A Prototype Real-time Plugin Framework for the Phase Vocoder

Richard Dobson

Media Technology Research Centre, University of Bath, Bath, BA2 7AY

With the dramatic increase in computing power over the last few years, computationally intensive tasks such as the phase vocoder can now be performed faster than real-time. This paper presents details of the modifications and enhancements to the phase vocoder required to support real-time performance, and describes new implementations both as conventional command-line tools and in the form of plugins. This leads to consideration of phase vocoder analysis data as one of a number of multi-dimensional streamable signal types that now demand direct support by streaming audio frameworks, and discuss the concomitant need for the definition of standardized, portable and streamable file formats for these signals.

Keywords: phase vocoder, analysis, benchmark, real time, file format, signal, plugin, framework

1. Introduction.

The Phase Vocoder has a long and distinguished history (Moore 1990: 227) as a powerful frequency-domain technique for sound transformation. Despite certain known limitations (Roads 1996: 553), it remains very popular, with both programmers and composers, and is available in several freely-available implementations. One of the first such public implementations was developed by Mark Dolson (Dolson 1986), as part of the CARL software distribution, and was later ported to the Atari ST computer as a central component of the Composer's Desktop Project system (Endrich 1997). The phase vocoder is generally recognized as one of the most processor-intensive audio algorithms. A typical analysis-resynthesis process involves in excess of 640 Fast Fourier Transform (FFT) cycles, each of 1024 samples, per second of input. Running on the Atari, without benefit of a numeric coprocessor, and with an 8 MHz clock, each second of source sound took around one hour to process.

Despite these difficulties, the phase vocoder offered sufficient processing power to attract the attention of many developers, such as Trevor Wishart, who developed his first set of transformation programs working on analysis data, while working at IRCAM on VOX-5 (Wishart 1988). This work formed the basis for IRCAM’s own 'Super Vocoder de Phase' (Depalle and Poirot 1991), and in particular for the wide range of transformation programs developed by Wishart for the CDP system (Wishart 1994), on which the plugins described in this paper are based. Another important implementation is that by F.R Moore, described in detail in his influential book (Moore 1990), and the basis for a number of important transformation programs by Paul Koonce, Eric Lyon, Christopher Penrose, and others. The Moore implementation is significant also for its inclusion of resynthesis by oscillator bank as an alternative to the usual FFT overlap-add method.

One of the most important properties of the phase vocoder is that it is a single-pass system, with constant frame rate and size; each input block of, say, 128 samples, generates a corresponding output block of the same size. Aside from the underlying tracking of phase between analysis frames, which is fundamentally no different from the running phase of any oscillator, no inter-frame context is involved, so that the phase vocoder is inherently capable of streaming audio data, with a latency equal to the frame overlap. This contrasts with other important analysis techniques such as the tracking vocoder (Roads 1990, McAulay and Quatieri 1986) and the Deterministic plus Stochastic analysis technique (Serra and Smith 1990). These require multiple passes over the input data, e.g. to track spectral peaks across analysis frames, and can generate multiple parallel data streams. However rapidly a computer may perform the processing, the latency is effectively that of the whole file.

Within the last few years, consumer-level computers have reached levels of processing power that at last enable the real-time capability of the phase vocoder to be realized. It remains important to implement the phase vocoder as efficiently as possible, but this capability raises new issues relating to the design and implementation of streaming audio systems in general. Where a series of transformations is to be applied to a sound, it is clearly inefficient to chain complete phase vocoders in series, where the obvious solution is to apply the transformations themselves in series, inside a single analysis-resynthesis framework. The fact that this frequency-domain processing can now easily be done in real-time suggests that such a framework is no longer speculative, and consideration can be given to practical questions of design, implementation, and, in particular, integration with existing framework paradigms.

2. Implementation of a real-time Phase Vocoder

2.1 Core analysis parameters.

As noted above, the analysis and resynthesis performed by the phase vocoder is based on the use of the FFT, which in the forward transform converts a block of time-domain samples (typically a power-of-two size such as 1024, for optimum speed) into an analysis frame of complex (real, imaginary) data representing the detected frequency components of the signal (Dolson 1986, Jaffe 1987). The inverse transform converts this (possibly modified) frame back into the time domain.

The use of the FFT imposes the well-known trade-off between frequency and temporal resolution. The size N of the sample block determines the lowest frequency F that can be detected, which is also the frequency resolution of the overall process:

$$F = SR / N$$

where SR = audio sample rate.

With SR = 44100 and N = 1024, F is 43Hz, adequate for most general purposes. However the temporal width of the frame is some 23 milliseconds, which is too large to capture rapid attack transients. The solution adopted by the phase vocoder is to overlap the frames; the degree of overlap then becomes, after N, the most important parameter in controlling the phase vocoder. It also has very much the dominant effect on processing time, as it sets the overall analysis rate. A minimum overlap of at least half a frame is required, depending on the window shape applied to the sample data to reduce spectral leakage. In practice, overlap factors between 4 and 8 are common; in the latter case, and with the values given above, the overlap equates to 128 samples, giving an overall latency of 2.9 milliseconds, and a control rate of 344.56 Hz. This is therefore a secondary trade-off - a large N (to achieve fine frequency resolution) will tend to give a large overlap, with a corresponding increased latency and lower control rate. Given that power-of-two sizes for the FFT are preferred, these values do not scale exactly with sampling rate, so that, in general, analysis settings may need to be tuned to each sampling rate.

2.2 Benchmarks.

The original CARL implementation of the phase vocoder employed the Fortran FFT routines from the IEEE DSP package. For the CDP port these routines were converted to C by Trevor Wishart and Keith Henderson; the resulting implementation is referred to here as CARL/CDP. The CARL implementation, unusually, supports FFT sizes other than powers of two - this enables more precise analyses of sources with known pitch. There are many competing FFT implementations around (and 'competing' may be understood literally here); most are limited to power-of-two sizes, and may also rely on features of specific processors to gain performance, which renders them unsuitable for general cross-platform implementations. The best library so far has proved to be FFTW (http://www.fftw.org), which is a highly optimized C library, supporting any FFT size, and, very conveniently, supporting a format that can substitute directly for the CARL/CDP routines:

```
#ifdef USE_FFTW
      rfftwnd_one_real_to_complex(forward_plan,anal,NULL);
#else /* CARL */
      fft_(anal,banal,1,N2,1,-2);
      reals_(anal,banal,N2,-2);
#endif
```

Table 1 lists some comparative timings of three implementations when performing a straight disk-based analysis and resynthesis of a 20-second mono source file, using a conservatively specified PC workstation. Not shown in this table is the timing for the very first run of each program. This was always significantly longer than subsequent runs, which benefitted from file data being already present in memory. For the same reason, the analysis-only behaviour of these programs can be misleadingly slow, because the large size of an analysis file incurs a significant system overhead. The timing for 'pvocex' is significant as it is clearly fast enough to permit processing of a stereo file in real time.

Both the original CARL and the Moore/PVC implementation employ only the primary frame format, where the raw complex arrays from the FFT are converted to amplitude/frequency bins (or amplitude/phase in some implementations). However, this conversion is costly, as it requires the use of square-root and arctan functions, for each analysis bin in each frame. Figure1lists the condensed output from the Visual C++ profiler, running **pvocex**, but writing analysis data to disk, demonstrating not only the disk overhead, but also that while the FFT has conventionally been assumed to incur the greatest processing cost, this is no longer the case.

Figure 1.

```
     Func                     Func+Child       Hit
Time         %       Time       %       Count  Function
1649.508  29.6     1649.508  29.6     3454 _pvoc_putframes
1150.897  20.7     5566.707 100.0        1 _main
1058.694  19.0     1058.694  19.0  1771389 _myatan2
 713.012  12.8      713.012  12.8  1771902 _myhypot
   5.585   0.1      131.840   2.4     3454 _wavReadFloatFrames
   4.841   0.1      756.688  13.6     3454 _rfftwnd_real_to_complex
```

As the profiler does not measure standard library functions directly, the important atan2() and sqrt() calls are wrapped by the functions shown. It is clear that these functions consume a significant proportion of the total CPU load, with the atan2 function being the most expensive (though its use is somewhat input-dependent). The overall cost of these functions increases in direct proportion to window size. A self-contained transformation process (i.e. not part of a framework) that deals with only a few analysis bins may well make significant savings by creating frames of standard complex data, and only converting the bins of interest. Such savings are not so easily available to a framework, which has to assume that all the data is required. The complex format is also required for convolution processes, such as reverberation.

The resynthesis stage has been measured to be some 30 percent faster than the corresponding analysis stage, reflecting the simpler calculations involved (see 2.3.2 below). For disk-based processing, therefore, the use of pre-analysed data offers a significant advantage, capacity permitting. Given that such data can be rendered in real-time, analysis files can be handled in the context of a track-based framework without the need for a pre-synthesis step; indeed, the user need see little or no functional difference from rendering a conventional soundfile, and there is no technical reason why such a framework cannot combine frequency-domain and time-domain tracks within a project.

These are core timings for the self-contained phase vocoder, which is written mostly as inline code, and which relies on short source sounds. As will be shown, the requirements of a plugin framework, and modifications for reliable continuous real-time performance, degrade these figures slightly, fortunately not so much as to give problems in practice.

2.3 A simple plugin framework.

The model developed here is of an initial analysis stage, which feeds analysis frames through one or more plugin transformations, selected and controlled by the composer, and resynthesized at the end. For simplicity, it is assumed that each plugin will over-write the incoming frame. A key principle of this model is that the analysis and resynthesis engines are independent, and may indeed be developed as such. Thus the format of the analysis frame needs to be specified formally. The approach taken here is to define a new streamable file format for analysis data, which can thus also define the format for analysis frames streamed through a framework. A C++ class wrapper has been developed from the original C code, facilitating not only the dynamic insertion of plugins, but also the creation of multi-channel processors and alternative resynthesis engines such as an oscillator bank. The analysis and resynthesis objects are created separately, and in accordance with the principles of object-oriented programming, do not share private data. They need not, in principle, be implemented identically. Where processing speed permits, the use of an oscillator bank for resynthesis may increasingly be preferred, especially where extensive pitch-related transformations are being used (Moore 1990: 247).

Three VST and DirectShow plugins have been implemented for this project, based on a simple C++ abstract base class, and drawing on algorithms by Trevor Wishart (Wishart 1994); they are available for free download, for both PC and Macintosh platforms (Dobson 2001). They demonstrate amplitude-only, frequency-only, and combination transforms, respectively:

pvexag:	exaggerates the spectral envelope (fourfold overlap)
pvtransp:	pitch shift +- one octave (sixfold overlap)
pvaccu:	spectral accumulation, with glissando (fourfold overlap)

The increased overlap for **pvtransp** reflects the need to ensure a reasonably clean pitch shift over half an octave.

2.3.1 The PVOC_EX file format.

Each phase vocoder implementation studied in the course of this project has implemented its own file format, in most cases without regard to portability, and in some cases with little or no format information. This is despite that fact that the data is virtually identical (the main difference being in the amplitude scaling employed). A Csound analysis file generated on an Intel platform cannot be used by Csound running on a big-endian platform such as the Macintosh. The CDP file formats are portable, being based on the WAVE and AIFF formats, but support only mono sounds, and their use is inappropriate now that the 32bit floating-point sample type is fully defined for both formats.

PVOC_EX has been developed as a robust, portable and streamable file format which can be supported by all phase vocoder implementations (Dobson 2000a). It exploits the new WAVE_FORMAT_EXTENSIBLE audio file format introduced by Microsoft (and is thus little-endian), and retains all the rendering information of that format, including the definition of speaker positions. It defines not only the standard amplitude/frequency format, but also the amplitude/phase and complex formats, enabling use in a variety of contexts. Full details of the specification, together with downloadable command-line implementations for the Windows and Linux platforms, have been published on the Internet (Dobson 2000b). PVOC_EX is intended as a replacement for CDP's current file formats, and support for it is also being added to Csound. Prototype programs have been developed to convert the Csound and PVC formats to PVOC_EX. By being based on a streaming audio format, it can also be regarded as the specification for analysis data streamed through a plugin framework, as described below.

2.3.2 Real-time considerations.

A key element of the overlap-add resynthesis technique is the maintenance of a running phase, incrementing as each frame is processed:

```
for(i=0, i0=syn, i1=syn+1; i<= NO2; i++, i0+=2,  i1+=2){
  mag = *i0;
  oldOutPhase[i] += *i1 - ((float) i * F);
  phase = oldOutPhase[ i] * TwoPioverR;
  *i0 = (float)((double)mag * cos((double)phase));
  *i1 = (float)((double)mag * sin((double)phase));
}
```

[CARL: pvoc.c]

This technique assumes the ability of the standard C sin and cos functions to return a correct result for an input phase value outside the nominal range. For the short sounds typically used with disk-based processing, this assumption is reasonable. However, in a prototype implementation using a SHARC dsp chip, without the benefit of double-precision processing, degradation of the audio was easily apparent after as little as ten minutes. Clearly, for stability over long periods, the phase calculation must be refined to keep it within bounds. The obvious, if costly, solution is to apply normalization to the phase value each time:

```
oldOutPhase[ i] += *i1 - ((float) i * F);
oldOutPhase[ i]  =  fmod(oldOutPhase[ i], TWOPI)
```

As, even in the SHARC implementation, a small amount of range error in phase can be tolerated, the otherwise unacceptable extra processing burden incurred by this solution can be mitigated by applying the correction incrementally by analysis bin over successive analysis frames. This entails a change to ensure both that the smallest values are accumulated, and that both corrected and uncorrected values are valid:

```
float angledif, phase;
angledif = TwoPioverR * (*i1  - ((float) i * F));
phase = *(oldOutPhase + i) +angledif;
if(i== bin_index)
        phase = (float) fmod(phase,TWOPI);
*(oldOutPhase + i) = phase;
```

This leads to each bin receiving a correction at the rate:

(overlap * (N/2)+1) / SR   secs.

Thus, at SR = 44100, with N = 1024 and overlap = 128, each bin is corrected every 1.489 seconds, well within the safe range.

With this change, and a with few other minor adjustments to the CARL code, the prototype plugins have been run without audio degradation for over eight hours, with an overall penalty in processing time of a modest 3 percent. This is increased to around ten percent, (referenced to Table 1) firstly by the overhead of the conversion to C++, and secondly by the need to support time-domain sample blocks of arbitrary size, currently requiring a method call (albeit inline) every sample.

The primary host application used for testing the VST plugin implementations was the shareware program AudioMulch, by Ross Bencina (http:/www.audiomulch.com). This provides an explicit percentage report, as plugins are running. The availability of low-load synthesis sources, combined with a static display, enables plugins to be tested with a minimum of overhead. Table 2 lists some representative measurements for monophonic processing at the 44100 sample rate, using the same platform as given in Table 1. It can be seen that for a given FFT size, CPU load is almost exactly proportionate to the overlap size, indicating that the phase vocoder can be used in this way with a high level of predictability.

2.3.3 Future prospects for the phase vocoder.

The examples described above have been realized on what even today can be considered a low-powered workstation, and rely on a sizeable latency, through the use of the overlap-add technique. However, it is most important to note that this is not the only way to implement a phase vocoder, and that this latency is no more than a consequence of the need to reduce computational cost. In a far-reaching article, James Moorer (Moorer 2000) argues that the "sliding" FFT, which operates at the audio sample rate (i.e. single-sample latency) will soon be an entirely practicable means of performing a running frequency transform. We may therefore reasonably predict that the phase vocoder will in time be regarded effectively as a time-domain process, albeit one in which the signal (running at the full audio sample-rate) comprises analysis frames, rather than samples of a waveform.

3. Extending the audio plugin framework - new signal types.

It is therefore this specific property, of being a streamable *signal*, that renders phase vocoder data suitable for incorporation in a general audio plugin framework. Attention then shifts from the question of the implementation of the phase vocoder itself, to that of framework design. Hitherto, such frameworks have (with the notable exception of the 'spectral' data type recently added to Csound (Vercoe 2000)) presumed that the only signal is the standard time-domain waveform (which may be represented also in down-sampled form as a control-rate signal). Since the dimension of time is common to all streamable signals of concern here, it can be factored out, so that we can describe such a signal as one-dimensional. Specifically, it is represented as a stream of samples, each in turn represented by a single digital word (memory location).

The paradigmatic challenge posed by the phase vocoder is that the signal has a dimension related directly to the FFT size N, defining the number of analysis 'bins' in each frame. This will always be an odd number, i.e. N/2 + 1. The lowest-level unit is the single analysis bin, represented firstly as a complex number, and eventually as an amplitude-frequency pair. While, on the face of it, the real and imaginary components of the complex number can be varied independently, in practice this is not the case; for stable signal processing the operations permissible on a complex number are tightly circumscribed (pun intended). So the complex number is not to be regarded as a composite of two independent one-dimensional values, but as a genuine two-dimensional object, associated with an allowable set of transformations. The amplitude-frequency pair is therefore in principle hardly less circumscribed. Emerging from the analysis stage, each analysis bin does not generally represent a single component of the source (indeed for arbitrarily complex sources it typically represents aspects of several components), and so can only be modified independently of other bins in a limited way, if the phase relationships of the source are to be preserved. Thus the analysis frame itself represents a truly multi-dimensional signal. While it may look indistinguishable from a frame of additive synthesis data (where each partial is presumed to allow independent processing), and can even be resynthesized by an oscillator bank, it is not the same.

Equally significant for a plugin framework is the fact, obvious enough, that the dimension of a signal is variable. It is a reasonable constraint that the dimension can only vary when the network is stopped, but otherwise all connected processing nodes need to receive the dimension of the signal as a parameter, either directly from the host, or, more naturally, from the source node. This is the principle underlying the DirectShow filter graph (where 'filter' is Microsoft's generic name for any source, transform or sink node), in which format negotiation is handled bi-directionally by the pins which connect filters to each other. In most cases the primary determinant of format acceptance is the rendering hardware (it is especially annoying when a plugin refuses a format that the hardware can accept), while transform filters can be designed to accept (within reason) any set number of dimensions, at least so long as each dimension is of the same type. This requirement cannot be presumed to be met in all cases, and would need to be defined as part of the framework specification. The SDIF format (Wright, Chaudhary, Freed, Khoury and Wessel 1999), though not strictly speaking a streamable format in the terms considered in this paper, supports the combination of multiple types of possible multi-dimensional signals in one file.

The often criticised complexity of the DirectShow filter graph derives largely from the requirement to support many different media types, including custom formats. The central enabling feature is that the format information (e.g. in a WAVE header), is passed to each filter in turn. A media type descriptor is used to identify which filters are suitable for given media format. The development of WAVE_FORMAT_EXTENSIBLE takes this a stage further by enabling any custom streaming format, including possibly a new custom media type, to be supported by the filter graph.

This multi-dimensional property is not confined to analysis frames. In the Ambisonic spatial encoding technique (Gerzon 1972), sound is represented in a phase-encoded form known as B-Format. This can be understood as representing (in the 'first-order' configuration) the output of three coincident figure-of-eight microphones arranged on the three spatial axes X,Y and Z, together with a fourth omni-directional component W. Thus a first-order B-Format signal comprises a four-channel audio stream, which can be reduced to three by eliminating the height component Z, and the full second-order format demands a nine-channel stream. As with the phase vocoder, the range of reasonable operations that can be applied to this signal is finite; in particular, to preserve the spatial integrity of the whole, the phase relationships between the channels must be preserved. This constraint nevertheless permits a useful range of idiomatic transformations, and most standard signal processing operations, so long as they are applied equally to each channel (Malham and Myatt 1995). A prototype file format for B-Format data has been defined (Dobson 2000c); extensions to this are in development, to provide support for higher orders. It is hoped that the final definition of this format will encourage the wider support for B-Format audio in digital audio workstations and frameworks.

The B-format signal also demonstrates the difference between a truly multi-dimensional signal and a standard multi-channel audio signal. While there may well be a semantic or musical relationship between channels (e.g. each instrument in a multi-track mix), this does not necessarily mean there is also a structural relationship. Hence, such signals can continue to be depicted, in graphic patchworks, by multiple one-dimensional wires. The number of wires tells the user that each channel can in principle be processed independently of the others.

Also in the domain of surround sound and spatialization, it is possible to define two and three dimensional control signals directly analogous to the familiar one-dimensional stereophonic pan control. The problem here is simply that current track-oriented plugin protocols such as VST and DirectShow do not support multi-dimensional parameters, much less enable them to be defined in alternative ways - a 3D co-ordinate can be expressed in both Cartesian (X,Y,Z) form, and in spherical form (radius, azimuth and elevation), each having advantages in particular contexts. Thus, even for the horizontal-only surround applications (which the industry invariably describes as "3D"), the X and Y coordinates have to be represented (e.g. for automation purposes) as separate signals. This can be useful, but makes it much more difficult to apply dynamic modulation of sound trajectories, and the absence of a standardised file format for positional data means that it is typically locked within a particular application. A plugin framework devoted to processing positional data would be a valuable resource for both composition and performance.

4. Conclusions.

Three examples have been given of multi-dimensional signals, demonstrating frequency domain, time domain, and control data, that it is argued the next generation of audio frameworks should seek to support. The computationally most demanding of these, the phase vocoder, has been demonstrated to be practical for real-time use, with low latency, on current platforms, and a prototype plugin framework has been described. For the track-based workstation, support for new signal types ultimately requires the design of one or more new plugin protocols, which cannot be expected to happen quickly. In the short term the practical solution is to define a mini-protocol that can be implemented with the current time-domain systems, as suggested by the examples described. For graphic patchwork systems, the key requirement is to be able to represent each signal by a single wire. Similarly, for acoustic compilers, a new signal data type is required, which can be used as both inputs and outputs by opcodes.

The extension of plugin frameworks to support multi-dimensional signals (at both audio and control rates) clearly represents a higher order of complexity than is currently supported by either track or network based applications. This complexity may yet prove intractable in practice, and the easier solution of a 'framework within a framework' is the more likely short to medium term solution, especially while processes such as real-time spectral transformation are regarded as interesting, but subordinate to the main business of time-domain audio processing. The level of complexity presented to the user is also an important consideration, though a key argument here is that a signal comprising analysis frames can appear to the user as functionally little different from a time-domain signal.

A core principle argued in this paper is that any signal for which a standardized streamable file format can be defined, is a signal which can be reasonably be expected to be supported in a plugin framework. It follows that an essential step in furthering the development of such extended frameworks is the definition of open, robust, portable and streamable file formats for each signal type. The PVOC-EX format has been presented as an example of what is required. These file formats will serve not only to facilitate the transfer of data between applications, but also to define the format of the data itself, together (where the signal is meaningful at audio rates) with all necessary rendering information. Once defined, the way is open for all audio software developers, and all composers and sound designers, to take these new signal formats, and find out what can be done with them.

5. Acknowledgements.

The work described in this paper has been supported by funds from the Arts and Humanities Research Board. I would also like to express my gratitude for the support and encouragement of Archer Endrich, John Fitch, and Trevor Wishart.

6. References.

Depalle, P. and Poirot, G. 1991. SVP: A Modular System for Analysis, Processing and Synthesis of Sound Signals. *Proceedings of the 1991 International Computer Music Conference*. San Francisco: International Computer Music Association.
Dobson, R. 2000a. Development in audio file formats. *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association: 178-181.
Dobson, R. 2000b. *PVOC_EX: File format for phase vocoder data, based on WAVE_FORMAT_EXTENSIBLE*. http://www.bath.ac.uk/~masjpf/NCD/researchdev/pvocex/pvocex.html
Dobson, R. 2000c. *WAVE-FORMAT-EXTENSIBLE and B-Format audio*. http://www.bath.ac.uk/~masjpf/NCD/researchdev/wave-ex/bformat.html
Dolson, M. 1986. The phase vocoder -a tutorial. *Computer Music Journal* **10**(4): 14-27.

Endrich, A. 1997. Composer’s Desktop Project - a musical imperative. *Organised Sound* **2**(1): 29-33.
Gerzon, M. A. 1972. Periphony: with-height sound reproduction. *Journal of the Audio Engineering Society* **21**(1): 2-10.
Jaffe, D.A. 1987. Spectrum analysis tutorial, part 1: the discrete Fourier transform. *Computer Music Journal* **11**(2): 9-24.
Malham, D. and Myatt, A. 1995. 3-d sound spatialization using ambisonic techniques. *Computer Music Journal* **19**(4): 58-70.
McAulay, R.J. and Quatieri, T.F. 1986. Speech analysis/synthesis based on a sinusoidal representation. *IEEE Transactions on Acoustics, Speech, and Signal Processing* **34**(4): 744-754.
Moore, F.R. 1990. *Elements of computer music*. New Jersey: Prentice Hall.
Moorer, J. A. 2000. Audio in the new Millennium. *Journal of the Audio Engineering Society* **48**(5): 490-498.
Roads, C. 1996. *The computer music tutorial.* Cambridge, MA: MIT Press.
Serra, X. and Smith, J.O. 1990. Spectral modelljng synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal* **14**(4): 12-24.
Vercoe, B. 2000. Understanding Csound’s spectral data types. In R. Boulanger (ed.) *The Csound Book*. Cambridge, MA: MIT Press.
Wishart, T. 1988. The composition of Vox-5. *Computer Music Journal* **12**(4): 21-27.
Wishart, T. 1994. *Audible Design*. York: Orpheus the Pantomime.
Wright, M., Chaudhary, A., Freed, A., Khoury,S., and Wessel, D. 1999. Audio applications of the Sound Description Interchange Format. *Proceedings of the 107th Convention*, Audio Engineering Society.

Table 1.

Program	Implementation	Execution time (seconds)
		(average of ten runs)
plainpv	(Moore/PVC)	19.05

pvoc	(CARL/CDP)	12.0
pvocex	(pvoc/FFTW)	8.0

Parameters: SR = 44100, N = 1024, overlap = 128
Platform: Pentium II, 333MHz, 128MB RAM, Windows2000
Compiler: Microsoft Visual C++ 5.0, Service Pack 3.

Table 2. VST plugin performance.

FFT size	Overlap	CPU Load (%)
1024	256	27
1024	160	47
1024	128	56
2048	512	32
2048	256	61
2048	160	97