
LowestPitch 200
DefaultPitch 233
HighestPitch 270
nSines 80

These were the only parameters that had to be changed from default values in order to obtain a usable analysis of this oboe tone. But it’s not always this easy!

Creating an SMS analysis file : smsanal

Now that we have an analysis parameter file we are ready to creat the actual analysis file with *smsanal*. The *smsanal* syntax is

smsanal inputfile

where *inputfile* is the name of the analysis parameter file. The output of *smsanal* will be an analysis file with the name (and path) specified by the *OutputSmsFile* argument in our parameter file.

Actually, there is more to the *smsanal* script than meets the eye. It solves some bugs in the current version of the SMS binary, as noted in the *smsanal man* page, but to do so it make temporary copies of both the input soundfile and the analysis parameter file. If you abort an *smsanal* job these two scratch files will be left on the disk, one in your $SFDIR and one in your current Unix directory.

While an *smsanal* job is running, or after it has completed, check to see if there are any error messages, and if so check your analysis parameter file for errors.

Running smstest to test the quality of an analysis file

Before we expend a lot of time and energy devising synthesis modifications with our analysis file, we should run a quick test to see if the analysis is any good — whether it has adequately modeled both the pitched and noise components of the source sound. *smstest* and *smstestup* are local utilities designed to quickly test the quality of *sms* analysis files. Using the analysis file as input, it attempts to make an exact clone of the original source soundfile. If this straight resynthesis sounds almost indistinguishable from the original source sound we can be fairly certain that the pitched and noise components of that sound have been modeled adequately in the analysis. If we are not happy with the re-synthesis, we will need to make one or more corrections in our analysis parameter file , run it through *smsanal* again and hope for better results. *smstestup* works identically to *smstest* except that it transposes the resynthesis up one semitone to test pitch transposition of the analysis file.

The *smstest* syntax is simple:

smstest [flag option] inputsmsanalysisfile [outputsoundfile]

or

smstestup [flag option] inputsmsanalysisfile [outputsoundfile]

where *inputsmsanalysisfile* is the name (and, if the analysis is not located in your *$SFDIR*, path) of the *sms* analysis file to be tested. The optional *outputsoundfile* argument is the name of the output soundfile, which should include a *.wav* extension (if your want a WAVE format output) or else a *.aif* or *.aiff* extension if you want an AIFF format output. If you omit these extensions a *.wav* extension will be added by the script. If you omit this argument entirely, the output soundfile will be named *smstest.wav*.

Three available flag options are available:

-1 : specifies that only the pitched sinuosidal component be resynthesized

-2 : specifies that only the residual noise component be resynthesized

-3 : specifies that both the pitched sinuosidal component and the residual noise component be resynthesized

If none of these flag options is included, the default is the same as the *-3* option: both the pitched sinuosidal and the residual (noise) components of the source sound will be resynthesized.

Example command lines: (1) The command

smstest scoobeydoo.sms

will create output soundfile *smstest.wav* using both the pitched and residual data contained within the analysis file *scoobeydoo.sms*.

(2) *smstest -2 /snd/allan/SMS/insects2.sms noisetest*

Result: Output soundfile *noisetest.wav* is created, using only the noise component contained within the analysis file *insects2.sms* in my soundfile subdirectory *SMS*.

For reasons discussed in the *man* page for *smstest*:

• Even if the default pitched-plus-residual test soundfile seems successful, it often is a good idea to run *smstest* with the *-2* flag as well to isolate only the residual component of the analysis.

• I recommend that in most cases you use *smstestup* rather than *smstest*.

9.2.2. SMS Synthesis

Once you do have a good analysis file, *sms* provides many ways to modify the original sound. The procedures for preparing and then running an SMS synthesis job are similar to the procedures for preparing and running SMS analysis jobs. Because there are so many possible synthesis options, these options are consolidated within an ASCII *synthesis parameter* file. Generally this parameter file is created with the ECMC script *smssynthtp*. The default values provided by the *smssynthtp* template will produce straight resynthesis. Therefore we need to edit the template and change some of the default arguments in order to produce the desired synthesis modifications. When the parameter file is ready, we feed it to the ECMC script *smssynth* to create the synthesis soundfile. A GUI application called *smsrtsynth*, discussed later, can sometimes be useful in testing sound modification possibilities in realtime.

Most of the bugs in SMS are in its synthesis resources. Unfortunately, a few of the most powerful sound transformation resources of SMS synthesis do not work correctly, or at all, in the current Linux version of the program. Even so, there are more than enough possibilities to keep you busy for a long time. The basic types of synthesis transformations available with SMS include:

Most commmon sound modifications:

(1) Time expansion or contraction without altering the pitch

(2) Amplitude alterations in the pitched component, the residual component, or both

(3) Pitch alterations

(4) Alterations in the harmonic or inharmonic frequency spectrum (timbre)

Additional modification possibilities:

(5) "Harmonization" -- the creation of 2, 3 or 4 rhythmically synchronized output notes ("chords")

(6) What the SMS authors call "hybridization" -- various types of cross-synthesis employing interpolations between two SMS analysis files

(7) Amplitude and frequency *modulation*

(8) Enhancement -- adding new harmonic frequencies to a sound to brighten it. So far I have not achieved good results with the SMS *enhancement* parameters.

Additionally, there are many parameters for (9) *Attributes*, which is never defined or explained clearly in the SMS documentation, and for (10) *Phase Alignment*, another area that is poorly documented and with which I have had little success. SMS also includes parameters for (12) *mixing* multiple output notes beginning at arbitrary times, but unfortunately these parameters currently do not work correctly. Finally, there are a few miscellaneous (but sometimes important) parameters that do not fit any of the categories above.

Obtaining an SMS synthesis template : smssynthtp

The ECMC *smssynthtp* utility provides a template for creating a synthesis parameter file. The syntax is:

smssynthtp [flag options] inputanalysisfile [outputsoundfile] [> filename]

where *inputanalysisfile* is the name of the input sms analysis file and *outputsoundfile* is the name of the output synthesis soundfile, which should include a .wav, .aif or .aiff extension. If you do not supply an *outputsoundfile* argument the output soundfile name will be set to *test.wav*.

The optional *flag arguments* add parameters for categories 5 (*harmonization*) through 8 (*enhancement*) in the list above. Usage of these *flag arguments* is somewhat non-standard, and you definitely should read the *smssynthtp man* page to supplement the summary on that follow.

By default, with no flag arguments, *smssynthtp* provides an uncommented "short" template containing only the more frequently used SMS synthesis parameters for categories 1 through 4 ("Most common sound modifications") above. The flag options supplement this basic template with parameters for synthesis mdification categories 5 through 9 above. These flag options begin with a + rather than the usual -, and include:

+v : a *verbose* template with comments is supplied
+h : *harmonization* parameters are added
+x : *hybridization* (cross-synthesis) parameters are added
+m : amplitde and frequency *modulation* parameters are added
+e : *enhancement* (artificial harmonic) parameters are added

Example command lines:

(1) *smssynthtp trp.a4.sms*

Result: A basic uncommented template for creating a synthesis soundfile from your analysis file *trp.a4.sms* will be displayed. The name of the output soundfile file will be set to *test.wav*.

(2) *smssynthtp +v /sflib/anal/sms/kantil.3.sms himetal.wav > sms.himetal1*

Result: A commented ("verbose") basic template for synthesizing a soundfile named *himetal.wav*, using the public domain analysis file */sflib/anal/sms/kantil.3.sms*, is written to a file named *sms.himetal1* in your current working Unix directory.

(3) *smssynthtp +v +h +m SMS/myvoice.sms > smstest3-1*

Result: A commented template that includes both harmonization and modulaton parameters is written to a file named *smstest3-1*. The input analysis file will be *myvoice.sms* in your *SMS* soundfile subdirectory, and the output soundfile will be named *test.wav*.

(4) *smssynthtp +v +x*

Result: Only commented parameters for performing hybridization are displayed. There are no input or output soundfiles, and none of the basic parameters is displayed.

Synthesis parameters

A complete summary listing of the many SMS synthesis parameter arguments, along with their default values and ranges, is available in an HTML *SMS Synthesis Parameters* document available of the *DOCs* page of the ECMC web site and in hardcopy within the *LINUXDOCs* binder. I have added some bracketed comments to this document to correct some errors and to include some ECMC annotations. You should refer to this document while reading the summary guidelines that follow. See also the HTML pages titled *Descriptions of many of the analysis and synthesis parameters* within the *SMS Manual* on the *docs* page of the ECMC web site.

If we type the command

smssynthtp +v /sflib/anal/sms/pn.bf2.sms sms.pianowarp.wav > pianowarp1

the resulting file *pianowarp1* will contain a commented basic template that looks like this:

```
// Fn indicates a time varying function can be applied to the parameter
// Do not use blank lines. Put all comments at the very beginnings of lines.
InputSmsFile /sflib/anal/sms/pn.bf2.sms
OutputSoundFile sms.pianowarp.wav
// ########   GENERAL PARAMETERS ################
// SamplingRate: sms default is 22050; ECMC default =  44100
SamplingRate 44100
// Type: 1 thru 7, def. 7, 1 = sine only, 2 = resid. only, 3 = sine + resid.
Type 3
// TimeStretch  .0001 to 1000, def. 1 : time expansion or contraction multiplier
TimeStretch 1
// NoUnvoicedTimeStretching: if != 0, do not time warp unvoiced noise bursts (e.g. speech consonants or the snap of a pizzicato)
      NoUnvoicedTimeStretching 0
// PhaseAlign def. 1 ; reset to 0 or pitch transposition will not work
PhaseAlign 0
// #####  AMPLITUDE  ######################
// Overall pitched & residual amplitude multiplier : 0 to 5, def. = 1
Amp 1
// AmpSine: Amp. multiplier for pitched component: 0 to 5, def. = 1
AmpSine 1
// Amp. multiplier for even harmonics including fundamental: 0 to 5, def. = 1
  AmpSineEven 1
// Amp. multiplier for odd harmonics: 0 to 5, def. = 1
  AmpSineOdd 1
// AmpSineList: h1 a1 h2 a2 etc. : h values = harmonic numbers & a values = amp. multipliers
//   AmpSineList 0 1. 1 1. 2 1. 3 1. 4 1. 5 1. 6 1. 7 1. 8 1.
// AmpSpec: Amp. multiplier for residual (noise) component: 0 to 5, def. = 1
AmpSpec 1
// ResCombfilter : 0 (default) = do not use, 1 = apply comb filter
      ResCombfilter 0
// ##### FREQUENCY ##########################################
// FreqSine: multiplier for all partial frequencies, 0 to 5., def. =1
FreqSine 1
// FreqSineEven: multiplier for fundamental & even partials, 0 to 5., def. =1
  FreqSineEven 1
// FreqSineOdd: multiplier for odd numbered partials, 0 to 5., def. =1
  FreqSineOdd 1
// FreqSineStretch: -.99 to 5, def. = 0, multiplier for ratio of fund. & highest harmonic
  FreqSineStretch 0
// FreqSineShift: 0 to 10000, def. = 0; hertz added to all partials
  FreqSineShift 0
// SameSpectralEnv 0 = transpse formants with pitch, 1 = keep original formants
SameSpectralEnv 0
// VibratoWeight : 0 to 10, default = 1 : multiplier for vibrato depth
      VibratoWeight 1
```

Here is a quick summary of how most of these parameters work:

General parameters :

Type : if set to 1 only the pitched component is synthesized; if set to 2 only the noise component is synthesized; if set to 3 both the pitched and noise components are synthesized; (do not worry about options 4 through 7, which require a soundfile with the noise component rather than the residual data within the analysis file)

TimeStretch : multiplier for duration; default = 1; if > 1. time stretching results; if < 1. time compression

results
NoUnvoicedTimeStretching : if changed from the default 0 to 1, noise bursts (such as consonants in speech are singing) are not altered in duration when time warping is in effect
PhaseAlign : an important and sometimes troublesome parameter; the template default is 0 (the original phases of partials are not maintained); if you get garbage in synthesizing a sound, especially when transposing the pitch, try resetting *PhaseAlign* to *1*; setting this parameter to 0 has generally worked better for me than setting it to 1, but not always
Amp : amplitude multiplier for the entire signal (both pitched and noise components)
AmpSine : amplitude multiplier for only the pitched component
AmpSineEven : amplitude multiplier for the "even" harmonics, including the fundamental, which is considered harmonic number 0 here; most people would consider these to be the "odd harmonics"
AmpSineOdd : amplitude multiplier for "odd" harmonics only, but since SMS numbers the fundamental as "harmonic number 0" rather than "harmonic number 1," most people would consider these to actually be the "even" nubered harmonics
AmpSpec : amplitude multiplier for only the residual component
ResCombfilter : if changed from the default 0 to 1, the residual will be comb filtered; it generally is not necessary to do this, but it may help if you are having trouble with the quality of the residual
FreqSine : pitch multiplier; .5 = 1 octave down, 1.5 = perf. 5th up, etc.; see the *ecmchelp* file *pitchratios* for help
FreqSineEven : multiplier for only the "even" numbered partials (these actually are the "odd"numbered partials including the fundamental)
FreqSineOdd : multiplier for only the "odd" numbered harmonics (actually the even numbered partials)
FreqSineStretch : if not set to 1. the sound will be detuned; if greater than 1., the partials will be further apart in frequency than in the original sound; if set to 1.33, for example, the highest partial will be raised in frequency by 33 %, and all other partials will be raised as well; if *FreqSineStretch* is less than 1., the partials will be closer in pitch than in the original sound; with a value of .75, the frequency of the highest partial will be only 75 % of its original value
FreqSineShift : hertz added to each partial frequency; another parameter that de-tunes the timbre; if set to 17.5, 17.5 hertz will be added to each partial frequency
SameSpectralEnv : flag; default is 0, no effect; if set to 1, SMS will try to maintain the original formants (emphasized frequency bands) when a sound is tranposed
VibratoWeight : a multiplier for the depth of pitch vibrato inthe source sound; default is 1, which has no effect; if set to 0 vibrato is removed; if set to a value greater than 1., the vibrato will be exaggerated

Editing the synthesis parameter file template

Let make some changes to our template above to alter the sound:
TimeStretch 2.5
This will stretch the piano tone to 2.5 times its original duration
NoUnvoicedTimeStretching 1
With this flag changed to 1, the initial attack of the piano tone will not be affected by the time stretch
FreqSine .749
This will transpose the sound down a perfect fourth.
SameSpectralEnv 1
This will cause the original formants of the tone to be retained, rather than be shifted down a perfect fourth because of the pitch transpositionwe are performing.
FreqSineShift 17.3
By adding 17.3 hertz to each partial, this will detune the piano tone, which will sound sharp, "out of tune" and somewhat like a bell or gong.

Be sure to read the *smssynthtp man* page for instructions on editing synthesis parameter templates. Comments should only be placed at the beginnings of lines, never after parameters. (All lines that include the comment symbol // anywhere on the line will be ignored.) Do not include blank lines, and place only one parameter and its argument on each line.

Creating an SMS synthesis file : smssynth

Once you have edited the synthesis parameter file to your satisfaction, use the ECMC script *smssynth* to create the synthesis soundfile:

smssynth parameterfile

where *parameterfile* is the name of your edited synthesis parameter file. As discussed in the *man* page for this script, *smssynth* corrects a couple of irksome bugs in SMS when it is run in synthesis mode.

Major bugs and limitations

Two of the most powerful syntehsis resources of SMS, both of which work very well in ur older SGI version of SMS, are broken in the current Linux version we are running.

(1) Multiple output "notes"

The SMS authors frequently refer to *synthesis parameter files* as "*score*" files and recommend naming them with a *.sco* extension. Using the *Mix* and *BeginEventTime* synthesis parameters, it should be possible to define multiple output "events" ("notes"), beginning and ending at different times and employing different syntehsis modifications, within a single synthesis parameter file, somewhat as we do with Csound score files. However, the *Mix* parameter is partially broken. It can only be used to write the samples created by the last "note" defined within a synthesis paramter file to a pre-existing soundfile. (See ECMC example files *sms3* through *sms3-10* for examples.) So for now, with the exception of *harmonization* (in which the two, three or four output "notes" begin and end simultaneously, as a "block chord"), SMS synthesis is limited to creating a single output "note."

(2) Time varying functions (envelopes)

In the SMS documentation of analysis and synthesis parameters you might note that many parameters are designed to accept not only *constant* arguments, but also time varying *function* definitions. SMS functions are defined by pairs of time/value breakpoints:

time1 value1 time2 value2 ... timeN valueN

For example, to apply an amplitude fade-in during the first 5 % of the total duration of a synthesized "note," and a fade-out during the final 25 % of the synthesis duration, we could apply this function definition to the *Amp* paramter:

Amp 0 0 .05 1. .75 1. 1 0

However, FUNCTIONS DO NOT WORK CORRECTLY in the current SMS, and the function above would cause discontinuities. Functions WILL create an audible change in the parameter, but only during the first one second duration of the synthesis. With complex functions involving several breakpoint pairs, often only the last envelope segment will have any effect.

Because envelope functions do not work correctly, amplitude fades, , pitch glissandi and similar types of time varying parameter changes generally will not work correctly.

Example synthesis parameter files and soundfiles

Use the *lssmsex* and *getsmsex* commands to list and display ECMC example SMS synthesis parameter files. All of the *synthesis parameter* files have been compiled into corresponding example soundfiles with the same name within the */sflib/x* directory. Here are some notes on these example files:

sms1-1 and *sms1-2* : These two similar examples illustrate the two most basic uses of SMS: for time warping and for pitch transposition. In both of these example files, a violin tone is doubled in duration (*TimeStretch* is set to *2*) and transposed down a minor sixth (*FreSine* is set to .63). The amplitude (*Amp* argument) also is reduced to avoid clipping. The only difference between these two examples is that in *sms1-2* the *SameSpectralEnv* is changed from the default 0 to 1, so that the formants of the original violin tone are retained and the resulting tone sounds more "nasal" and more like a violin. (Example *sms1-1* has a "mellower" timbre, more like that of a viol, because the formants as well as the pitch have been shifted down).

Examples *sms2-1* through *sms2-7* illustrate independent control of the "even" numbered and "odd" numbered harmonics. However, what *sms* calls the "*even*" numbered harmonics actually include the fundamental and harmonincs 3,5,7, etc., while what *sms* calls the "*odd*" numbered harmonics actually includes harmonics 2,4,6, etc). (The SMS authors apparently consider the fundamental to be partial number 0 rather than partial number 1. This is very annoying!) In all of these examples a violin tone is transposed up a major second and *SameSpectralEnv* is set to *1* t oretain the formants of the original tone.

In example *sms2-1* only the "even" (actually the "odd") numbered harmonics are synthesized, along with

the residual, and the timbre consequently sounds somewhat like that of a stopped pipe (e.g. a clarinet). In *sms2-2* only the "odd" (actually the "even") numbered harmonics are synthesized, and the pitch of the tone therefore sounds an octave above the fundamental.

sms2-3 illustrates a "morph" from only the "even" (actually the "odd") harmonics at the beginning of the tone to only the "odd" (actually the "even") harmonics by using time varying functions with opposite trajectories for the *AmpSineEven* and *AmpSineOdd* parameters. Remember that time varying FUNCTIONS DO NOT WORK PROPERLY in the present version of SMS. In this case, the "morph" should occur over the entire duration of the note, but instead occurs too quickly over the first one second of the tone. Even though this is not correct, the example still illustrates "morphing" between the two sets of harmonics. (To hear how this example is *supposed* to sound, import this synthesis parameter file into *smsrtsynth*).

More tricks: Example *sms2-4* is identical to *sms2-1* except that the "even" (actually the odd) numbered harmonics are transposed up an octave. Example *sms2-5* is identical to *sms2-2* but now these harmonics are transposed down an octave so that the fundamental pitch is heard. Example *sms2-6* synthesizes both sets of harmonics, but tranposed as in *smsex2-4* and *smsex2-5* so that the odd harmonics become the even harmonics and vice versa. Finally, exmaple *sms2-7* is identical to example *sms2-6* except that *SameSpectralEnv* is set to *0* so that the original formants are NOT retained, and the timbre sounds quite different.

Example soundfile */sflib/x/sms3* actually is a composite, or mix, of ten synthesized violin pizzicato tone transformations created by example SMS synthesis paramter files *sms3* and *sms3-2* through *sms3-10*. In all 10 tones the pizzicato note is doubled in duration (*TimeStretch 2*) but not during its initial attack (*NoUnvoicedTimeStretching 1*. By setting the *Mix* parameter to *1* in parameter files *sms3-2* through *sms3-10* we mix the synthesis samples into a pre-existing soundfile (initially created by parameter file *sms3*) rather than creating a new soundfile. This example illustrates detuning using the *FreqSineStretch* parameter, and also different timbral results that often can be achieved by setting the *Phase Align* parameter either to 1 or to 0). See example file *sms3* for the details.

Examples *sms4-1, sms4-2, sms4-3* and *sms4-4* apply downward transposition and then frequency shifting to a soprano tone to detune the timbre and shift the tone into various registers.

Examples *sms.harm1* and *sms.harm1* illustrate harmonizing.

Example *sms.mod* first mangles the timbre of a piano tone (tranposing the tone down a major second, tranposing the even harmonics down further by slightly more than an octave, and tranposing the odd harmonics up by slightly more than an octave) and then applies both amplitude modulation (tremolo) and frequency modulation (vibrato) to the resulting sound.

Example *sms.hyb1* presents one possible hybridization of a gamelan metallohpne and a bass choir tone. tone.

Hybridization : caveat emptor

SMS "hybridization" (or cross-synthesis) requires the use of two analysis files within a synthesis parameter file. Because timevarying functions do not work correctly in our current version of SMS, hybridization *morphing* procedures (e.g. a piano tone seemlessly transforming into a vocal tone) do not work, except under very limited conditions.[2]
Still, it is possible to create intriguing new sonic offspring from two "parent" sounds, so long as the parameter relationships between the two sources remain fixed throughout the duration of the synthesis "note."

Hybridization between two analysis files can quickly become very complicated. First, one can make all of the usual synthesis modifications, such as pitch transposition, to the data from the "source" (initial) analysis file. Then, for many synthesis parameters, one can choose whether

(1) to use the data from the "source" (initial) analysis file (if the corresponding *hybridizing* parameter is set to *0*); or

(2) to use instead the data for this parameter from the hybridizing (second) analysis file (if the corresponding *hybridizing* parameter is set to *1*); or

[2] Several of the available SMS hybridization parameters, such as *HybrizeEnv* and hybridizing parameters whose name includes the string "*weight*", are specifially designed for "morphing." I have not included most of these parameters within the hybridizing template provided by *smssynthtp*, but a few of them are necessary, and require edits to *two* parameters to implement a change in the sound.

(3) if the hybridizing parameter is set to some value between 0 and 1., to interpolate a value somewhere in between the values in these two analysis files

Under our current version of SMS I do not recommend trying to employ *hybridization* procedures, especially for beginning SMS users. Since functions do not work correctly it can be difficult or sometimes even impossible to line up the two analysis files temporally in a manner that yields usable musical results. Hybridization procedures are documented poorly by the SMS authors, and sometimes you may have little of value to show (or hear) for all you efforts. If you *do* try employing hybridization procedures, I recommend changing only one or two parameters at a time and, step by step, seeing (hearing) what works and what does not. And don't be afraid to bail out if you are not making any progress.

9.2.3. Using smsrtsynth to perform real time synthesis tests

The SMS distribution includes a graphical application called *smsrtsynth* ("*SMS* **real time** *synthesis*") that can be useful in testing out synthesis possibilities. This application is rather primitive in several respects — there are no *Preferences* settings to set default paths, for example, and the program has limitations and bugs. Notably, you cannot write a soundfile with *smsrtsynth*; you can only test changes in synthesis parameter settings, listening to the results in real time. However, when you have arrived at settings that produce a good ressult, you can export these settings into a "score" file (a synthesis parameter file) and, with a little additional work, then use *smssynth* to create a synthesis soundfile with these settings. I am hesitant to recommend *smsrtsynth* because it can be buggy at times. Sometimes it will become "stuck" and, regardless of any parameter changes you make, not respond to these changes. Still, the application sometimes can save you a lot of typing and time when experimenting with synthesis possibilities.

smsrtsynth will only display synthesis parameters that have been loaded. By default, the program opens with a blank window, and in order to edit parameters you must click on the word *Parameters*, then on *Add*, then highlight the parameter or (consectutive) group of parameters you want displayed and click on *Select*. This is tedious when you want to load displays for several parameters. Groups of parameter displays, called "Properties," can be saved to a file and then loaded. I have created several "properties" files, with *.prp* filename extensions, that you can load when you first open *smsrtsynth*. First copy these "property" files to the directory in which you will be working with *smsrtsynth*, by typing:

getsmsprp

The following *.prp* files will be copied to your directory:

smsrtsynth.prp : contains "properties" for editing the most commonly used SMS synthesis parameters, similar to the "basic" short template obtained with the *smssynthtp* command with no options

The following three "properties" files include all of the parameter displays available in the *smsrtsynth.prp* file, but also add more displays for editing additional parameters:

smsrtsynth.harm.prp : adds widgets for editing *harmonization* parameters

smsrtsynth.mod.prp : adds displays for editing *modulation* parameters

smsrtsynth.hyrbid.prp: adds displays for editing *hybridization* parameters

One of the bugs in *smsrtsynth* is that it does not properly load analysis files. Before using *smsrtsynth*, therefore, you first should create a synthesis parameter file, using *smssynthtp*. When you first open *smsrtsynth*, begin by loading the *.prp* file that suits your needs. To load a basic "property" template, click on *File*, then on *Load Properties*. In the window that opens, if you see the message "No matching files," click on the **.prp* button and then select the appropriate *prp* file.

Next load a previously created synthesis parameter file (also called a "score" or *.sco* file in SMS), which must include the name of the analysis file to be used. Click on *File*, then on *Load score file* and select the synthesis parameter file with which you want to experiment.

Begin making changes in the parameter values and press the *Play* button to hear the results. Changes can be made in real time, and you can use the *Loop* button to the right of the *Stop* button to loop the synthesis continuously. To remove a parameter, select it and click on the *Remove* button. To add additional parameters for editing click on the Add button.

When you have arrived at settings that merit saving, click on *File* and then on *Export score file* and save your settings to an ASCII file. This file will look somewhat more complicated than those you create with *smssynthtp* because each parameter will be defined as a time varying function:

AmpSine 0 1.000000 1 1.000000

This means that at time 0 (the beginning of the note) *AmpSine* will have a value of 1.00, and at time *1* (the end of the note) it will (also) have a value of *1.00*. Alas, the resulting synthesis parameter file is not yet ready for use by *smssynth*. If you display the file, you will see that *smsrtsynth* does not save the filename parameters or the names of any input or output files when it exports parameter values, and it does export a setting for real time output (rather than writing the synthesis to a soundfile), which you do not want. To prepare the exported file for use by *smssynth* use the ECMC utility *fixsmsscore*, which has the syntax:

fixsmsscore inputparamterfile analysisfile [outputsoundfile]

where

nputparamterfile is the name of the input sms analysis file ;

nalysisfile is the name of the sms analysis file to be used for synthesis and

utputsoundfile is the name of the output synthesis soundfile, generally including a .wav, .aif or .aiff extension. The default soundfile name, if you omit this argument, is *test.wav*.

Example: *fixsmsfile cellotest1 vc.gs2.sms cellotest1.wav*

Result: The synthesis parameter file *cellotest1*, created with smsrtsynth will be edited to that it can be used by smssynth to synthesize a soundfile. The following 2 lines will be added at the top of the file:

InputSmsFile /snd/allan/cellotest1.sms

OutputSoundFile cellotest1.wav

and the line setting the *SynthesisOutput* to 2 (for real time playback) will be removed. File *cellotest1* now will be ready to use with *smssynth*.

See the *fixsmsscore man* page if you want more information.

If *smsrtsynth* gets in a weird state and does not respond to your edits select the last parameter you changed, click on Properties and then on OK and if you are lucky *smsrtsynth* will begin responding again. If not, export your settings to a file (if you want to save your work), quit and reopen the app.

9.3. LPC analysis and resynthesis procedures

Linear predictive coding (lpc) algorithms are based on estimation procedures (mathematical operations that try to predict what will come next on the basis of what has already occurred within a linear system).[3]

These techniques have applications in many fields. For musical purposes, linear prediction routines most often are used to determine the time-varying resonances, or formant frequencies, of a given sound. This data is stored as filter coefficients in a file. These filter coefficients can then be used to resynthesize the original sound, usually with changes in timbre, duration, or pitch, or to "cross-breed" it with some other sound to produce a hybrid offspring with some characteristics of both "parents." A "talking cello" would be a simplistic example of such cross-synthesis.

Most of the *lpc* programs and utilities we use are based upon procedures originally developed by Paul Lansky and others at Princeton University, and subsequently modified at MIT. The musical results of these techniques are perhaps best known through such compositions as Lansky's *Idle Chatter, Just More Idle Chatter, Guy's Harp* and *Six Fantasies on a Poem by Thomas Campion.* Charles Dodge, best known today for his work with fractal compositional algorithms, also created several works in the early to mid 1980s based almost entirely on *lpc* techniques. Many compositions realized here at the Eastman Computer Music Center, including compact disc recordings of my own computer works, also make extensive use of *lpc* and related analysis/resynthesis techniques.

Like *phase vocoder, sms* and other types of *analysis/resynthesis* procedures, linear prediction is a two-step process that involves

(1) analysis of a sound source (almost always monophonic)
followed by

[3] Tutorial introductions to *lpc* procedures are included in *An Analysis/Synthesis Tutorial* by Richard Cann (reprinted on pages 114-144 in *Foundations of Computer Music*, edited by Roads and Strawn), and in an article by Paul Lansky in Chapter 1 of *Current Directions in Computer Music.* A more technical and detailed introduction can be found in *The Use of Linear Prediction of Speech in Computer Music Applications* by James A. Moorer, *Journal of the Audio Engineering Society*, Vol. 227 number 3, March 1979, beginning on page 134. The *LPC* discussion within Roads' *Computer Music Tutorial* also is recommended.

(2) resynthesis operations, in which we use this analysis data to create a new soundfile.

The *lpc* analysis stage itself is generally a two-step process, consisting first of a *spectral* analysis followed by a separate *pitch* analysis, which then is merged into the spectral (*lpc*) analysis file.

lpc techniques can be fun to use, but they can also be time-consuming and at times frustrating. A usable, high quality *lpc* spectral and pitch analysis frequently is more difficult to obtain than a correspondingly usable phase vocoder analysis of the same soundfile. Several tries, or possibly some "massaging" (editing and smoothing) of the analysis data may be necessary to achieve the desired result, and *lpc* procedures work much better in some instances than in others. Determination and patience often (but not always) will be rewarded. The more complicated your initial sound source or resynthesis procedures, the more likely you are to encounter problems.

On the up side, however, *lpc* procedures offer some unique possibilities for modifying, shaping and combining acoustic sounds. Pitch, timbre and duration can become independently controllable dimensions. Once we have created a successful analysis of a soundfile, we can use this analysis data to create any number of resynthesized variations or transformations of the original sound. A door slam might be stretched to a duration of six seconds, or transposed to the register of a bass drum, or "played" as a pitch melody, or be used as a resonator for violin tones.

Eastman software for performing lpc analysis and resynthesis

☞ At Eastman, the recommended way to perform an *lpc* **analysis** on a soundfile is with **mixviews**:[4]

☞ To perform *lpc* **resynthesis**, two methods are recommended:

(1) **mixviews** : Generally this is the easier method, and is recommended for resynthesis of isolated sounds

(2) **Csound** : Eastman Csound Library algorithm *resyn* provides many modification possibilities for resynthesis, while a Library module called *xsyn* offers similar possibilities for cross-synthesis between two sound sources. These algorithms are particularly recommended when you want to create *many* resynthesized sounds (e.g. rhythms, melodies, chords, complex textures, and so on).

9.3.1. LPC Analysis Procedures

An *lpc* analysis consists of filter coefficients that represent the strongest formants (resonances), and their relative amplitudes, of the source sound at evenly spaced time intervals, called *windows* or *frames*.

The resulting analysis data is in a binary floating point format, and includes a special type of header. This data cannot be played or otherwise used like a normal soundfile; it can be displayed only by *lpc* software designed for this purpose (and not by Unix programs such as *cat*), and it is useful only as input to *lpc* resynthesis software.

Some good news: Unlike the many phase vocoder analysis procedures surveyed earlier, *mixviewswrites analysis files in a format that can can be used for subsequent lpc* resynthesis either by *Csound* or by *mixviews*.

Spectral analysis parameter values

When performing the spectral (formant) portion of the analysis, the user must make two basic decisions:

- how many filter *poles* to use, and
- how many analysis frames to create for each second of sound; this value can be provided either in terms of a *frame offset* argument or else in terms of a *frame rate* argument

In setting these values, and also the the pitch parameter arguments that will follow, keep in mind that, for historical reasons, and to conserve disk space, the default values generally are optimized for lower sampling rates, such as 22k, rather than for 44.1k soundfiles. In fact, in some cases *lpc* resynthesis procedures will work *better* with 22k soundfiles than with 44.1k source sounds. However, most of your work likely will be

[4] An alternative standalone program called **lpanal**, distributed with *Csound* and documented at the end of the *Csound* manual, provides a Unix command line syntax for performing *lpc* spectral and pitch analysis. However, as of this writing (Csound version 3.49), there are a number of problems with this program, and it is not recommended. Unless the recently added *-a* flag is included on the command line to perform pole stabilization, analyses of speech and other sounds that change rapidly in timbral formants tend to produce obnoxious chirps and pops. If a *-a* flag is included, the resulting analysis file is only readable by Csound unit generator *lpreson*, not by the more powerful *lpfreson* opcode used in Eastman Csound Library algorithms *resyn* and *xsyn*.

done on 44.1k soundfiles, and you will want to change some of the default values to achieve better results.

	Analysis: *mixviews*		**Resynthesis** *mixviews*	Csound
	Default	Maximum	Maximum	Maximum
Number of *poles*	34	64	64	c. 50
Frame offset or *Frame rate*	220 (0)	500		

The *Poles* argument determines the number of separate filter coefficients (partial frequencies) that will be analyzed and written into each analysis frame. The more complex the timbral spectrum of a source sound, and the lower its pitch, the higher you should set this value. A value of 34 poles specifies that the 17 strongest frequency components of the source sound will be extracted. In setting the *poles* value, the basic rule of thumb is:

> Determine how many spectral frequencies you want to analyze. Double this number and, if it does not exceed the maximum value allowed, use it as your *poles* argument. 44.1k soundfiles often require higher values than 22 k soundfiles. Note, however, that while up to 64 pole can be used in resynthesis with *mixviews*, Csound has a lower limit, which seems to be about 50, and a Csound job will abort when it encounters an *lpc* analysis file with "too many" poles.

When analyzing 44.1k sounds with complex timbral spectra, such as a tone in the bottom register of the cello, or most percussive sounds, values such as 40, 50 or (if you will be using *mixviews* rather than *Csound* to perform resynthesis) even 64 may be appropriate. When analyzing acoustically simpler sounds, such as a flute tone that may contain only eight or ten significant partials, *pole* arguments between 16 and 24 often work well. Specifying too many poles may result, upon resynthesis, in amplitude beating, unwanted "reverberation," "flanging," smearing, or other types of spectral distortion, especially when the pitch is transposed or the duration is stretched.

When analyzing vocal sounds, a value of somewhere between 20 and 30 generally is recommended for an initial try. If the resynthesis sounds thin, dull or muffled, try increasing this value, but keep in mind your ultimate resynthesis goals. Higher *pole* values may produce a "warmer" resynthesis if the pitch is not transposed, but if you *do* eventually perform pitch transposition, or shift the formants, the original pitch of the source soundfile may bleed through, producing unwanted "harmonizing."

> For advanced users: Sometimes, deliberately "underspecifying" the number of poles — say, with a value of 16 or so for a rich vocal bass or trombone tone, or for a percussive sound — can create interesting timbral effects upon resynthesis, simplifying the timbre such that only its "skeleton" remains.

The *Frame offset* argument, given in samples, determines (somewhat indirectly) how frequently the analysis will be updated (that is, how many analysis windows will be created to represent each second of sound). With the *mixviews* default value of 220, the first analysis frame will begin at sample 0, the second frame at sample number 220, the third frame at sample number 440, and so on.[5]
If we are analyzing a 44.1k soundfile the *lpc* analysis will be updated roughly every five milliseconds

[220 / 44100 = .0049886 seconds]

and the *frame rate*, which equals the *sampling rate* divided by the *frame offset*, or the number of frames per second, will be 200.45

[44100 / 220 = 200.45455 frames per second]

Almost surely, this update rate is much higher than necessary for good signal representation. An analysis update rate of 100 frames per second is sufficient for all but the most complex and rapidly changing timbres and amplitudes. In fact, at higher sampling rates the default *frame offset* value of *mixviews* can cause overspecificity (unduly large analysis files, needlessly long run times for the analysis and often, in fact, a poor analysis, since each analysis window encompasses such a tiny portion of the source sound). The

[5] For the curious: Actually, this is an oversimplification. *lpc* analysis programs typically create double the number of user-specified analysis frames, which overlap (sharing many of the same samples with adjacent frames, as in phase vocoder analysis procedures) in order to prevent discontinuities during resynthesis.

default *frame offset* value provided by *mixviews* actually is optimized for 22k soundfiles, where it will create updates every 10 milliseconds, or 100 frames per each second of sound.

☞ When analyzing **44.1k** soundfiles, therefore, you generally will achieve better results by changing the **frame offset** value to **400 or 500**.

Most users rarely fiddle with the alternative *Frame rate* argument (which, again, equals the *sampling rate* divided by *frame offset* value). However, if you wish, you can set this argument to a desired value, which *mixviews* then will use in place of the currently specified *frame offset* argument.

Setting pitch analysis argument values:

For string, wind, sung and spoken sounds, accurate pitch data is essential to achieving high quality resynthesis. However, pitch tracking sometimes is the most difficult or least successful portion of *lpc* analysis. In fact, many noise-like, percussive or otherwise aperiodic timbres (such as a maraca roll) may not yield a usable pitch analysis, and the spectral analysis data may be all that is needed for successful resynthesis. This can be true even for such seemingly pitched, or quasi-pitched, timbres as crotales and temple blocks.

The more complex the pitch contour and spectral evolution of a sound source, the more likely the pitch extractor algorithm will have trouble accurately tracking the pitch. Fundamental frequencies below 100 herz, or above 1000 herz, may be difficult or even impossible to capture. Glissandos, rapid or wide vibratos and other pitch inflections may not be captured accurately. The rapidly changing pitch of spoken sounds can be particularly difficult to capture.

In this area, as in certain aspects of *lpc* spectral analysis, *mixviews* provides some handy tools unavailable with *LPC.app*. If we can get close to an accurate pitch analysis with *mixviews*, we can applying a simple smoothing operation, provided in the application, that can significantly improve the quality of the pitch data.

The principal *pitch analysis* arguments, and their default values, are summarized in the following table:

Pitch analysis arguments:		**mixviews**	
		Default	Maximum
Frame size	(number of samples in each frame)	350	1000
Frame offset	(number of frames per second)	200	505
Frame rate		(0)	
Highest estimated frequency (herz)		1000	
Lowest estimated frequency (herz)		100	

The *frame size* value should be sufficiently large so that each frame encompasses at least one complete frequency cycle of the waveform. The maximum value allowed by *mixviews* is 1000. The default value of 350, again, seems to be optimized for 22k soundfiles.

☞ For 44.1k soundfile, this default often should be raised to about 500.

☞ Use higher values for *low* pitched tones, lower values for *high* pitched tones.

(The lower the pitch of a tone, the longer its wavelength period, and the more samples required to represent each cycle).

Generally, the *Frame offset* argument should be set to about 1/2 the *frame size* value.

To provide the analysis program with some initial help, so that it does not mistake a strong harmonic or a resonant frequency for the fundamental pitch, give careful attention to the *High estimate* and *Low estimate* arguments. These two arguments respectively specify the highest and lowest possible fundamental frequencies that you believe may occur within the soundfile. If you know the approximate pitch of a source soundfile, you can use the *help* file *herz*, or else the Eastman *midinote* script, to find the frequency of this pitch. The narrower the range between the *High boundary* and *Low boundary* values you provide, the more likely your pitch analysis will be successful. However, be careful not to specify a pitch range that is too narrow, or the analysis may miss pitch inflections, especially during attacks. Also, remember that the perceived pitch of a sound does not always coincide with the physical frequencies of a timbre, particularly in the case of the strike tones of percussive sounds.

Example lpc and pitch arguments :

To help users in their initial attempts to master *lpc* resynthesis and cross-synthesis procedures, we have placed several *lpc* analysis files in the */sflib/anal/lpc* directory of both *arcana* and *syrinx*. These analysis files also have been used in example scores for Eastman Csound Library algorithms *resyn*, *xsyn* and *lpcpitch*. The table below provides most of the spectral and pitch analysis arguments used to create some of these public domain analysis files.

	LP ANALYSIS				*PITCH ANALYSIS*			
	Poles:	*Frame offset:*	*Frame rate:*	*Dur:*	*Frame size:*	*Frame offset:*	*High est.:*	*Low est.:*
[default mixviews argument:]	[34]	[200]		[EOF]	[350]	[200]	[1000]	[100]
Analysis file:								
sop1.fs4.lpc	24	441	100	8.47	1000	441	400	320
source soundfile: *sflib/voice/sop1.fs4.snd*								
crt.fs5.lpc	34	200	110.25	5.46	Not performed			
source soundfile: */sflib/perc/crt.fs5.snd*								
22voicetest.lpc	34	200	100.25	7.26	500	200	310	90
source soundfile: *sflib/x/voicetest.snd* (22k)								
maracaroll.lpc	34	400	100.25	7.26	Not performed			
source soundfile: */sflib/perc/maracaroll.snd* (44k version)								
vc.p.c3.lpc	34	200	220.5	2.28	350	150	200	125
source soundfile: *sflib/string/vc.p.c3.snd*								
sdrum1.broll.lpc	34	504	87.32	1.64	Not performed			
source soundfile: */sflib/perc/sdrum1.broll.snd*								
oboe.d4.lpc	34	200	220.5	2.62	1010	200	350	261
source soundfile: *sflib/wind/oboe.d4.snd*								
fl.e4.lpc	20	504	87.32	3.47	1010	505	369	310
source soundfile: *sflib/wind/fl.e4.snd*								

Notes on these analysis files:

• Note that the sampling rate of the *voicetest* source soundfile is 22k rather than 44.1 k, simply because I had better luck analyzing a 22k copy than the 44.1 k original soundfile. When using an analysis file derived from a 22k source soundfile (including any analysis files within */sflib/anal/lpc* whose name begins with the prefix *22*), we also must employ a 22k sampling rate in our resynthesis jobs.

• Although it might at first seem surprising, no pitch analysis is required for successful resynthesis of the crotale tone (*crt.fs5.lpc*); the *f#5* pitch we hear is a strike tone — a result of certain quasi-harmonic ratios within a basically inharmonic timbral spectrum.

• The snare drum brush roll analysis (*sdrum1.broll.lpc*) captured the lower frequencies of the source sound better than the higher frequencies. (One can compensate for this, somewhat, by boosting the *brightness* score parameter for Library algorithm *resyn*, or by processing the resynthesized soundfile through an EQ network, such as with the *gQ* application, and boosting higher frequencies.) In addition, the analysis file produced an attack and decay that are more abrupt than the original.

• The cello pizzicato tone analysis (*vc.p.c3.lpc*) results in significant amplitude loss in resynthesis, which can be restored by boosting the amplitude multiplier argument in *resyn* to about 2.0.

Special problems in analyzing spoken and percussive sound sources:

The analysis and resynthesis of speech presents some special problems, and it is recommended that you do not attempt to analyze and resynthesize spoken sound sources until you have gained some experience with *lpc* analysis and resynthesis procedures. The three most vexing problems are

• the rapidly changing pitch inflections of speech, which in some cases may exceed the capabilities of *lpc pitch tracking* algorithms;

• rapid changes in timbral spectrum, particularly on consonants; and, especially

• the many brief silences (e.g. from glottal stops) that often occur within speech.

Occasionally, the *lpc* analysis values that result from consonants, brief silences and near-silences produce horrendous noise bursts (resynthesis amplitude values far in excess of 32767). With *mixviews*, these generally can be eliminated by performing pole stabilization and smoothing smoothing operations on the

analysis data.

For similar reasons, quasi-pitched percussive sounds with very sharp attack and decay transients can be difficult or impossible to analyze and resynthesize successfully. For example, I have had little luck with tom toms, timbales or most other types of drum sounds.

The problem of silences within sounds can be very annoying. *lpc* analysis programs, including *mixviews*, typically have difficulty even when confronted with brief silences (as short as 20 or 30 milliseconds, if this duration is greater than the duration of a single analysis frame), and often will abort with an error message such as

"*Cannot analyze frames with zero amplitude*" (mixviews)

or the cryptic "*gauss ill conditioned*." If an analysis aborts with such an error message, first determine the source of the problem:

• If you are analyzing a continuous tone or sound, the silence may well be at the very beginning or very end of the sound, due to inadequate trimming. Try correcting this by eliminating this beginning or ending silence. (With *mixviews*, make a selection by dragging over a the input soundfile waveform, but leave out the silence.)

• If you are analyzing speech or someother sound source that may include several brief silences, the best solution is to give the analysis program something to analyze during these silences. The */sflib/anal/lpc* directory includes a "soundfile" called *subaudionoise* designed solely for this purpose. Open this soundfile in a new window, then select (drag over) a duration that matches the duration of the soundfile to be analyzed. Then, in the window of the soundfile to be analyzed, select *Mix* under the *Edit* menu. The samples from the *subaudionoise* window (which are inaudible) will be mixed in with (added to) the sample values for the source signal, eliminating the zero amplitude "silences." Do not save this altered version of the source soundfile.

Performing LPC spectral and pitch track analysis with MIXVIEWS

(1) Open *mixviews* (by typing *mxv* in a shell window) and then open the soundfile to be analyzed:

• Under *File*, select *Open*. In the selection box that appears, choose or type in the name (and, if necessary, the directory path) of the soundfile to be analyzed.

(2) A *soundfile window* will open with an amplitude display of this soundfile.

In this window, you must select (highlight) the portion of the soundfile to be analyzed.

To select the entire soundfile, click anywhere within the waveform display with the right mouse button. To select only a portion of the complete soundfile, click with the *left* mouse button to select the *beginning* point, and with the *middle* mouse button to select an *end* point. The region of the soundfile to be analyzed now should be highlighted.

Reminder: Take particular care in making this selection. If the soundfile includes even a brief silence at the beginning or end, use the left and middle mouse buttons to delete this silence — but none of the actual sound waveform — from the selection to be analyzed.

(3) In the soundfile window, click on *Analyze*, then on *LPC & Pitch Envelope*.

This will initiate first an *lpc* spectral analysis, then a *pitch* analysis. Before each of these analyses, a "dialog box" will open in which you must set the parameters for the analysis.

(4) Within the *lpc analysis*: dialog box you must set either

☞ a *Frame size* (in samples; default = 220, but higher values often work better))

or a

☞ *Frame rate* (default, in herz, is 0)

If the *Frame rate* is left at zero, the value for the *Frame offset* will be used. If the *Frame rate* is set to any value, this value will be used for the analysis, and the *Frame offset* value will be ignored.

Generally it is easier to deal with the *Frame offset* value. If you are analyzing a 44.1k soundfile, raising the *Frame offset* value to 400 or 500 (in order to create approximately 100 analysis frames for each second of sound), usually will produce a better analysis, and also may avoid a beginning or ending frame of zero amplitude.

After setting these two values click on Confirm to launch the analysis.

If, while performing the analysis, *mixviews* encounters frames with zero amplitude, it will display the error message

Cannot analyze frames with zero amplitude

If this happens, you should abort the analysis, reset the soundfile region to be analyzed (or else mix in some subaudio noise) and try again:

- In the error message box, click on OK
- A *Pitch Track* analysis dialog box will appear. In this box, abort the pitch analysis by clicking on Cancel

(5) After performing the *lpc* spectral analysis, *mxv* will open two windows:

☞ A full screen window will include four graphical displays of the *lpc data*:

- The top display, generally of little interest, graphs the *residual* signal (*Resid. R*) produced by the analysis.
- The second display (*Signal R*), very similar to the display within the *soundfile window*, graphs the root-mean-square amplitude of the soundfile
- The third display shows the *Error* function produced during the *lpc* analysis.

For tones with a well defined pitch, such as an arco violin tone, this value will remain very close to 0 throughout the analysis, and its display may even *appear* to be blank.

- The bottom display, which is blank, is reserved for pitch track data, which has not yet been performed.

☞ Superimposed on the *lpc* data display window will be a dialog box in which you set parameters for a pitch analysis. Generally, it is best to perform a pitch analysis at this time, and return to the *lpc* display later.

(6) In the *Pitch Track Analysis* box:

Even if you are dealing with an aperiodic signal without a well defined pitch, you should perform a pitch analysis. In this box you must set the "pitch track" parameter values for

- *Frame size* (in samples; default = 350 samples ; maximum = 1000)
- *Frame offset* (in samples; default = 200 samples) (Number of frames per second)
- *Frame rate* (default = 0) (As in the *lpc* dialog box, this flag parameter generally can be ignored.)
- *High Freq. Boundary* (highest estimated frequency, in herz; default = 1000)
- *Low Freq. Boundary* (lowest estimated frequency, in herz; default = 100)

With 44.1 k soundfiles, raising the *Frame size* to 500, or to the maximum 1000, and perhaps raising the *Frame offset* value as well, may produce a better analysis.
Adjust the *high* and *low* estimated pitch values to a narrower range.

After setting these values, click on Confirm or tap a carriage return to launch the pitch analysis.

(6) After the pitch analysis has been performed a window will open that displays the pitch data ("*frequency envelope*") and, again, an amplitude envelope of the source soundfile.

If you see abrupt discontinuities in the pitch envelope display (other than during the attack) which you believe may be erroneous (not a true reflection of the actual pitch envelope of the source soundfile) you can smooth out these discontinuities. Select a region for editing, then, under the *Modify* window, select *smooth curve*. You can perform this "smoothing" operation on the selected region several times in succession if necessary.

("Smoothing" of the pitch track data also can be performed, however, after merging this data into the *lpc* analysis.)

Advanced users should note that the pitch data also can be modified in various other ways (for example, reversed, or cross faded with another pitch analysis_) by various operations available under the *Edit* and *Modify* menus.

If the pitch data looks okay, select the region to be used by clicking in the display with the mouse. To select and highlight the entire pitch analysis for inclusion within an *lpc* file, click anywhere within the display with the right mouse button.
Now move this window out of the way, either by *minimizing* the window (by clicking in the small box near the right edge of the titlebar), or, less commonly, by dragging it on its titlebar to a corner of the monitor display.
Although it is possible to save this pitch data to a file, it is more common to discard the data after merging it into an *lpc* analyis file.

(7) Return to the *lpc analysis window*. To merge the currently selected pitch data into this analysis, select

merge pitch data under the *LPC* menu. The data from the pitch track window now should appear in the bottom display (*freq. in*) within the lpc window.
A highly recommended additional step: Make sure that the entire *lpc* display still is highlighted. Then, under the *LPC* menu, select *stabilize frames*. This operation will search for abrupt discontinuities within the *lpc* data, which can result in chirps, bloops, maxamp white noise and other artifacts during resynthesis, and will smooth out these discontinuities through a process of interpolation.

The *LPC* menu also contains a selection called *adjust pitch deviation*, Choosing this option will open another box in which one sets values by which the pitch track data can be smoothed.
(Alternatively, though much less commonly, pitch inflections can be exaggerated by this operation.)

Advanced users: Note than various additional types of modification of the *lpc* data are available under the *Edit* and *Modify* menus.

(8) At this point, it usually is best to save the *lpc* data to a file. The filename should end with the extension *.lpc*. Otherwise, *mixviews* will not be able to reopen this file in the future.
To obtain information on the analysis parameters that created this file, either now or any time in the future, select *file information* under the *File* menu.

9.3.2. LPC Resynthesis

In *lpc* **resynthesis**, two synthetic audio signals
— a pitched pulse train waveform with exactly harmonic partials, and
— white noise
are mixed to form a composite **driver** (source) signal, which then is filtered through the time varying resonances provided by the filter coefficients of an *lpc* analysis file. Pitch data from the analysis file controls the pitch of the pulse train oscillator, and the relative amplitudes of the pitched and noise components are controlled by the (time varying) *error* coefficients within the analysis. If the source sound that was analyzed was an oboe tone, the pitched component (the pulse wave) will be dominant in the resynthesis; if a maraca was analyzed, the driver might consist entirely, or almost entirely, of white noise.

The combination of pulse wave and broad band noise models some acoustic sounds (such as the human voice) much better than other sounds (such as a piano tone). The exact harmonicity of the pulse train also does not match the frequency ratios of an acoustic piano tone, in which harmonics become progressively "sharper"). Thus, vocal tones generaly are better candidates for *lpc resynthesis than piano tones. However, lpc* analyses of piano tones may prove very useful in *lpc* **cross synthesis**. In this resynthesis technique, some other acoustic sound, such as a trumpet or gong tone, is used as a driver (in place of the synthetic pulse train and white noise), and is processed through the resonances and amplitude envelope of the piano tone.

Performing lpc resynthesis with mixviews

After creating an *lpc* analysis file with *mixviews*, it is a good idea to test this analysis immediately by performing *lpc* resynthesis. If the analyis does not produce resynthesis of acceptable quality, you will need to redo it, changing some of the analysis parameters.

To perform *lpc* resynthesis with *mixviews*, follow these steps:

(1) Open a window for the *lpc* analysis file, if it is not already open, and, select the region of this analysis to be used in resynthesis.
To select the entire analysis file, click anywhere within the display with the *right* mouse button.

(2) Under the *File* menu, select *New Type*, then *Sound file*.

(3) In the *Create New Soundfile* box that opens, set the following parameter values for this resynthesis soundfile:

- an output *duration* (The default value of *one second* almost always must be changed.)
  In setting the output *duration* argument, you determine whether or not time warping will be applied during resynthesis. To perform "straight" resynthesis, with no time expansion or

compression, your *duration* argument should match the duration of the *lpc* analysis. If you *do* wish to perform time warping, simply set this argument to the desired output duration.

• the *sampling rate*, which should match the sampling rate of the soundfile that was analyzed

If the sampling rate does not match the sampling rate of the *lpc* file you will be warned but allowed to continue if you wish. The formants and duration of the resulting output soundfile will not be "correct," and, unless you get lucky, you probably will not like the resulting resynthesis.

• a *name* for the new soundfile.

After setting these parameters, click on the Confirm box.

(4) A blank new soundfile window will open. Under the *Sound* menu, select *Synthesis*, and then *LPC resynthesis*. Yet another "dialog" box will open, in which you can set resynthesis arguments for

• *Gain factor* (default = 1.)

If the resulting soundfile is too loud or too soft, the resynthesis job can be redone and this value adjusted.

• *Unvoiced threshold* (default = 1.0)

This argument determines the frame *error* value *above* which the driver for resynthesis will consist entirely of white noise.

• *Voiced threshold* (default = 0.0) This argument determines the frame *error* value *below* which the driver for resynthesis will consist entirely of a pitched pulse train.

Actually, since the *lpc error* data always will be between 0 and 1., and most often below .3, these two default thresholds always will result in a driver that is a mixture of white noise and a pitched pulse train.

• *Voiced/unvoiced amp factor* (default = 3.0)

By default, the amplitude of the pulse train component of the resynthesis driver will be three times the amplitude of the white noise component.

The default values for these four parameters often are a good beginning point. However, to achieve high quality audio resynthesis, or a particular timbral quality, it sometimes will be necessary to make adjustments in these values.

lpc error data above .3 or so is typical of percussive, unpitched sounds. Reducing the *Unvoiced threshold* to a value of .8, or perhaps .5, or even lower, and raising the *Voiced Threshold* value slightly, perhaps to .02 or so, may produce better resynthesis. To increase the noise component within a resynthesis (often important for high frequency resolution and attack noise), decrease the *3.* in the *Voiced/Unvoiced* parameter. To reduce the noise component, increase this value.

After setting these parameters, tap a carriage return or click on *confirm* to begin resynthesis compilation.

(5) After resynthesis has been completed, a waveform display of the resulting sound will appear, and you can play this sound. If you are happy with the results, perform a *Save* operation. If not, and if you wish to redo the resynthesis within this window, select *remove* from the *Edit* menu to delete the current resynthesized sound, and then, depending upon what you wish to change, return either to step 3 or step 4 above.

Performing lpc cross synthesis with MIXVIEWS

To use *mixviews* to filter a soundfile through the formant data from a previously created *lpc* analysis file, follow these steps:

(1) Open a soundfile. In the waveform display, select all or a portion of this soundfile to use as a driver to be filtered through the resonances of the analysis.

(2) Make a copy of this selection. This precaution will assure that you do not inadvertently destroy data in the original soundfile.

Under the *Edit* menu, select *Copy to New*.

A new window will open, with a copy of the waveform display. Perform all further work in this copy window.

(3) Change the format of the soundfile copy from 16 bit integer to floating point.

Applying the filter coefficients from an *lpc* analysis to an integer soundfile often results in very high amplitude values and, as a result, maxamp white noise. To avoid this, the driver selection should be converted to *floating points* format, in which all sample values range between 0 and 1.

• Under the *Sound* menu, select *Change sample format*.

• In the box that opens, the default new format should be set to *floating point*. If not, click on this

button, then on *Confirm*.
(After conversion to floating point format, the soundfile copy still can be played.)

(4) Open a window for an *lpc* analysis file, and select the region (often the entire analysis) to use as a filter source. Then minimize this window.

(5) Filter the floating point driver copy through the formants of the *lpc* file data:
- Under the *Modify* menu, select *Filter*, then *LPC Formant*.
- A box will open in which you can set a *Gain factor* (default = 1.). Then click on *Confirm* to perform the cross synthesis operation.

(6) When the cross-synthesis is completed, play the resulting sound.
If you wish to save this sound, it first must be converted from floating point back to 16 bit integer format. To do this:
- Make certain that the whole new soundfile, or the region you wish to save, is selected and highlighted. Then
- Under the *Sound* menu, select *Change sample format*.
- In the box that opens, click on *16 bit linear* if this button is not already selected, and then click on confirm
- Now perform a save operation on the soundfile in the normal manner.

Performing lpc resynthesis and cross-synthesis with Csound

In order to compile a resynthesis or cross-synthesis soundfile with *Csound*, one must create a suitable orchestra file algorithm and a companion score file. To simplify this process, the Eastman Csound Library includes several instrument algorithms and processing modules (subroutines included within some other instrument) designed to facilitate various types of *lpc* resynthesis operations. These resources include:

resyn — creates resynthesis soundfiles

xsyn and *gxsyn* — create a cross-synthesis soundfiles

lpcpitch, xsynpitch and *gxsynpitch* — read in the pitch track data from an *lpc* analysis file and pass it to some instrument (such as algorithm *samp*)

Manual documentation, along with the customary score templates and example scores for these algorithms, are included within the *Eastman Csound Library* binder available in the studio. Soundfiles compiled from these algorithms and example scores are located, as usual, in the *sflib*x directory. Advanced users may wish to write their own resynthesis and cross-synthesis *Csound* algorithms for particular purposes, perhaps using the generic Library algorithms we have provided as models. The unit generators *lpslot* and *lpinterpol*, introduced in 1996, enable one to interpolate between two or more analysis files.

Creating a link to an analysis File : LPLINK

In addition to preparing orchestra and score files for a *Csound* resynthesis job, however, the user also must create a *link* to each analysis file to be used. In order for us to be able to specify particular *lpc* analysis files in a score parameter, Csound requires that these analysis files be called *lp.#*, where *#* is an integer (*lp.1, lp.3, lp.14*, and so on). Furthermore, the analysis file must be located in the same directory from which you submit a *Csound* job. As previously noted, at Eastman we store analysis files on the *snd* disks, rather than in user directories on the smaller system disks, since the large size of these files could quickly fill up the */u* partition of the system disk. Therefore, it is necessary to create a Unix **link** file called *lp.#* (where *#* is an integer) which "points to" the desired analysis file on your *snd* directory. Example:

lplink myvoice.lpc 2

Result: This command will create a link file called *lp.2* in your current working Unix directory, that points to analysis file *myvoice.lpc* in your *$SFDIR* directory. See the manual page on *lplink* for more details and options, or else type the command *lplink* with no arguments. Be sure to remove these *lp.#* link files as soon as they are no longer needed.