The Phase-Vocoder and its Realization

Magnus Erik Hvass Pedersen (971055)
Daimi, University of Aarhus, May 2003

1 Introduction

This document describes the mathematical concepts and implementation of the phase-vocoder.

Although the relevant theory is outlined, the reader is assumed to be familiar with both theory and terminology of discrete-time signal processing, algorithms in general, and the computation of sound. The terminology is inspired by the audio-purposes, although the theory is applicable to all sequences satisfying the given conditions.

Additional documentation can be found in the source- and header-files.

2 Motivation

Changing the length of a sequence $x[n], n \in \mathbb{N}$ can be done by computing $y[n] = x[np], p \in \mathbb{R}$.[1] This however has the inherent side-effect of changing the pitch of the output sequence $y[n]$ compared to $x[n]$ when they are played back at the same rate. Conversely, changing the pitch by this method also changes the duration.

A method is therefore desired to alter either one of the pitch or duration without inverse proportionally altering the other. Some of the conceptual uses for such a method are:[2]

Time-stretching:	If $x[n]$ is of length N, then the output of the time-stretching algorithm is $y[n]$ of length $M \neq N$, however with same perceived spectral content or tone.
Pitch-scaling:	This is best understood spectrally; a pitch-scaling of p means the output spectrum is x-axis multiplied by p.

Both of which will be treated in this document, starting with time-stretching.

3 Methods

Generally such methods decompose the input sequence into components that are easier to manipulate than the sequence itself, then modify these simpler components and combine them to form the output sequence. Common solutions are to use either a time- or frequency-domain decomposition - or both.

The method described in this document is of course the phase-vocoder that uses a frequency-domain representation obtained by a derivative of the Fourier transform.

[1] This is a crude form of sample-rate conversion, there are various ways to improve quality.

[2] Historically, the phase-vocoder was developed as a means to reduce bandwidth expenditure in speech transmissions and it was later applied to music production as an effect.

3.1 Granulation

One way of using the time-domain representation is by granulation which models the input as impulses going through a resonance filter. Small pieces of the input are extracted and multiplied by an envelope to form socalled grains. If a higher pitch is desired more grains are output, and likewise fewer grains for a lower pitch. Similarly, grains are played fewer or more times according to the desired time-stretching.

Since the grains are chosen under the assumption they are impulse responses, the method has the added benefit of preserving formants[3] because they formants are already imposed on the impulse responses.

4 DFT

Sampling the Fourier transform at equally spaced intervals of $\frac{2\pi}{N}, N \in \mathbb{N}_+$, we arrive at the following model known as the discrete Fourier transform (DFT):

$$\text{Analysis:} \quad X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{1}$$

$$\text{Synthesis:} \quad x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn} \tag{2}$$

Where $W_N = e^{-j\frac{2\pi}{N}}$ is N-periodic so $W_N^{n+rN} = W_N^n$ for all $r \in \mathbb{Z}$. The phase of $X[k]$ is $\angle X[k]$ and the magnitude is $|X[k]|$, and $X[k]$ is also referred to as the k'th bin. Note also the analysis and synthesis formulae differ only in a factor $\frac{1}{N}$ and the exponential sign of W_N, which can be used in the actual implementation of the DFT.

The direct realization of Eqs. (1) and (2) as an algorithm must perform N complex multiplications and $N-1$ complex additions[4] for each k, and to get all $X[k]$ this must be performed N times, thus yielding an $O(N^2)$ algorithm. For larger N this is too slow for realtime use.

4.1 Identities

When the time-domain sequence $x[n]$ is real-valued, i.e. $x[n] = \Re\{x[n]\}$, only half the frequency-domain representation is needed, the other half is obtained by complex conjugation, first observe:

$$\overline{W_N^{(N-k)}} = e^{j\frac{2\pi}{N}(N-k)} = e^{j2\pi} e^{-j\frac{2\pi}{N}k} = e^{-j\frac{2\pi}{N}k} = W_N^k$$

[3] Formants are characteristics of the spectrum regardless of the pitch or fundamental frequency of the signal or sequence.

[4] Strictly speaking, for some k and n fewer operations will do, however the asymptotic time complexity remains the same.

Using this we obtain:

$$\overline{W_N^{(N-k)n}} = \left(\overline{W_N^{N-k}}\right)^n = W_N^{kn}$$

Which reveals the wanted identity:

$$\overline{X[N-k]} = \overline{\sum_{n=0}^{N-1} x[n] W_N^{(N-k)n}} = \sum_{n=0}^{N-1} x[n] \overline{W_N^{(N-k)n}} = \sum_{n=0}^{N-1} x[n] W_N^{kn} = X[N]$$

Still assuming $x[n]$ is real, then because $W_N^0 = 1$ it again follows from Eq. (1) that $X[0] = \sum_{n=0}^{N-1} x[n]$ and hence $X[0] = \Re\{X[0]\}$. In a similar fashion:

$$W_N^{n\frac{N}{2}} = \left(e^{j\frac{2\pi}{N}\frac{N}{2}}\right)^n = \left(e^{j\pi}\right)^n = (\cos(\pi) + j\sin(\pi))^n = (-1)^n$$

Again this translates to: $X[\frac{N}{2}] = \Re\{X[\frac{N}{2}]\}$. However, since $X[k]$ is only defined for integer k, N must be even to use this.

These identities are used in the implementation of the DFT as described in section 10.

5 FFT

To improve the time-complexity of the DFT algorithm a number of solutions are available, commonly referred to as the fast Fourier transform (FFT). A classic approach is *decimation-in-time*, a divide-and-conquer algorithm splitting $x[n]$ into successively smaller sub-sequences, recursively computing the FFT of those and combining the result to form the output DFT. So apart from the difference in rounding errors imposed by the discrete and bounded nature of digital computers, the output of the direct implementation of the DFT is identical to that of the FFT.

The algorithm can be quite easily understood once a few structural identities of the DFT sequences have been uncovered. In the following N is assumed to be a power of 2, though algorithms exist that do not require this. The sequence $x[n]$ of length N can be split into its even and odd (indexed) parts:

$$\left.\begin{array}{ll} \text{Even:} & x_e[n] = x[2n] \\ \text{Odd:} & x_o[n] = x[2n+1] \end{array}\right\}, n \in \{0, \ldots, N/2 - 1\}$$

Similarly the DFT of these are sequences of length $N/2$ denoted by $X_e[k]$ and $X_o[k]$. Now observe:

$$W_N^2 = e^{-j\frac{2\pi}{N}2} = e^{-j\frac{2\pi}{N/2}} = W_{N/2}$$

Then by splitting $x[n]$ we find a way to split $X[k]$:

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n]W_N^{nk} = \sum_{n=0}^{N/2-1} \left(x[2n]W_N^{2nk} + x[2n+1]W_N^{(2n+1)k} \right) \\ &= \sum_{n=0}^{N/2-1} x_e[n]W_N^{2nk} + W_N^k \sum_{n=0}^{N/2-1} x_o[n]W_N^{2nk} \\ &= \sum_{n=0}^{N/2-1} x_e[n](W_N^2)^{nk} + W_N^k \sum_{n=0}^{N/2-1} x_o[n](W_N^2)^{nk} \\ &= X_e[k] + W_N^k X_o[k] \end{aligned}$$

However to obtain an algorithm with better performance than $O(N^2)$ we still need to utilize each calculation of $X_e[k]$ and $X_o[k]$ for more than one calculation of $X[k]$, the following identity provides this. First note

$$W_N^{k+\frac{N}{2}} = W_N^k W_N^{\frac{N}{2}} = W_N^k e^{-j\frac{2\pi}{N}\frac{N}{2}} = W_N^k e^{-j\pi} = -W_N^k$$

and because of the periodicity of W_N

$$W_{\frac{N}{2}}^{\frac{N}{2}n} = 1$$

using all of this, the identity is found:

$$\begin{aligned} X\left[k+\frac{N}{2}\right] &= X_e\left[k+\frac{N}{2}\right] + W_N^{k+\frac{N}{2}} X_o\left[k+\frac{N}{2}\right] \\ &= \sum_{n=0}^{N/2-1} x_e[n]W_{N/2}^{(k+\frac{N}{2})n} + W_N^{k+\frac{N}{2}} \sum_{n=0}^{N/2-1} x_o[n]W_{N/2}^{(k+\frac{N}{2})n} \\ &= \sum_{n=0}^{N/2-1} x_e[n]W_{N/2}^{kn}W_{\frac{N}{2}N/2}^{\frac{N}{2}n} + W_N^{k+\frac{N}{2}} \sum_{n=0}^{N/2-1} x_o[n]W_{N/2}^{kn}W_{\frac{N}{2}N/2}^{\frac{N}{2}n} \\ &= \sum_{n=0}^{N/2-1} x_e[n]W_{N/2}^{kn} + W_N^{k+\frac{N}{2}} \sum_{n=0}^{N/2-1} x_o[n]W_{N/2}^{kn} \\ &= X_e[k] - W_N^k X_o[k] \end{aligned}$$

Hence $X_e[k]$ and $X_o[k]$ can be used to calculate both $X[k]$ and $X\left[k+\frac{N}{2}\right]$.

5.1 Algorithm

Noting that for $N = 1 : X[0] = x[0]$ we get the following pseudo-code:

```
X[k] FFT(x[n], N)
{
    if (N > 1)
    {
        X_e[k] = FFT(x_e[n], N/2);
        X_o[k] = FFT(x_o[n], N/2);

        for (i = 0; i < N/2; i++)
        {
            X[i] = X_e[k] + W_N^k X_o[k];
            X[i + N/2] = X_e[k] - W_N^k X_o[k];
        }
    }
    else
        X[0] = x[0];
}
```

The two recursive calls that result in the arrays $X_e[k]$ and $X_o[k]$ - both of length $\frac{N}{2}$ - each require $T(\frac{N}{2})$ operations. The loop that generates the output array from those two, requires (a constant factor of) N operations, hence

$$T(N) \approx \begin{cases} 2T(\frac{N}{2}) + N & , N > 1 \\ 1 & , N = 1 \end{cases}$$

By induction it follows that $T(N) \approx N(\log_2(N) + 1)$, i.e. a time-complexity of $O(N \log(N))$.

6 STFT

To improve the DFT's ability to accurately describe what frequencies are in the input sequence at a given time, a derivative of the short-time Fourier transform[5] (STFT) is introduced, first the STFT is defined by:

$$X[n, \lambda) = \sum_{m=-\infty}^{\infty} x[n + m]w[m]e^{-j\lambda m}$$

Where the sequence $w[m]$ is a socalled window of length $L \leq N$ so that $w[m] = 0$ for all $m \notin \{0, \ldots, L - 1\}$. When sampling the STFT at discrete intervals of $\frac{2\pi}{N}$, as was done with the Fourier transform, the following results:

$$X[n, k] = X\left[n, \frac{2\pi}{N}k\right) = \sum_{m=0}^{L-1} x[n + m]w[m]W_N^{km} \tag{3}$$

[5] Also called the time-dependent Fourier transform.

Calculating the N-length FFT for each input sample results in a $O(LN\log(N))$ algorithm, so it is necessary to compute Eq. (3) at larger intervals; while retaining the same asymptotic time complexity, its constant factor becomes much less:

$$X_t[k] = X[tR_a, k] = \sum_{m=0}^{L-1} x[tR_a + m]w[m]W_N^{km}, R_a \leq L, t \in \mathbb{Z} \tag{4}$$

This provides N bins to describe L input samples, but only calculated for every $R_a = \frac{N}{O_a}$ samples of the input sequence, where O_a is referred to as the analysis hop-factor, and R_a as the analysis hop-size.

So $X_t[k]$ is the DFT of the t'th input frame, i.e. the windowed input sequence segment $x_t[m] = x[tR_a + m]w[m], m \in \{0, \ldots, L-1\}$. Therefore many of the discussions below on $X[k]$ are equally valid for $X_t[k]$ as no particular assumptions are made on the input sequence.

There are no requirements for O_a to be an integer, in fact R_a need not be an integer either, so *fractional delays* may be used when *sliding* the window across $x[n]$.

6.1 Leakage

In musical terms one might think of $X[1]$ as denoting the fundamental and $X[k]$ for $k > 1$ are then the harmonics.[6]

In general though, the sequence $x[n]$ contains sinusoids[7] of frequencies that are not multiples of $\frac{2\pi}{N}$ and hence not N-periodic. The resulting effect on the DFT of the windowed segment $x_t[m]$ is known as leakage, because surrounding bins in the spectrum are also affected. It is therefore necessary to find better frequency-estimates rather than using the DFT directly. Section 7 describes just that.

6.2 The Effect of Windowing

It is beyond the scope of this document to give a detailed mathematical explanation of the windowing concept, as it will require an equally detailed explanation of FIR filter design theory also. Let it therefore suffice with a brief outline of the conclusions.

The spectrum of the window is considered by its main- and side-lobes. The width and shape of the main-lobe are important for frequency-resolution, and the side-lobes are important for leakage-reduction.

As the main-lobe gets wider the frequency resolution gets poorer, but as the side-lobes become low quicker relative to the main-lobe, the leakage is reduced. So while it becomes harder to resolve two sinusoids close in frequency, the bins

[6] $X[0]$ is a DC-offset.

[7] The term sinusoid is used throughout for both the real-valued sequence $x_1[n] = A\cos(\omega n + \phi)$ and the complex exponential sequence $x_2[n] = Ae^{j(\omega n+\phi)}$ with $A \in \mathbb{R}$ for both.

farther away are now less affected by non-N-periodic sinusoids, in short, the *smearing* of the main-lobe is increased while it is decreased for the side-lobes.

6.2.1 Hanning Window

This implementation uses the Hanning window:

$$w[m] = \begin{cases} k - (1-k)\cos\left(\frac{2\pi}{L}m\right) & , 0 \leq m \leq L \\ 0 & , \text{else} \end{cases}$$

With $k = 0.5$ for the Hanning window. To make this symmetric[8] over the course of N samples, set $L = N - 1$. Both end-points are then zero $w[0] = w[N-1] = 0$. Either changing k (e.g. to a *Hamming* window) or setting $L = N + 1$ and evaluating $w_1[m] = w[m+1]$ for $m \in \{0, \ldots, N-1\}$ will give non-zero end-points and a symmetrical curve.

6.3 Post-Synthesis Windowing

There are two purposes of post-synthesis windowing. One is to smooth the transition of one frame to the next: The more dramatic changes in phase and magnitude of a bin in successive frames, the more dramatic artifacts arise in the form of *clicks* at the frame-transitions.

Furthermore post-synthesis windowing may be introduced to form the identity function along with pre-analysis windowing and overlap-adding, as described below.

6.4 Overlap-Add

When any non-rectangular windowing (pre-analysis as well as post-synthesis) has taken place, it becomes necessary to overlap successively synthesized frames to reconstruct the sequence.[9] This form of overlapping is known as overlap-add.

Let $y_t[n]$ denote the N-length frames that are to be overlapped. The synthesis hop-size is given by $R_s = \frac{N}{O_s}$, where $O_s \in \mathbb{N}_+$ is the synthesis hop-factor and should satisfy $R_s \in \mathbb{N}_+$.[10] The t'th output frame of the overlap-adding is $y[n]$ of length R_s. The overlap-add formula is then:

$$y[n] = \sum_{\Delta t=0}^{O_s-1} y_{t-\Delta t}[n + \Delta t R_s] \tag{5}$$

Where $y_t[n] = 0, n \notin \{0, \ldots, N-1\}$. Intuitively $y[n]$ can be understood as the summation of the current and previous $O_s - 1$ frames of $y_t[n]$, each frame being offset by another R_s samples.

[8] This implementation uses the periodic window though.

[9] Windows with all samples being non-zero have an inverse (the reciprocal) that can be used in reconstruction. But as post-synthesis windowing is required for this application, overlap is needed anyway.

[10] This implementation will crash otherwise.

For perfect reconstruction, the combined effect of windowing and overlap-adding must result in the identity function.

Since $y[n]$ is only R_s samples long, the overlap-adding must be performed O_s times to produce N samples, where each input frame $y_t[n]$ must be generated according to the principles described below.

6.5 Amplitude Modulation of Windowing

If no overlap-adding is used, the windowing will trivially imply amplitude modulation. Depending on the window type, different amounts O_s of overlap-adding are needed to overcome amplitude modulation. Although not studied further in this document, it is however noted that the Hanning window will exhibit amplitude modulation for $O_s < 3$ when both pre-analysis and post-synthesis windowing is used. If windowing is only performed once, then amplitude modulation will occur for $O_s < 2$.

6.6 Amplitude Compensation

Depending on the type of window and R_s, overlap-adding potentially increases the amplitude of $y[n]$ compared to $x[n]$, which must be compensated. Assuming $y_t[n]$ equals $x_t[n]$ windowed, the resulting scaling corresponding to Eq. (5) is:

$$s[n] = \sum_{\Delta t=0}^{O_s-1} w[n + \Delta t R_s]$$

That is the sequence $s[n]$ is of length R_s. Also assuming no amplitude modulation occurs, $y[n]$ must be $x[n]$ scaled by the average of $s[n]$:

$$\begin{aligned} avg(s[n]) &= \frac{\sum_{n=0}^{R_s-1} s[n]}{R_s} = \frac{\sum_{n=0}^{R_s-1}\sum_{\Delta t=0}^{O_s-1} w[n+\Delta t R_s]}{R_s} \\ &= \frac{\sum_{n=0}^{R_s-1} w[n]+w[n+R_s]+w[n+2R_s]+\cdots+w[n+(O_s-1)R_s]}{R_s} \\ &= \frac{\sum_{n=0}^{N-1} w[n]}{R_s} \end{aligned}$$

Because $x_t[n]$ is itself windowed with $w[n]$ (i.e. pre-analysis windowing), the actual amplitude scaling of $x[n]$ is thus given by:

$$\frac{\sum_{n=0}^{N-1}(w[n])^2}{R_s} \tag{6}$$

7 Phase Unwrapping

As noted above, in general it can not be assumed that a sequence is generated from a sum of sinusoids equally spaced in frequency, or specifically N-periodic. The leakage of the DFT that results from an input sequence with a sinusoid of frequency not matching any of the bins' frequencies, tells us that a bin may

contain information of sinusoids not belonging to the frequency-band the bin is thought to cover.

Therefore it is desirable to reduce the number of bins affected by this leakage and decide the *true* frequency for the bins that are still affected. Using Eq. (4) to transform $x[n]$ the leakage is reduced, while the following provides a way to uncover the true frequency of the tracked sinusoid.

7.1 True Analysis Frequency

The following values are found for each bin.

Let ω_a denote the true analysis frequency. We then wish to find $\omega_a = \omega + \omega_{\Delta a}$ where $\omega = \frac{2\pi}{N}k$ is the frequency associated with the k'th bin, and $\omega_{\Delta a}$ is its deviation in frequency from the true analysis frequency.

Let $\phi_t = \angle X_t[k] \in [-\pi, \pi]$ be the phase at the t'th (current) frame. And similarly ϕ_{t-1} denotes the phase of the previous analysis frame. Let $([\omega]) \in [-\pi, \pi]$ denote ω wrapped to the range $[-\pi, \pi]$ so that $e^{j\omega} = e^{j([\omega])}$, also known as the *principal determination*.

The key to discovering the true frequency is to understand how $X[k]$ is expected to evolve from one frame to the next, how it actually evolved, and deducing from this the deviation of phase and frequency. $X[k]$ can be considered a complex exponential sequence in that this is how it contributes to synthesizing $x[n]$, by determining the phase-offset and magnitude of the k'th partial W_N^k, which follows from the simple fact:

$$X[k]e^{j\Omega} = |X[k]|e^{j\angle X[k]}e^{j\Omega} = |X[k]|e^{j(\Omega + \angle X[k])}$$

7.2 Expected Phase-difference

Assuming the input sequence $x[n]$ is N-periodic, the expected phase difference of $X_t[k]$ from one frame to the next is $\left(\frac{2\pi}{N}k\right) R_a$, i.e. the frequency multiplied by the number of samples the two frames are apart. The phases must then satisfy:

$$\begin{aligned} & e^{j\phi_t} = e^{j(\phi_{t-1} + \left(\frac{2\pi}{N}k\right)R_a)} = e^{j(\phi_{t-1} + \left(\frac{2\pi}{N}k\right)\frac{N}{O_a})} = e^{j(\phi_{t-1} + \frac{2\pi}{O_a}k)} \\ \Updownarrow & \\ & ([\phi_t]) = \left(\left[\phi_{t-1} + \tfrac{2\pi}{O_a}k\right]\right) \end{aligned}$$

Which of course is equivalent to

$$\phi_t = \left(\left[\phi_{t-1} + \frac{2\pi}{O_a}k\right]\right) \tag{7}$$

Note, if $O_a = 1$ (i.e. the hop-size is N, hence there is no overlapping of the analysis frames), then

$$\phi_t = ([\phi_{t-1} + 2\pi k]) = ([\phi_{t-1}]) = \phi_{t-1}$$

7.3 Phase-deviation

But when $x[n]$ is not N-periodic, the phase of a bin in successive frames will differ from that predicted by Eq. (7). This deviation in phase is produced over R_a samples, so the frequency deviation $\omega_{\Delta a}$ must satisfy:

$$e^{j\left(\omega_{\Delta a}R_a+\frac{2\pi}{O_a}k+\phi_{t-1}\right)} = e^{j\phi_t} \tag{8}$$

Intuitively, because the function $e^{j\Omega}$ is 2π-periodic it is not possible to determine from just ϕ_t and ϕ_{t-1} how many cycles (excl. deviation) were actually in the input sequence - it may be any $r \in \mathbb{Z}$ (within bandwidth boundaries), so all that can be known is how much the phase deviated within the range of $[-\pi, \pi]$ and how this translates to a frequency deviation over R_a samples. Eq. (8) becomes:

$$\begin{aligned} & e^{j\left(\omega_{\Delta a}R_a+\frac{2\pi}{O_a}k+\phi_{t-1}\right)} = e^{j\phi_t} \\ \Updownarrow \quad & \\ & e^{j\omega_{\Delta a}R_a} = e^{j(\phi_t-\phi_{t-1}-\frac{2\pi}{O_a}k)} \\ \Updownarrow \quad & \\ & ([\omega_{\Delta a}R_a]) = ([\phi_t - \phi_{t-1} - \tfrac{2\pi}{O_a}k]) \end{aligned}$$

Since $R_a \in \mathbb{R}_+$ is given, $\omega_{\Delta a}$ must satisfy:

$$\begin{aligned} & -\pi \leq \omega_{\Delta a}R_a \leq \pi \\ \Updownarrow \quad & \\ & -\tfrac{\pi}{R_a} \leq \omega_{\Delta a} \leq \tfrac{\pi}{R_a} \end{aligned}$$

Or equivalently $\omega_{\Delta a} \in \left[-\frac{\pi}{R_a}, \frac{\pi}{R_a}\right]$, which of course is satisfied by the following (known as *phase unwrapping*):

$$\omega_{\Delta a} = \frac{([\phi_t - \phi_{t-1} - \frac{2\pi}{O_a}k])}{R_a}$$

To recap, the true analysis frequency is thus given by:

$$\omega_a = \omega + \omega_{\Delta a} = \frac{2\pi}{N}k + \frac{([\phi_t - \phi_{t-1} - \frac{2\pi}{O_a}k])}{R_a} \tag{9}$$

The true analysis bin is the non-discrete bin-number associated with a true frequency: $b_a = \omega_a\frac{N}{2\pi} \in \mathbb{R}$, so that when $\omega_a = \frac{2\pi}{N}k$ then $b_a = k$. Since the deviation $\omega_{\Delta a}$ is limited to $\pm\frac{\pi}{R_a}$ in frequency, it is equivalently limited to $\pm\omega_{\Delta a}\frac{N}{2\pi} = \pm\frac{N}{2R_a} = \pm\frac{O_a}{2}$ bins. If the actual deviation is greater, then this method no longer guarantees correct estimates, it is therefore essential that the leakage is greatly reduced at bins farther away, which is (somewhat) provided by Eq. (4).

7.4 Time Stretching

In [5] a simple way of doing *time-varying* time-stretching is described. By keeping R_s fixed and varying only R_a, the time-stretching factor $\frac{R_s}{R_a}$ can be changed for each frame. The method allows for *sweeping* between regular time-stretching, time-stretching while playing the input backwards ($R_a < 0$), *freezing* the input-frame by not moving the analysis window ($R_a = 0$), etc. For $|R_a| > N$ the analysis will skip samples of $x[n]$, but this was not found to be a problem.

7.5 Mapping ω_a to $\angle Y_t[k]$

In general the true synthesis frequency $\omega_s = \omega_a$ does not match any bin frequency exactly $\nexists k \in \{0, \ldots, N-1\} : \omega_s = \frac{2\pi}{N}k$, i.e. $b_s \notin \{0, \ldots, N-1\}$. So the situation of obtaining $\angle Y_t[k]$ can be considered the inverse of obtaining the true analysis frequency ω_a from $\angle X_t[k]$, i.e. we must model how an analysis $Y_t'[k]$ of $y_t[n]$ would change the analyzed $\angle Y_t'[k]$ over the course of one frame if $y_t[n]$ had a sinusoid of frequency ω_s, so $\angle Y_t[k]$ must be offset with the appropriate phase-deviation from the previous to the current frame.

Now let $\phi_t = \angle Y_t[k] \in [-\pi, \pi]$ be the phase at the t'th (current) synthesis frame. And similarly ϕ_{t-1} denotes the phase of the previous synthesis frame.

The complex exponential sequence $x_1[n]$ of frequency ω_s and phase ϕ_{t-1} will over R_s samples evolve to

$$x_1[R_s] = e^{j(\omega_s R_s + \phi_{t-1})}$$

Therefore we wish to obtain the phase ϕ_t satisfying:

$$\begin{array}{rcl} e^{j\phi_t} & = & x_1[R_s] = e^{j(\omega_s R_s + \phi_{t-1})} \\ \Updownarrow & & \\ ([\phi_t]) & = & ([\omega_s R_s + \phi_{t-1}]) \\ \Updownarrow & & \\ \phi_t & = & ([\omega_s R_s + \phi_{t-1}]) = \left(\left[b_s \frac{2\pi}{N} \frac{N}{O_s} + \phi_{t-1}\right]\right) \\ & = & \left(\left[b_s \frac{2\pi}{O_s} + \phi_{t-1}\right]\right) \end{array}$$

Conceptually one might think of the phase deviation as occuring over R_s samples, hence ϕ_t is only fully accumulated at the first sample of the next frame $t+1$. Adhering to this concept would split the polar representation into $|Y[k]|$ for frame t and $\angle Y[k]$ for frame $t+1$, and would furthermore force all partials for the first output frame to have 0 phase. Although special cases, such as integer time-stretching ratios, can be improved by proper selection of initial phases, it appears to be of less import in general, and the previous phases ϕ_{t-1} are hence initialized to zero, and the polar representation is not split between adjacent frames.

7.6 Algorithm Overview

- Window input to generate the input or analysis frame (N-length) $x_t[m] = x[tR_a + m]w[m]$.
- FFT this to generate $X_t[k]$.
- For each k calculate $Y_t[k]$ by first using Eq. 9 to find b_a for $X_t[k]$, then map this to $\angle Y_t[k]$, and finally set $|Y_t[k]| = |X_t[k]|$.
- Inverse FFT on $Y_t[k]$ to generate $y_t[n]$.
- Window this to generate $w[n]y_t[n]$.
- Overlap-add with previous output to produce the actual output frame $y[n]$ (R_s-length) of the phase-vocoder. Amplitude-compensate using Eq 6.

8 Phase Locking

The algorithm outlined in section 7.6 does not take the structure of $X[k]$ into account. Since leakage is only reduced and not completely removed, sinusoids will still influence their neighbourhood bins. This is used explicitly in the following methods, the general idea being that $Y[k]$ should mimic how $X[k]$ looks in the vicinity of peaks.

8.1 Phase Coherence

The kind of phase coherence of the synthesis phases adressed in section 7 is known as *horizontal* phase coherence since it ensures the phases are kept coherent on a bin-by-bin basis over time. Phase locking deals with the socalled *vertical* phase coherence which is the phase relationship between the bins within a frame.

In general manipulating the phases is risky business, if done slightly wrong the effect will sound *phasey* as two sinusoids of the same frequency - but at skewed phases - are crossfaded (i.e. overlap-added). When done optimally, the partials will sum up to produce the correct non-N-periodic time-stretched or pitch-scaled sinusoids.

8.2 Peaks

A bin $X[k]$ is a peak iff $|X[k-1]| \leq |X[k]| \geq |X[k+1]|$. In the following it is assumed that at least one peak was found. Since it is possible for all bins to have the same magnitude, using strictly less and greater comparisons for determining whether a bin is a peak (which conceptually makes sense), could result in no peaks being found.

Let k_l designate an index associated with index k, meaning that $X[k_l]$ is the peak dominating $X[k]$, and $Y[k_l]$ is then the mapping of this $X[k_l]$. The

dominating peak is taken to be the closest peak, in the case of equal distance to left- and right-hand peaks, the largest one (magnitude-wise) is dominating.

The algorithm for finding peaks calculates a peak-map that maps each index k to k_l: First the peaks are found, then for each bin the distance to the left-hand peak is found, then this is compared to the distance to the right-hand peak and the closest one is chosen.

8.3 Identity Phase Locking

This method directly imposes the relationship between the analysis phases of the source bins $X[k]$ and their peaks $X[k_l]$ to the destination bins $Y[k]$:

$$\angle Y[k] = \angle Y[k_l] + \angle X[k] - \angle X[k_l] \tag{10}$$

Now define the *phasor* $Z[k_l] = e^{j\theta}$, where $\theta = \angle Y[k_l] - \angle X[k_l]$. This yields $Y[k]$ for a peak's region of influence by performing a complex multiplication:

$$Y[k] = Z[k_l]X[k] = e^{j\theta}X[k] = e^{j\theta}|X[k]|e^{j\angle X[k]} = |X[k]|e^{\angle Y[k_l]+\angle X[k]-\angle X[k_l]}$$

8.3.1 Algorithm

The algorithm of section 7.6 has its step mapping $X_t[k]$ to $Y_t[k]$ replaced by the following:

- Generate peak-map for $X_t[k]$.
- For each peak $X_t[k_l]$ calculate its true analysis frequency ω_a, map this to the true synthesis frequency and synthesis phase. Calculate the phasor $Z[k_l] = e^{j\theta}$.
- For each k calculate $Y[k] = Z[k_l]X_t[k]$.

9 Pitch Scaling

As depicted in section 2, altering the length of a sequence also alters its pitch. Time-stretching this new sequence back to the original length results in the desired pitch-scaled sequence. It is possible to interchange the ordering of time-stretching and sample-rate conversion, but the combination of these two are the basis of traditional pitch-scaling with the phase-vocoder.

9.1 Spectrum Scaling

A method is described in [10] for pitch-scaling by mapping of $X[k]$ onto $Y[k]$ (of equal length). The idea is closely related to the conceptual idea of scaling the spectrum in that the contents of bins are simply moved according to the pitch-scaling factor p.

A first attempt could be to calculate the true synthesis frequency $\omega_s = p\omega_a$ for each ω_a associated with the bins $X[k]$. Since only one bin $Y[pb_a]$ is then

used per ω_a or b_a, the necessary leakage is not recreated in $Y[k]$, because it can be expected that there will be gaps in $Y[k]$ of empty bins as several $X[k]$ will map to the same $Y[pb_a]$, or because pb_a is beyond $\{0, \ldots, N-1\}$.

To recreate the appropriate leakage in $Y[k]$ needed to model sinusoids in $y[n]$ of frequencies that are not multiples of $\frac{2\pi}{N}$, the approach is reversed in that for each $Y[k]$ the source-bin[11] $X[\frac{k}{p}]$ is found and its true analysis frequency ω_a is used to find the true synthesis frequency $\omega_s = p\omega_a$ (or equivalently the true synthesis bin number $b_s = pb_a$) that is to be mimicked by $Y[k]$.[12]

This *backward propagation* may cause problems however, since the factor p in $p\omega_a$ will also scale the frequency deviation and for $p > 1$, ω_s may end up exceeding the range of frequencies the synthesis bin $Y[k]$ is capable of mimicking. This is presently ignored though.

For $p < 1$ (decreasing the pitch) the upper part of the spectrum will be empty. A method called *spectral copying* is outlined in [11] for regenerating these higher frequencies from the lower ones.

Since pitch-scaling requires $y[n]$ and $x[n]$ to be of the same length, the analysis and synthesis hop-sizes should be equal: $R_a = R_s$ - although they can differ if time-stretching is to be performed simultaneously.

9.1.1 Algorithm Overview

The algorithm of section 7.6 has its step mapping $X_t[k]$ to $Y_t[k]$ replaced by the following:

- For each k calculate the true analysis bin b_a associated with $X_t[k]$.
- For each k calculate $\angle Y_t[k]$ from the true analysis bin b_a associated with $X_t[\frac{k}{p}]$, scaled to obtain the true synthesis bin: $b_s = pb_a$. Set $|Y_t[k]| = |X_t[\frac{k}{p}]|$.

9.2 Pitch Shifting

Pitch-shifting could be added for an interesting effect. While pitch-scaling conceptually scales the x-axis of the spectrum, pitch-shifting of course shifts it and hence does not preserve the harmonic relations of the spectrum. The source bin would then be: $X[\frac{k}{p} - s]$ with s being the amount of shift, and the true synthesis bin would then be given by: $b_s = pb_a + s$.

9.3 Spectrum Scaling With Identity Phase Locking

Combining spectrum scaling with identity phase locking, leads to a method where the entire region of influence surrounding a peak $X[k_l]$ is *shifted* by $b_\Delta =$

[11] The real-valued index is simply truncated or rounded to an integer. Similar to section 2 there are more sophisticated methods.

[12] *Recreation of leakage* may be considered a simple attempt to provide vertical phase coherence as described in section 8.1.

$k_l(p-1)$, corresponding to scaling the peak only and shifting the bins in the region of influence.[13] Again it is assumed that the shift is an integer $b_\Delta \in \mathbb{Z}$.

This region-shifting is well-defined for $p > 1$, but when $p < 1$ the *compression* of the spectrum will cause overlapping of the mapped regions. The following rules were used instead of the summing suggested in [12].

- If more than one peak maps to a $Y[k']$ for $k' = k + b_\Delta$, then the peak with largest magnitude $|X[k]|$ is chosen.
- At most one $X[k]$ maps to each $Y[k']$ - i.e. no overlapping of region of influence.
- Overlapping regions of influence are divided evenly between the peaks.

The moving of a bin requires an adjustment of the phases, again the entire region may by multiplied by a phasor:

$$Z = e^{jb_\Delta \frac{2\pi}{N} R_s} = e^{jb_\Delta \frac{2\pi}{O_s}}$$

To ensure horizontal phase coherence, this is accumulated for each destination bin: $Z_t[k'] = Z_{t-1}[k']Z$, with $Z_0[k'] = 1$. This means that no phase-unwrapping is required. If $O_s = 2$ then the phasor is simply $Z = e^{jb_\Delta \pi}$, meaning:

$$Z_t[k] = \begin{cases} Z_{t-1}[k] & , b_\Delta \text{ even} \\ -Z_{t-1}[k] & , b_\Delta \text{ odd} \end{cases}$$

Although, $O_s = 2$ will cause amplitude modulation with the chosen window (Hanning), as described in section 6.5.

Since no *backward propagation* is used, there will likely be holes in $Y[k]$ for $p > 1$, perhaps these can be cleverly filled to improve quality further.

9.3.1 Algorithm Overview

The algorithm of section 7.6 has its step mapping $X_t[k]$ to $Y_t[k]$ replaced by the following:

- Generate peak-map for $X_t[k]$.
- Map each peak $X_t[k_l]$ to $Y_t[pk_l]$, resolving conflicts by choosing the one with the greatest magnitude.
- Create phasors for the chosen peaks.
- For each of the mapped peaks in $Y_t[k']$, shift the corresponding region of influence from $X_t[k]$, solving any conflict by selecting the source having the shortest distance to its peak.
- Accumulate the phasor for each $Y_t[k]$ and multiply $Y_t[k]$ with it.

[13] Pitch-shifting of the entire spectrum can then trivially be added by further shifting b_Δ.

10 Implementation

- The programming language used is C++. Implementation is done in the context of the author's own DSP-library named *Yggdrasil*[14].
- The output is an executable file that will process a 16-bit `WAV`-file. Although the conversion to/from internal representation (floating-point), should support platform dependent endian-conversion aswell, it has only been tested on Mac PowerPC.
- The virtual function `DoTick` implements the different phase-vocoding schemes in their respective subclasses. Some of these assume that `anaHop` (i.e. R_a) equals R_s.
- The FFTW implementation of the FFT is used to realize the one-dimensional real-valued FFT. A seperate Codewarrior project has been made that compiles the FFTW as a library. A wrapper-class is then made utilizing this library, for smooth integration with *Yggdrasil* - it does not utilize all of FFTW's features however (its measure and wisdom features in particular).

  FFTW uses the identities found in section 4.1 to minimize the size of the arrays.
- Wrapping of $\angle Y[k]$ is not necessary if the trigonometric implementation takes arguments beyond $[-\pi, \pi]$. It does however keep $\angle Y[k]$ stable in that it will not go into $\pm\infty$.
- The parameters $N = 2048$ and $O_s = 4$ i.e. 75% overlap, and the IPL pitch-scaling phase-vocoder are used by default, but may be changed by altering `mainpvoc.cpp`.
- The output is not clipped/saturated to $[-1, 1]$, though the input of the file-writing is in order to avoid *wrapping distortion* when converting to integer. No dithering is applied.
- There is no explicit error-handling.

10.1 Relevant Source-code Files

- The actual phase-vocoder variations are implemented in a number of C++ classes:
  - `LYggPhaseVocoder.h/cpp`
    Is the base-class and implements the basic phase-vocoder algorithm of section 7.6. It can be found in appendices A and B.

[14] In minimized and modified form to retain only the parts relevant for this project. The copyright notice is void in these files as they are part of a University project.

  - `LYggPhaseVocoderIPL.h/cpp`
    Is the *Identity Phase Locking* algorithm of section 8.3.1. It can be found in appendix C.
  - `LYggPhaseVocoderPitch.h/cpp`
    Is the pitch-scaling algorithm of section 9.1.1. It can be found in appendix D.
  - `LYggPhaseVocoderPitchIPL.h/cpp`
    Is the pitch-scaling algorithm of section 9.3.1. It can be found in appendix E.

- The additional routines and wrappers needed by the phase-vocoder can be found in `LYggRFFT.h/cpp`, `YggWrap.h`, `LYggAudioFileWAV.h/cpp`, etc.

10.2 Installation

- The Codewarrior FFTW project is found in `FFTW/fftw.mcp`, however the library `fftw.lib` is already built. All files of the FFTW distribution are included.
- The Codewarrior phase-vocoder project is found in `pvoc.mcp` and is built to the executable file `pvoc`.

10.3 Optimization

Beyond replacing the DFT with the FFT not much attention was paid to optimization. The inner-loops contain a number of trigonometric function-calls and their combined uses could be replaced with appropriate approximations. Innerloop division with constants are replaced by multiplication of the reciprocal constant as division is much more computationally intensive.

It is possible that some choices of windows and frame-sizes give room for further optimization. Also vector-computation is possible for the FFT and calculation of the properties related to $X[k]$ and $Y[k]$. And the usage of temporary buffers/arrays could also be optimized.

11 Testing

All variants were tested with an excerpt of polyphonic classical music sampled at 44.100 Hz, and various choices of parameters. A number of tests are included on an audio-CD. In general it was found that selecting N too low (e.g. 512) would introduce very audible artifacts for both time-stretching and pitch-scaling. Setting $N = 2048$ would give decent results, albeit still with artifacts. Increasing N further gives an even *softer* sound, but also more *reverberant* and *distant*.

For time-stretching the output of the standard phase-vocoder is consistently lower in amplitude than that of the IPL-version. When comparing the output the levels must be matched to avoid the *louder is better* syndrome. The difference

between the two methods is most audible with large time-stretching-factors in which the IPL version sounds less *jittery* and more focused, but still having *drunk warbling* though.

Regarding pitch-scaling, for larger factors (e.g. $p > 2$) the IPL-version takes the cake. It remains rather smooth compared to the standard version which sounds terribly jittery. For $p < 1$ and also factors close to 1, the difference is hard to assess, but the standard version appears to sound better in that it is slightly *less* jittery. This can be helped somewhat by increasing N though.

11.1 Consistency Measure

Introducing a mathematical measure of sound-quality is interesting not to say brave. First off a mathematical measure of quality seems to be required simply because the quality is actually not good enough, hence the effect of phase-vocoding is also not fully understood.

Therefore one should expect a mathematical measure to have a bias towards the existing solutions to the problem of time-stretching or pitch-scaling, and as such may be rooted in a sub-optimal understanding.

It is further noted in [4] that the time-domain based *PSOLA* algorithm *sounds* better (on speech) but rates worse in their mathematical *consistency measure*.

11.2 Inverse Processing

Another interesting testing method is described in [9] in which two phase-vocoders are put in succession one with the inverse time-stretching factor. The idea is appealing because a time-stretched sound may sound artificial simply because its duration has changed, and the method then seeks to isolate the artifacts introduced by the process of phase-vocoding - of course realizing that the inverse processing might remove these again. The result of their efforts was however still that phase-vocoding suffers from the introduction of artifacts.

12 Improvement

- Perhaps fewer peaks can further improve the phase locking methods, this can be implemented as post-processing of the peak-map in a number of ways, e.g. only allowing one peak for every l bins, selecting the (magnitude-wise) largest.
- Choice of windows could be investigated further.
- Choice of datatype should be investigated in general, i.e. where does `float` suffice and where is `double` needed, etc. The feedback in (some) IIR filters can cause DC-offsets if too low precision is used, perhaps similar errors exist in the accumulation of phases.

- Investigate if $Y[k]$ can be even better generated, perhaps some form of convolution has its place in ensuring vertical phase coherence.
- Investigating other meta-representation of $X[k]$ than just peak-maps of section 8.2. Perhaps some type of analysis could allow for formant-preservation or formant-only modification.
- Stereo-imaging is normally sought preserved in signal processing by linear phase processing, identical alteration of dynamics, and the like. The phase-vocoder works independently for each channel, and hence disregards stereo imaging. Once the general auditory issues have been resolved, stereo imaging should be considered.

A LYggPhaseVocoder.h

```
//.....................................................................................
//
//  Copyright2002-2003 by Thurs. All rights reserved.
//
//  LYggPhaseVocoder
//
//  Base-class for the phase vocoder, implements the standard phase vocoder
//  for time-stretching.
//
//  Magnus EH Pedersen
//
//.....................................................................................

#pragma once

#include "LYggRFFT.h"
#include "LYggHanningWindow.h"
#include "LYggOverlapAdd.h"
#include "YggSamplePt.h"
#include <complex>

namespace Yggdrasil
{

//.....................................................................................
class   LYggPhaseVocoder
{
public:
    // n is the number of bands in the FFT (i.e. n/2 becase x[n] is
    // realvalued), assumed to be even.
    // Os is the number of synthesis overlaps, or synthesis hop-factor
    // (synthesis hopsize is n/Os).
                        LYggPhaseVocoder    (int n, int Os);

    virtual             ~LYggPhaseVocoder   ();

    // input is of length N, output of length Rs = N/Os
    // input is offset by anaHop samples from the previous call.
    // (Fractional delay is possible).
    void                Tick                (YggSamplePt *input,
                                    YggSamplePt *output, float anaHop);

protected:
    // Override this to do mapping from analysis to synthesis bins.
```

```
    virtual void        DoTick              (float anaHop);

protected:
    std::complex<YggSamplePt>   *mFreqDomain;   // X[k], analysis bins.
    YggSamplePt                 *mTimeDomain;   // Temp. buffer for windowing.

    YggSamplePt                 *mPrevSynPhase; // Accumulation of synthesis phase.
    YggSamplePt                 *mPrevAnaPhase; // Previous analysis phase.
    YggSamplePt                 *mTrueBin;      // True analysis bin.
    YggSamplePt                 *mAnaNorm;      // Magnitude/norm of analysis bins.

    YggSamplePt                 mRescale;       // Rescale-factor due to windowing
                                        // and overlap-add.

    const YggSamplePt           kPhaseExpect;   // Expected synthesis phase difference.

    const int                   kRs;            // kN/kOs
    const int                   kOs;            // Synthesis overlap factor.

    const int                   kN;
    const int                   kHalfN;         // kN/2

private:
    LYggRFFT                    mRFFT;
    LYggHanningWindow           mWindow;
    LYggOverlapAdd              mOverlapAdd;
};
} //end namespace Yggdrasil
```

B LYggPhaseVocoder.cpp

```
//_____________________________________________________________________________________
//
//  Copyright2002-2003 by Thurs. All rights reserved.
//
//  LYggPhaseVocoder
//
//  Magnus EH Pedersen
//
//_____________________________________________________________________________________

#include "LYggPhaseVocoder.h"
#include "YggWrap.h"
#include "YggConstants.h"

namespace Yggdrasil
{
//_____________________________________________________________________________________
LYggPhaseVocoder::LYggPhaseVocoder      (int n, int Os) :
mRFFT(n),
mWindow(n),
mOverlapAdd(n, n/Os),
kN(n),
kHalfN(n/2),
kRs(n/Os),
kOs(Os),
kPhaseExpect(kPi2/Os)
{
    int i;

    // Assume all allocations OK ###

    mTimeDomain = new YggSamplePt[kN];
    mFreqDomain = new std::complex<YggSamplePt>[kHalfN];

    mPrevSynPhase = new YggSamplePt[kHalfN];
    mPrevAnaPhase = new YggSamplePt[kHalfN];
    mTrueBin = new YggSamplePt[kHalfN];
    mAnaNorm = new YggSamplePt[kHalfN];

    // Reset previous phases.
    for (i=0; i<kHalfN; i++)
    {
        mPrevAnaPhase[i] = 0;
        mPrevSynPhase[i] = 0;
```

```
    }

    // Calculate rescale-factor due to windowing and overlap-adding.
    for (i=0, mRescale=0; i<kN; i++)
    {
        YggSamplePt myW = mWindow.Lookup(i);
        mRescale += myW*myW;
    }

    mRescale = kRs / mRescale;
}
//_____________________________________________________________________________
LYggPhaseVocoder::~LYggPhaseVocoder     ()
{
    delete [] mTimeDomain;
    delete [] mFreqDomain;

    delete [] mPrevSynPhase;
    delete [] mPrevAnaPhase;
    delete [] mTrueBin;
    delete [] mAnaNorm;
}
//_____________________________________________________________________________
void
LYggPhaseVocoder::Tick                  (YggSamplePt *input, YggSamplePt *output,
                                         float anaHop)
{
    int i;

    // Window input frame.
    for (i=0; i<kN; i++)
        mTimeDomain[i] = input[i] * mWindow.Lookup(i);

    // Transform to get frequency domain representation.
    mRFFT.Transform(mTimeDomain, mFreqDomain);

    // Calculate norm for analysis bins
    for (i=0; i<kHalfN; i++)
        mAnaNorm[i] = std::abs(mFreqDomain[i]);

    // (Virtual) Map from analysis to synthesis bins.
    DoTick(anaHop);

    // Inverse transform to get back time domain representation.
    mRFFT.iTransform(mFreqDomain, mTimeDomain);
```

```
    // Window output of iFFT to smooth transition between frames.
    for (i=0; i<kN; i++)
        mTimeDomain[i] *= mWindow.Lookup(i);

    // Overlap-add and rescale to output.
    mOverlapAdd.Tick(mTimeDomain, output, mRescale);
}
//_____________________________________________________________________________
void
LYggPhaseVocoder::DoTick        (float anaHop)
{
    const YggSamplePt kAnaPhaseExpect = anaHop*(kPi2/kN);
    const YggSamplePt kAnaPhaseExpectR = 1.0/kAnaPhaseExpect;

    for (int i=0; i<kHalfN; i++)
    {
        // Analysis
        YggSamplePt myNorm = mAnaNorm[i];
        YggSamplePt myPhase = std::arg(mFreqDomain[i]);
        YggSamplePt myFreqDeviate = YggWrap(myPhase - mPrevAnaPhase[i]
                                - kAnaPhaseExpect*i, kPi);
        YggSamplePt myBinDeviate = myFreqDeviate * kAnaPhaseExpectR;
        YggSamplePt myTrueBin = i + myBinDeviate;

        mPrevAnaPhase[i] = myPhase;

        // Synthesis
        YggSamplePt mySynPhase = YggWrap(myTrueBin*kPhaseExpect
                                + mPrevSynPhase[i], kPi);

        mPrevSynPhase[i] = mySynPhase;
        mFreqDomain[i] = std::polar<YggSamplePt>(myNorm, mySynPhase);
    }
}
//_____________________________________________________________________________
} //end namespace Yggdrasil
```

C LYggPhaseVocoderIPL.cpp

```
//-----------------------------------------------------------------------------
//
//  Copyright2002-2003 by Thurs. All rights reserved.
//
//  LYggPhaseVocoderIPL
//
//  Magnus EH Pedersen
//
//-----------------------------------------------------------------------------

#include "LYggPhaseVocoderIPL.h"
#include "YggPeakMap.h"
#include "YggWrap.h"
#include "YggConstants.h"

namespace Yggdrasil
{
//-----------------------------------------------------------------------------
LYggPhaseVocoderIPL::LYggPhaseVocoderIPL        (int n, int Os) :
LYggPhaseVocoder(n, Os)
{
    // Assume all allocations OK ###

    mPhasor = new std::complex<YggSamplePt>[kHalfN];
    mPhasorPhase = new YggSamplePt[kHalfN];

    mPeakMap = new int[kHalfN];
}
//-----------------------------------------------------------------------------
LYggPhaseVocoderIPL::~LYggPhaseVocoderIPL       ()
{
    delete [] mPhasor;
    delete [] mPhasorPhase;

    delete [] mPeakMap;
}
//-----------------------------------------------------------------------------
void
LYggPhaseVocoderIPL::DoTick                 (float anaHop)
{
    const YggSamplePt kAnaPhaseExpect = anaHop*(kPi2/kN);
    const YggSamplePt kAnaPhaseExpectR = 1.0/kAnaPhaseExpect;

    int i;
```

```
    // Find local maxima.
    YggPeakMap(mAnaNorm, mPeakMap, kHalfN);

    // Create phasor for the peaks.
    for (i=0; i<kHalfN; i++)
    {
        YggSamplePt myPhase = std::arg(mFreqDomain[i]);

        if (mPeakMap[i]==i)
        {
            // Analysis
            YggSamplePt myFreqDeviate = YggWrap(myPhase - mPrevAnaPhase[i]
                                    - kAnaPhaseExpect*i, kPi);
            YggSamplePt myBinDeviate = myFreqDeviate * kAnaPhaseExpectR;
            YggSamplePt myTrueBin = i + myBinDeviate;

            // Synthesis
            YggSamplePt mySynPhase = YggWrap(myTrueBin*kPhaseExpect
                                    + mPrevSynPhase[i], kPi);

            // Create and store phasor and its phase.
            mPhasor[i] = std::polar<YggSamplePt>(1.0, mySynPhase-myPhase);
            mPhasorPhase[i] = mySynPhase-myPhase;
        }

        mPrevAnaPhase[i] = myPhase;
    }

    // Multiply all bins with the phasor of their peak.
    for (i=0; i<kHalfN; i++)
    {
        YggSamplePt mySynPhase;

        mFreqDomain[i] *= mPhasor[mPeakMap[i]];
        mySynPhase = mPhasorPhase[mPeakMap[i]]+mPrevAnaPhase[i];

        // This is needed for calculating the phasor of the future peaks.
        mPrevSynPhase[i] = mySynPhase;
    }
}
//_____________________________________________________________________________
} //end namespace Yggdrasil
```

D LYggPhaseVocoderPitch.cpp

```
//_____________________________________________________________________________
//
//  Copyright2002-2003 by Thurs. All rights reserved.
//
//  LYggPhaseVocoderPitch
//
//  Magnus EH Pedersen
//
//_____________________________________________________________________________

#include "LYggPhaseVocoderPitch.h"
#include "YggWrap.h"
#include "YggConstants.h"

namespace Yggdrasil
{
//_____________________________________________________________________________
LYggPhaseVocoderPitch::LYggPhaseVocoderPitch        (float pitchScale,
                                            int n, int Os) :
LYggPhaseVocoder(n, Os),
mPitchScale(pitchScale),
mPitchShift(0)
{
    // Assume all allocations OK ###

    mFreqDomain2 = new std::complex<YggSamplePt>[kHalfN];

    mTrueAnaBin = new YggSamplePt[kHalfN];
}
//_____________________________________________________________________________
LYggPhaseVocoderPitch::~LYggPhaseVocoderPitch       ()
{
    delete [] mFreqDomain2;

    delete [] mTrueAnaBin;
}
//_____________________________________________________________________________
void
LYggPhaseVocoderPitch::DoTick                       (float anaHop)
{
    const YggSamplePt kAnaPhaseExpect = anaHop*(kPi2/kN);
    const YggSamplePt kAnaPhaseExpectR = 1.0/kAnaPhaseExpect;

    const YggSamplePt myPitchScale = mPitchScale;
```

```
    const YggSamplePt myPitchScaleReciprocal = 1.0/myPitchScale;

    int i;

    // Calculate true analysis bins
    for (i=0; i<kHalfN; i++)
    {
        YggSamplePt myPhase = std::arg(mFreqDomain[i]);
        YggSamplePt myFreqDeviate = YggWrap(myPhase - mPrevAnaPhase[i]
                                - kAnaPhaseExpect*i, kPi);
        YggSamplePt myBinDeviate = myFreqDeviate * kAnaPhaseExpectR;
        YggSamplePt myTrueBin = i + myBinDeviate;

        mPrevAnaPhase[i] = myPhase;

        mTrueAnaBin[i] = i + myBinDeviate;
    }

    // Calculate synthesis bins
    for (i=0; i<kHalfN; i++)
    {
        // Scale/shift pitch (bin), and round to nearest
        int src = std::round(i*myPitchScaleReciprocal - mPitchShift);

        if ((src>=0) && (src<kHalfN))
        {
            YggSamplePt myNorm = mAnaNorm[src];
            YggSamplePt myTrueSynBin = mTrueAnaBin[src]*myPitchScale
                                    + mPitchShift;
            YggSamplePt mySynPhase = YggWrap(myTrueSynBin*kPhaseExpect
                                    + mPrevSynPhase[i], kPi);

            mFreqDomain2[i] = std::polar<YggSamplePt>(myNorm, mySynPhase);
            mPrevSynPhase[i] = mySynPhase;
        }
        else
        {
            mFreqDomain2[i] = 0;
            mPrevSynPhase[i] = 0;
        }
    }

    std::swap(mFreqDomain, mFreqDomain2);
}
//_____________________________________________________________________________
} //end namespace Yggdrasil
```

E LYggPhaseVocoderPitchIPL.cpp

```
//_____________________________________________________________________________
//
//  Copyright2002-2003 by Thurs. All rights reserved.
//
//  LYggPhaseVocoderPitchIPL
//
//  Magnus EH Pedersen
//
//_____________________________________________________________________________

#include "LYggPhaseVocoderPitchIPL.h"
#include "YggPeakMap.h"
#include "YggConstants.h"

namespace Yggdrasil
{
//_____________________________________________________________________________
LYggPhaseVocoderPitchIPL::LYggPhaseVocoderPitchIPL      (float pitchScale,
                                                         int n, int Os) :
LYggPhaseVocoder(n, Os),
mPitchScale(pitchScale)
{
    // ### Assume all allocations OK.

    mFreqDomain2 = new std::complex<YggSamplePt>[kHalfN];

    mPhasorAccum = new std::complex<YggSamplePt>[kHalfN];
    mPhasor = new std::complex<YggSamplePt>[kHalfN];

    mPeakMap = new int[kHalfN];

    mSrcMap = new int[kHalfN];
    mDistMap = new int[kHalfN];

    for (int i=0; i<kHalfN; i++)
    {
        mPhasorAccum[i] = std::polar<YggSamplePt>(1, 0);
    }
}
//_____________________________________________________________________________
LYggPhaseVocoderPitchIPL::~LYggPhaseVocoderPitchIPL     ()
{
    delete [] mFreqDomain2;
```

```
    delete [] mPhasorAccum;
    delete [] mPhasor;

    delete [] mPeakMap;

    delete [] mSrcMap;
    delete [] mDistMap;
}
//_____________________________________________________________________________
void
LYggPhaseVocoderPitchIPL::DoTick                        (float anaHop)
{
    const YggSamplePt myPitchScale = mPitchScale;

    int i;

    for (i=0; i<kHalfN; i++)
    {
        mSrcMap[i] = -1;
        mDistMap[i] = kN;
    }

    // Find local maxima.
    YggPeakMap(mAnaNorm, mPeakMap, kHalfN);

    // Map peaks to source-map
    for (i=0; i<kHalfN; i++)
    {
        if (mPeakMap[i]==i)
        {
            int shift = std::round(i*(myPitchScale-1));
            int dest = i+shift;

            if (dest>=0 && dest<kHalfN &&
                (mSrcMap[dest]==-1 || mAnaNorm[mSrcMap[dest]]<mAnaNorm[i]))
                mSrcMap[dest] = i;
        }
    }

    // Create phasors
    for (i=0; i<kHalfN; i++)
    {
        int curSrc = mSrcMap[i];

        if (curSrc != -1)
        {
```

```
            int shift = i-curSrc;
            YggSamplePt phaseInc = kPhaseExpect*shift;

            mPhasor[curSrc] = std::polar<YggSamplePt>(1, phaseInc);
        }
    }

    int lastPeakIndex;

    // Move left->right and shift peak's region of influence
    for (i=0, lastPeakIndex=-1; i<kHalfN; i++)
    {
        int curSrc = mSrcMap[i];

        if (curSrc != -1 && mPeakMap[curSrc] == curSrc)
        {
            lastPeakIndex = i;
        }
        else if (lastPeakIndex != -1)
        {
            int peakSrc = mSrcMap[lastPeakIndex];
            int dist = i-lastPeakIndex;
            int shiftSrc = peakSrc+dist;

            if (shiftSrc>=0 && shiftSrc<kHalfN &&
                mPeakMap[shiftSrc] == peakSrc && curSrc == -1)
            // Dist-checking not necessary first time around.
            {
                mSrcMap[i] = shiftSrc;
                mDistMap[i] = dist;
            }
        }
    }

    // Move right->left and shift peak's region of influence, resolving conflicts.
    for (i=kHalfN-1, lastPeakIndex=-1; i>=0; i--)
    {
        int curSrc = mSrcMap[i];

        if (curSrc != -1 && mPeakMap[curSrc] == curSrc)
        {
            lastPeakIndex = i;
        }
        else if (lastPeakIndex != -1)
        {
            int peakSrc = mSrcMap[lastPeakIndex];
```

```
            int dist = lastPeakIndex-i;
            int shiftSrc = peakSrc-dist;

            if (shiftSrc>=0 && shiftSrc<kHalfN && mPeakMap[shiftSrc] == peakSrc
                && (curSrc == -1 || mDistMap[i]>dist))
            {
                mSrcMap[i] = shiftSrc;
                mDistMap[i] = dist;
            }
        }
    }

    // Map from X[source-map] to Y[k]
    for (i=0; i<kHalfN; i++)
    {
        int src=mSrcMap[i];

        if (src!=-1)
        {
            mPhasorAccum[i] *= mPhasor[mPeakMap[src]];

            mFreqDomain2[i] = mFreqDomain[src] * mPhasorAccum[i];
        }
        else
        {
            mFreqDomain2[i] = 0;
            mPhasorAccum[i] = std::polar<YggSamplePt>(1, 0);
        }
    }

    std::swap(mFreqDomain, mFreqDomain2);
}
//_____________________________________________________________________________________________
} //end namespace Yggdrasil
```

References

[1] Discrete-Time Signal Processing
Oppenheim, Schafer, Buck
Prentice Hall
ISBN 0-13-083443-2

[2] Course Notes For Algorithms
Sven Skyum
Daimi, University of Aarhus
Spring 2000

[3] Programmeringsteori og Datastrukturer
E. Schmidt, M. Schwartzbach
Daimi FN-56, University of Aarhus
January 1997

[4] Improved Phase Vocoder Time-Scale Modification of Audio
Jean Laroche, and Mark Dolson
IEEE Transactions on Speech and Audio Processing, Vol. 7, no. 3
May 1999

[5] Time-Stretching with a Time Dependent Stretch Factor
Peter Moller-Nielsen, Steffen Brandorf
Daimi, University of Aarhus
Rev. 22.10.02

[6] Sound manipulation in the Frequency Domain
Nielsen, Brandorff
Computer Science, Information and Media Studies, University of Aarhus
rev. 28.02.02

[7] Traditional (?) Implementations of a Phase-Vocoder
Daniel Arfib, et. al.
Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)
December, 2000

[8] A wavelet based method for audio-video synchronization in broadcasting
Pallone, et. al.
Laboratoire de Mecanique et dAcoustique, CNRS
Dec. 1999

[9] Design and Implementation of a Phase Vocoder With Many Channels.
Rune Orsval

University of Aarhus
8th Jan. 2001

[10] Pitch Scaling Using The Fourier Transform
Stephan M. Sprenger
http://www.dspdimension.com/
August 29th 2000 version.

[11] Non-parametric techniques for pitch-scale and time-scale modification of speech
Eric Moulines, Jean Laroche
Elsevier, Speech Communication 16 (1995) pp. 175-205

[12] New Phase-vocoder Techniques For Pitch-shifting, Harmonizing And Other Exotic Effects
Jean Laroche, Mark Dolson
Proc. 1999 IEEE Workshop on Applications of Signal Processing to Audio And Acoustics
New Paltz, New York, Oct. 17-20, 1999

[13] FFTW
Matteo Frigo, Stevenj G. Johnson
http://www.fftw.org/
Version 2.1.3, 7 November 1999