TRADITIONAL (?) IMPLEMENTATIONS OF A PHASE-VOCODER: THE TRICKS OF THE TRADE

Amalia De Götzen — Università di Padova, Padova, Italy, corvo@dei.unipd.it

Nicola Bernardini — Conservatorio "C.Pollini", Padova, Italy, nicb@axnet.it

Daniel Arfib — LMA-CNRS, Marseille, France, arfib@lma.cnrs-mrs.fr

ABSTRACT

Although the use of the phase-vocoder is not a very recent technique in musical applications and an extensive literature exists on the subject (cf.[1, 2, 3, 4, 5, 6, 7, 8]), hardly any fairly complete reference implementation can be found (cf. for example[9] pp.256-258, where the resynthesis part is performed through an oscillator bank instead of an iFFT).
This paper describes in depth an implementation of a phase-vocoder which was entirely coded in ***Matlab***™ to be added to the ***Cost-G6—DAfx*** web site[1] as the reference source code implementation of all phase-vocoder based effects. The code is licensed in the terms of the *General Public License* (GPL) used in open-source projects and can thus be picked up from the ***Cost-G6—DAfx*** web site and re-used for further research and development.

1 INTRODUCTION

The phase-vocoder is a well-known technique that uses frequency—domain transformations to implement a variety of digital audio effects (e.g. time-stretching, pitch-shifting, spectral image processing, etc.). Since its theory is vastly documented (cf.[2, 4, 5, 8]), we will summarize its functionality very briefly in this paper.

1.1 How it works

A short-time Fourier transform (STFT) is performed on a windowed time-domain real signal to obtain a succession of overlapped spectral frames with minimal side-band effects (analysis stage). The time delay at which every spectral frame is picked up from the signal is called the *hop size*. The time-domain signal may be rebuilt by performing an inverse Fast-Fourier transform on all frames followed by a successive accumulation of all frames (an operation termed *overlap-add*) (resynthesis stage).
Knowing the modulus of every bin is not enough: the phase information is necessary for a perfect recovery of a signal without modification. Furthermore the phase information allows an evaluation of 'instantaneous frequencies' by the measure of phases between two frames, which is needed for introducing effects. Thus, in a traditional phase-vocoder implementation the output of the analysis should be in explicit polar form (moduli and phases) in order to achieve fine-grain tracking of frame by frame frequency changes.

1. (//http://echo.gaps.ssr.upm.es/COSTG6/)

Between the analysis and the resynthesis stage a number of operations may be performed to obtain a multitude of different effects (e.g. hop size modification, reading direction inversion, frame shuffling, etc.) Furthermore, keeping track of phase changes between one frame and the others leads to a finer grain indication of frequency contours in the input signal — an information which has proven to be very useful in yet another category of processing (spectral peak following, etc.)
In summary, here is how a phase vocoder works:

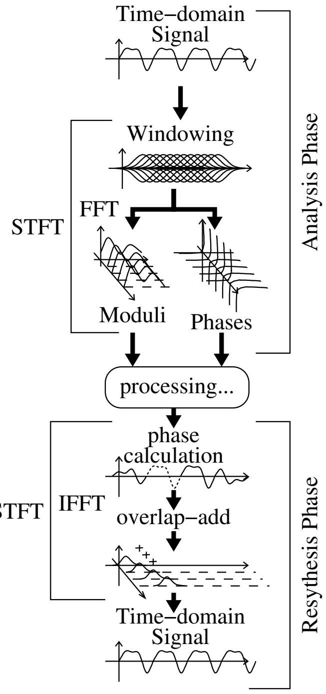

Figure 1. *Phase-Vocoder functionality*

As an example, one of the better known phase-vocoder effects is signal time-stretching without pitch modification. This effect is obtained by modifying the hop size ratio between analysis and synthesis.

2 A REAL-LIFE IMPLEMENTATION

As simple as the theory may seem to be, a good implementation of a phase vocoder contains a significant number of tricks which

often go undocumented as small cooking recipes secrets. The only way of unveil (at least most of) these small tricks of the trade is to deliver (and document thoroughly) a full implementation in open source.
We choose ***Matlab***™ as an implementation language for several reasons, mainly:

a. it is a full-fledged mathematical tool available on any computing platform (and indeed, where the ***Matlab***™ commercial product is not available, its public-domain open-source implementation ***Octave***™ may be used with almost little or no work[2])
b. its vector-based syntax hides most of the unimportant complexities
c. it is one of the languages of choice for the open source effect implementations which are present on the ***Cost-G6—DAfx*** web site

2.1 General implementation idea

The essential idea is to build two ***Matlab***™ functions (`pv_analyze` and `pv_synthesize`) which are intended to work as a tightly coupled set. Between these two function calls, however, any number of manipulations can be performed to obtain the desired effects:

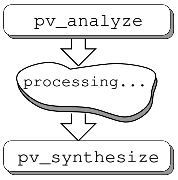

Figure 2. *Implementation scheme*

This implementation allows to build sophisticated time-frequency domain effects with very simple ***Matlab***™ scripts which calls the two above-mentioned functions.

2.2 Implementation basics

Here is what is supposed to happen in our implementation scheme. The code written in the following paragraphs has been somewhat cleaned up for the sake of clarity and length. Things like some error checks, argument processing, heading comments etc. have been removed. However the essential code is there, and the reader may refer to the full-blown implementation available from the ***Cost-G6—DAfx*** web site.
Keeping in mind that software like this, particularly when available in open-source, is under constant upgrading and enhancement, the reader should consider the code included in this paper as documentation, while the actual upgraded code will always be the one available via the Internet.

3 THE ANALYSIS PART

The analysis part boils down to:

```
function [Moduli,Phases]=pv_analyze(X,win,hop)

 [nr, nc] = size (X);
 %
 % compute the window coefficients
 %
 WIN_COEF = hanningz(win);
 %
 % create a matrix Z whose columns contain the
 % windowed time-slices
 %
 num_win =ceil((nr-win+hop)/hop);
 %Z= zeros (win,num_win);

 %
 % now modulate the signal with the window,
 % frame by frame taking due care that the
 % signal is zero padded at the end
 %
 start = 1;
 for i = 0:num_win
  frame_end = win-1;
  if (start+frame_end >= size(X,1))
   frame_end = size(X,1)-start;
  end
  win_end = frame_end+1;
  Z = X(start:start+frame_end) .* \
                  WIN_COEF(1:win_end);
  FZ(1:win_end,i+1) = fft(fftshift(Z));
  start = start + hop;
 end;

 Moduli = abs(FZ);
 Phases  = angle(FZ);
end
```

where:

- `X` is a vector containing a real signal
- `hop` is the analysis hop
- `Moduli` is the returned matrix of moduli
- `Phases` is the returned matrix of phases
- the `for` loop performs the actual continuous framing of the signal (performed at a distance of `hop` frames one from the other); actually, as it will be shown later on (cf. the paragraph on the infinite signal implementation), in real-world implementations analysis, transformation and synthesis can be combined and written in a frame by frame setup

Some minor (but influential) elements may be mentioned even in such a short fragment of code, namely:

- the choice of the proper framing window
- the choice of window and hop size
- fft centering
- correct windowing

3.1 Choice of the framing window

A well-known fact is that framing portions of the signal with rectangular windows introduces unwanted noise and discontinuity in its spectrum. Windowing schemes with better S/N ratio and frequency resolution have been developed to avoid these problems. In general, a window which is smoother on its sides will have a wider principal lobe (that is, a worse frequency resolution), whereas a window with a better frequency resolution will imply higher sides.

2. ***Octave***™ is available from `http://www.che.wisc.edu/octave`

Thus, the choice of the framing window is a tricky one; it often done considering the better (i.e. less influential) spectral response of one type over another. These plots

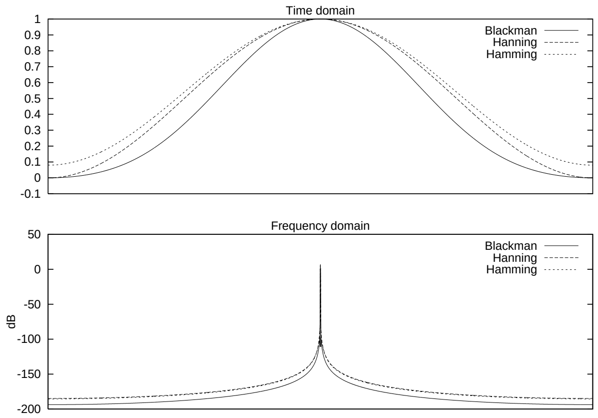

Figure 3. *Window comparison*

show that:

a. the Hamming and Hanning windows provide a spectrum with a thinner primary lobe ($\frac{4}{NT}$, where N is the size of the window and T is the sampling period) than the Blackman window ($\frac{6}{NT}$) (cf.[10])

b. the Blackman window provides a better primary lobe/secondary lobe ratio ($57dB$) than the Hamming ($41dB$) and Hanning ($31dB$) (cf. *ibidem*)

However, in phase vocoding the framing window produces also an overall effect due to the overlap-add procedure during resynthesis. As can be expected, different types of windows behave differently in the overlap-add process; just to show some examples, here is how:

• an overlap-added hanning window behaves with hop/window ratios set to $\frac{3}{4}$ $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, $\frac{1}{5}$ and $\frac{1}{8}$:

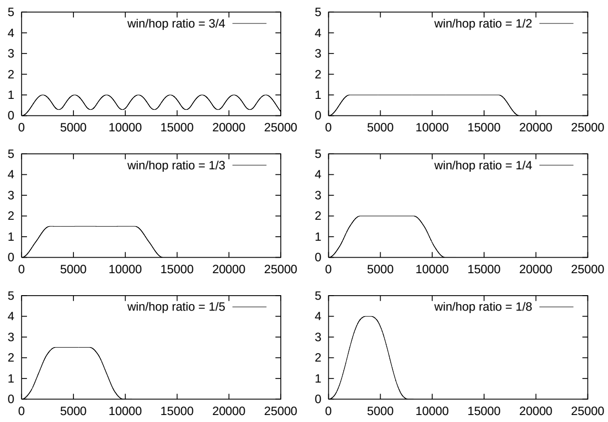

Figure 4. *Hanning overlap adding*

• an overlap-added blackman window behaves with hop/window ratios set to $\frac{3}{4}$, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, $\frac{1}{5}$ and $\frac{1}{8}$:

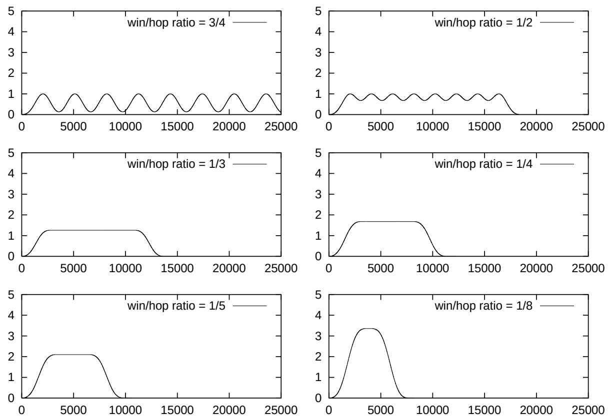

Figure 5. *Blackman overlap adding*

• an overlap-added hamming window behaves with hop/window ratios set to $\frac{3}{4}$, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, $\frac{1}{5}$ and $\frac{1}{8}$:

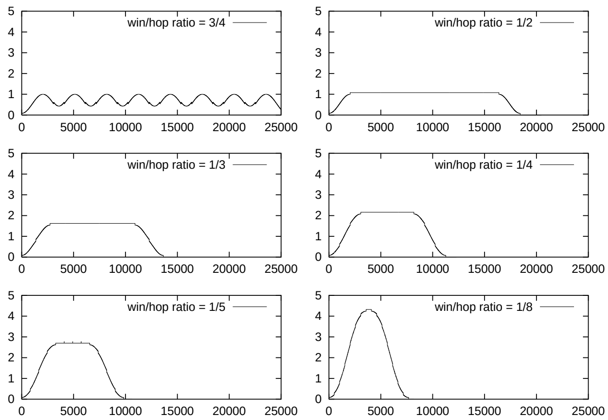

Figure 6. *Hamming overlap adding*

These plots show several interesting properties:

a. the sum of hamming windows presents a discontinuity on both ends; these discontinuities may have a small impact on long signals but can be quite present on short ones;

b. the blackman window modulates the signal when used with a $h_r \geq \frac{1}{2}$ where h_r is the hop/window size ratio

c. the hamming and hanning windows modulate the signal when used with a $h_r > \frac{1}{2}$

d. all windows require (as expected) a variable rescaling factor according to the hop/window ratio; in this respect, the hanning and hamming windows seem to behave more linearly, turning up to something like

$$R = \frac{1}{h_r} \tag{1}$$

where R is the rescaling factor

In the code presented in this paper, we have used the hanning window.

3.2 Choice of window and hop size

A well-known fact is that the choice of the size of the window is extremely important: since the frequency resolution of an FFT depends on

$$F_r = \frac{F_s}{W_s} \tag{2}$$

where:
— F_r is the frequency resolution
— F_s is the sampling rate
— W_s is the window size
it is clear that a bigger window will perform a better FFT. However, in the case of the phase vocoder a bigger FFT is not the only option to be kept in mind. To avoid the loss of substantial signal information it is also important to perform a STFT with many FFTs spreaded over time, that is, to perform a STFT with a small hop size.

3.3 Fft centering

Performing a FFT of a signal is equivalent to the scalar product of the signal and a complex exponential. In an FFT the time vector starts from the left of the window: this means that a pulse in the middle of the window has a phase of $0\ \pi\ 0\ \pi\ 0\ \pi\ \cdots$ This is because the cosine components of that impulse have values $+1\ -1\ +1\ -1\ \cdots$ in the middle of the window. This means that the phase will unwrap in opposite directions for odd and even bins.
In order to have simpler phase relationships, a shift of the signal around its time origin time origin may be performed through the `fftshift` ***Matlab***™ function.

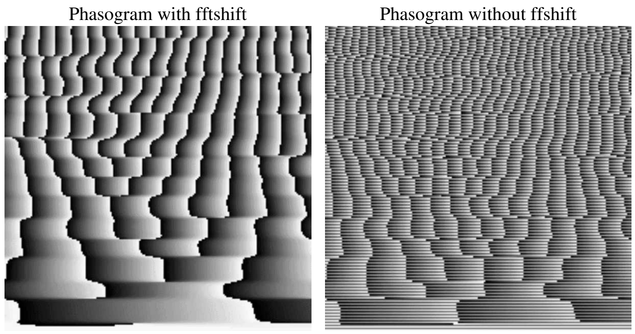

Figure 7. *Phasograms of shifted and non/shifted signals*

The above phasograms[3] of signals analyzed with and without prior `fftshifting` should clarify the simpler phase relationships of the former analysis.

3.4 Correct Windowing

In an STFT, it is important to ensure that the periodicity of the framing window is correct: the periodicity of the framing window should be equal to the declared argument of its function definition. A hanning window, for example, must begin by a zero-valued sample and end by a non-zero valued sample (whose value must be the same as the second sample) - that is, a correct hanning vector must be:

$$hanning(n) = [\ k_0 = 0, k_1, k_2, \ldots, \quad k_{n-1} = k_1\ ]; \tag{3}$$

In ***Matlab***™, picking up the standard hanning window gives an incorrect periodicity, because the boundary samples are non-zero; in ***Octave***™, both boundary samples are zero, which still gives an incorrect periodicity.
This is why we use `hanningz`, a modified version of the hanning window available with the ***Matlab***™ toolboxes:

```
function w = hanningz(n)
w = .5*(1 - cos(2*pi*(0:n-1)'/(n)));
```

4 THE RESYNTHESIS PART

The resynthesis part is slightly more complicated; a simplified code looks like this:

```
function X = pv_synthesize(M, P, win, synt_hop, an_hop)

 [ num_bins, num_frames ] = size(P);
 delta_phi=zeros(num_bins, num_frames-1);
 PF=zeros(num_bins, num_frames);
 window=hanningz(win); % tapering window
 %
 % phase unwrapping
 %
 two_pi=2*pi;
 omega = two_pi*an_hop*[0:num_bins-1]'/num_bins;

 for idx=2 : num_frames
  ddx = idx-1;
  delta_phi(:,ddx) = \
                princarg(P(:,idx)-P(:,ddx)-omega);
  phase_inc(:,ddx)=(omega+delta_phi(:,ddx))/an_hop;
 end

 %
 % now prepare a matrix of complex numbers which
 % recombine modulo and phases to be able to feed
 % the ifft algorithms
 %
 % the idea here is to use the values of the previous
 % phases, and calculate the current phases computing
 % the current phase difference multiplied by the
 % current hop size
 %

 PF(:,1)=P(:,1); % the initial phase is the same
 for idx = 2:num_frames
  ddx = idx-1;
     PF(:,idx)=PF(:,ddx)+synt_hop*phase_inc(:,ddx);
 end;
 Z=M.*exp(i*PF);

 %
 % perform inverse windowing and overlap-adding
 % of the resulting ifft frames
 %

 X = zeros((num_frames*synt_hop)+win, 1);
 curstart = 1;
 for idx = 1:num_frames
  curend = curstart + win - 1;
  RIfft = fftshift(real(ifft(Z(:,idx))));
  X([curstart:curend])= \
           X([curstart:curend]+RIfft.*window;
  curstart = curstart + synt_hop;
 end

 k=sum(hanningz(win) .* window)/synt_hop;
 X=X/k;

end
```

where:
• `synt_hop` is the synthesis hop
• `an_hop` is the analysis hop
In the next sub-paragraphs, we will go over each section.

3. *phasograms* are a representation of phase values with a single-sample analysis hop size

4.1 Phase unwrapping

Phase unwrapping is required to recover the precise phase value for each bin.
The phase values are given modulo 2π. The unwrapping is done by first calculating for all bins the difference `delta_phi` between the actual phase and a target phase which would correspond to the nominal frequency of the bin, and then by calculating the phase increment relative to one sample.

```
two_pi=2*pi;
omega = two_pi*an_hop*[0:num_bins-1]'/num_bins;

for idx=2 : num_frames
 ddx = idx-1;
 delta_phi(:,ddx) = \
                 princarg(P(:,idx)-P(:,ddx)-omega);
 phase_inc(:,ddx)=(omega+delta_phi(:,ddx))/an_hop;
end
```

In this fragment
• `omega` is the nominal phase increment for the analysis hop size for each bin
• `princarg` is a simple function that returns the principal argument of the nominal initial phase of each frame; the `princarg` function is

```
function Phase=princarg(Phasein)
  two_pi=2*pi;
  a=Phasein/two_pi;
  k=round(a);
  Phase=Phasein-k*two_pi;
end
```

(the `princarg` function was written by Carlo Drioli)
• `delta_phi` contains the difference between the phases of two adjacent frames for each bin and its nominal phase
• `phase_inc` is the phase increment for each bin

4.2 Synthesis phase calculation

After finding the correct phase increment, that phase increment may be multiplied the synthesis hop. After that, it may be accumulated frame by frame.
This is what happens in the following code snippet:

```
PF(:,1)=P(:,1); % the initial phase is the same
for idx = 2:num_frames
 ddx = idx-1;
    PF(:,idx)=PF(:,ddx)+synt_hop*phase_inc(:,ddx);
end;
Z=M.*exp(i*PF);
```

After recalculating the correct phase with the synthesis hop, the signal is recombined in a rectangular array of complex numbers. That's done in a single ***Matlab***™ line,
`Z=M .* exp(i*PF);`
Quite a lot goes on in this single line:
• `M` is still the rectangular array of the moduli of each bin in each frame coming out of the analysis stage
• `exp(i*PF)` is the rectangular array of recalculated phases for each bin in each frame (the $e^{i\phi}$ side)
• and `.*` is the member-by-member multiplication

4.3 Overlap-Add

After rebuilding the complex signal and performing iFFTs on all frames, it is necessary to overlap-add each frame every synthesis hop.

```
42  X = zeros((num_frames*synt_hop)+win, 1);
43  curstart = 1;
44  for idx = 1:num_frames
45   curend = curstart + win - 1;
46   RIfft = fftshift(real(ifft(Z(:,idx))));
47   X([curstart:curend]) = \
       X([curstart:curend]) +  RIfft .* window;
48   curstart = curstart + synt_hop;
49  end
```

Here an IFFT is performed on each frame, and each frame is added to the preceding one every `synt_hop` samples.
Another trick is performed right before the overlap-adding (line 47): the reconstructed signal is again multiplied by a (hanning) window, to make sure no phase discontinuities spilled out at the beginning or at the end of a frame (we call this `window tapering`).
After overlap-adding, the current position pointer is upgraded by `synt_hop` samples.

4.4 Signal rescaling

The overlap-add produces (as the name itself says) an addition of portions of the output signal. The size of these portions depends on the selected hop. Supposing that the hop is R samples big, and considering that

$$Y_r(e^{i\omega_k}) \tag{4}$$

is the STFT of a signal calculated every R samples, the resynthesis equation is

$$y(n) = \sum_{r=-\infty}^{r=+\infty}\left[\frac{1}{N}\sum_{k=0}^{N-1} Y_r(e^{i\omega_k})e^{i\omega_k n}\right] = \sum_{r=-\infty}^{r=+\infty} x(n)w(rR-n) = x(n)\sum_{r=-\infty}^{\infty} w(rR-n) \tag{5}$$

However, since

$$\sum_{r=-\infty}^{r=\infty} w(rR-n) = W(\frac{e^{i0}}{R}) \tag{6}$$

then

$$x(n) = \frac{y(n)}{\frac{W(e^{i0})}{R}} \tag{7}$$

that is, the amplitude of the original signal can be restored by dividing the overlap-added signal by the amplitude of the base frequency of the window rescaled by the synthesis hop.
This is exactly what happens in the following code snippet:

```
k=sum(hanningz(win) .* window)/synt_hop;
X=X/k;
```

The `sum(hanningz(win) .* window)` that is, the sum of all samples of the analysis window multiplied by the tapering window is a faster equivalent of performing the fft of the argument and picking up the value of the first bin.

5 MORE TRICKS

Up to now, we have shown a standard implementation. Other additional tricks can be applied to a standard implementation to either simplify it or to obtain better results, among which:
• simplified implementations for integer stretching ratios
• zero-padding the analysis windows
• performing phase vocoding on signals of unlimited length

5.1 Simplified implementations

When all is needed is integer stretching ratios, that is

$$\frac{H_s}{H_a} = [2, 3, \cdots] \tag{8}$$

the implementation can be greatly simplified. The code that follows is a functional time stretcher for integer stretching ratios coded by Daniel Arfib:

```
function MXral(w1,w2,n1,n2)
global DAFx_in;
global DAFx_out;

% w1 and w2 windows (analysis and synthesis)
% lfen is the length of the windows
% n1 and n2: steps (in samples) for
% the analysis and synthesis

ral=n2/n1;
lfen=length(w1)
grain=zeros(lfen,1);
pin=0;pout=0;
pend=length(DAFx_in)-lfen;

while pin<pend
  grain = DAFx_in(pin+1:pin+lfen).* w1;
%======================================
  f = fft(fftshift(grain));
  r = abs(f);
  theta = angle(f);
  ft = (r.* exp(i*ral*theta));
  grain = fftshift(real(ifft(ft))).*w2;
% ====================================
  DAFx_out(pout+1:pout+lfen) = \
      DAFx_out(pout+1:pout+lfen)+grain;
  pin=pin+n1;pout=pout+n2;
end;
```

In this function, the actual core of the processing is performed between the two `%====` lines, and:
• `n1` is the analysis hop size
• `n2` is the synthesis hop size
• `lfen` is the size of the analysis/synthesis windows
• `ral` is the stretching factor
• `w1` is the analysis window
• `w2` is the synthesis (tapering) window
• the signal is read from the global variable `DAFx_in` and written into the global variable `DAFx_out`

The most important line is:

```
ft = (r.* exp(i*ral*theta));
```

where the stretching factor is applied directly to the angle. This can be done here with a considerable simplification because we imposed it to be an integer multiplier.

5.2 Zero-padding frames

In general, large analysis windows provide a better frequency grid, while smaller ones can keep better track of fast spectral modifications.
A nice trick that reduces this problem is to run zero-padded analysis frames; here is how it works:

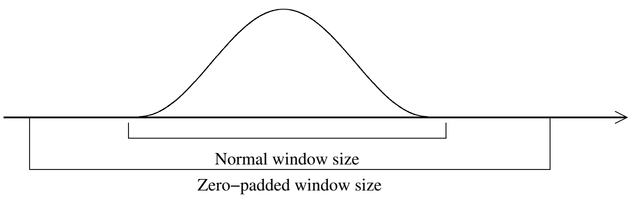

Figure 8. *Zero-padding analysis frames*

In this way, given the same sampling frequency F_c it is possible to obtain substantial frequency precision increases; for example, at $F_c = 44100$, a 1024 samples-sized window has bins that are 43 Hz wide; a 4096 samples-sized window has bins that are 11 Hz wide: thus, by zero-padding a 1024 samples window up to a 4096 samples window it is possible to increase the frequency precision by four.

5.3 Signals of unlimited length

In the implementation shown, all processing is done in core, that is:
• the input signal is loaded entirely into the core memory of the computer
• during analysis and resynthesis, all necessary memory resources (window frames, etc.) are allocated at once
This implementation is simpler and faster to test, but it uses a lot of core memory even for a short sound and it is limited to whatever core memory is available.
Since it is possible to perform phase-vocoding on a limited portion of the signal at a time, the *DAfx* signal framework can be used:

```
while fin(1)~=0
   [x,fin]=readwav(Nx,fin);
   inbuffer=[inbuffer(Nx+1:Ni);x];
   [outbuffer, state]=dafx(inbuffer,state,args...);
   y=outbuffer(1:Ny);
   fout=writewav(y,fout);
end
```

(this framework was suggested in[11] and is an extract provided by Javier Casajus as a unified template for audio effects - the complete template may be found in the ***Cost-G6—DAfx*** website[‡]). In this context, the `dafx` function call may be replaced by a specific effect using a modified version of the `pv_analyze` and `pv_synthesize`. The modification implies saving of the previous buffer state (in the variable `state`) between one buffer and the next.
Here, the signal gets read in chunks (of `Nx` size), processed, and written out in chunks (of `Ny` size); there are no limits on the number of chunks that can be processed: longer input signals will imply longer processing time but not increased resource usage.

6 CONCLUSIONS

When a sophisticated frequency-domain sound processing effect like the phase-vocoder is turned into a real-life implementation, a significant number of tricks have to take place in order to achieve even the most minimal quality required in musical applications. In this paper we document the excerpts of a traditional implementation of a phase-vocoder (if a tradition can be established in such an endeavor), along with most (if not all) the tricks required to run it properly in the chosen implementation setting and language, ***Matlab***™. The complete implementation is available in source from the ***Cost-G6—DAfx*** website[‡].
This implementation was written in the framework of a ***Cost-G6—DAfx*** action working group held at the *Laboratoire de Mécanique et d'Acoustique* of the CNRS in Marseille on May 24-27, 2000. Everybody in that working group contributed to some extent to the making of it.
Moreover, it required a significant amount of help from a number of people we expressly wish to thank here: Emmanuel Favreau, Javier Casajùs, Giovanni De Poli, Carlo Drioli and Riccardo Di Federico.

‡ (//http://echo.gaps.ssr.upm.es/COSTG6/)

7 REFERENCES

[1] Arfib, Daniel, “Analysis, Transformation and Resynthesis of Musical Sounds with the help of Time-Frequency Representation” in *The Representation of Musical Sounds,* Depoli, Giovanni, Piccialli, Aldo, and Roads, Curtis eds., 87-118, MIT Press (1991).

[2] Dolson, Mark, “The Phase-Vocoder: a tutorial,” *Computer Music Journal,* **10,** 4, 14-27, The MIT Press, Cambridge, MA. (1986).

[3] Laroche, Jean and Dolson, Mark, “About this phasiness business” in *Proceedings of the 1997 International Computer Music Conference,* International Computer Music Association, San Francisco USA (1997).

[4] Jean Laroche, “Time and pitch scale modification of audio signals” in *Applications of digital signal processing to audio and acoustics,* Kahrs, Mark and Brandenburg, Karlheinz eds., 279-309, Kluwer Academic Publishers (1998).

[5] J.L.Flanagan and R.M. Golden, “The Phase vocoder,” *The Bell System Technical Journal,* **45,** 8, 1493-1509 (october 1966).

[6] Sussman, Rob and Laroche, Jean, “Application of the phase vocoder to pitch-preserving synchronisation of an audio stream to an external clock” in *Proc. 1999 IEEE Workshop an applications of signal processing to audio and acoustics,* 75-77 (1999).

[7] Laroche, Jean and Dolson, Mark, “New phase-vocoder techniques for pitch shifting, harmonizing and other exotic effects” in *Proc. 1999 IEEE workshop on application of signal processing to audio and acoustic,* 91-94 (1999).

[8] Serra, Marie-Helene, “An Introduction to the Phase Vocoder” in *Musical Signal Processing,* Roads, Curtis, Pope, Stephen, Piccialli, Aldo, and De Poli, Giovanni eds., Swets & Zeitlinger, Lisse, Holland (1997). ISBN: 90265-1482 4.

[9] Moore, Richard, *Elements of Computer Music,* Prentice Hall, Englewood Cliffs (1990).

[10] Cariolaro, Gianfranco, *Teoria unificata dei segnali,* UTET, Torino (1991).

[11] Arfib, Daniel, “Different Ways to Write Digital Audio Effects Programs” in *DAFX98 Proceedings, Barcelona,* 188-191, Barcelona, Spain (November 19-21, 1998).