# The Phase Vocoder

The term ‘phase vocoder’ is used to describe a group of sound analysis-synthesis techniques where the processing of the signal is performed in the frequency domain. Most audio effects, including delays, filters, compression and distortion, are implemented directly on the incoming signal in the time domain. By contrast, phase vocoder effects use frequency and phase information calculated from Fourier transforms to implement a variety of audio effects, including time stretching, pitch shifting, robotization and whisperization.

The basic phase vocoder operation involves segmenting the incoming signal into discrete blocks, converting each block to the frequency domain, performing amplitude and phase modification of specific frequency components, and finally converting each block back to the time domain to obtain the final output [1-2]. There is now a widely recognized ‘standard’ implementation [3-4], which is well documented, and for which the term ‘phase vocoder’ is most commonly used. The specific effect that is produced depends on the type of processing done on the frequency domain signal.

A detailed introduction to Fourier transform theory can be found in several excellent texts [5-6], and this chapter cannot substitute for a complete digital signal processing course. Instead, it will provide an overview of how Fourier transforms are used to create the overall phase vocoder structure, with a focus on practical implementation strategies and specific audio effects which make use of the phase vocoder.

**Auto-Tune**

From 1976 through 1989, Dr. Andy Hildebrand worked for the oil industry, interpreting seismic data. By sending sound waves into the ground, he could detect the reflections, and map potential drill sites. Dr. Hildebrand studied music composition at Rice University, and then developed audio processing tools based on his knowledge in seismic data analysis. He was a leading developer of a variety of plug-ins, including MDT (Multiband Dynamics Tool), JVP (Jupiter Voice Processor) and SST (Spectral Shaping Tool). At a dinner party, a guest challenged him to invent a tool that would help her sing in tune. Based on the phase vocoder, Hildebrand's Antares Audio Technologies released Auto-Tune in late 1996.

Auto-Tune was intended to correct or disguise off-key vocals. It moves the pitch of a note to the nearest true semitone (the nearest musical interval in traditional, equal temperament Western tonal music), thus allowing the vocal parts to be tuned. The original Auto-Tune had a speed parameter which could be set between 0 and 400 milliseconds, and determined how quickly the note moved to the target pitch. Engineers soon realised that by setting this 'attack time' very short, Auto-Tune could be used as an effect to distort vocals, and make it sound as if the voice leaps from note to note in discrete steps. It gives it an artificial, synthesiser like sound, that can be appealing or irritating depending on taste. This unusual effect was the trademark sound of Cher's 1998 hit song, 'Believe.'

Like many audio effects, engineers and performers found a creative use, quite different from the intended use. As Hildebrand said, "I never figured anyone in their right mind would want to do that." Yet Auto-Tune and competing pitch correction technologies are now widely applied (in amateur and professional recordings, and across many genres) for both intended and unusual, artistic uses.

## Phase Vocoder Theory

### Overview

The phase vocoder is based on the short-time Fourier transform (STFT). What sets the STFT apart from other Fourier transform techniques is that on it deliberately operates only a small time segment of the input signal, giving a snapshot of the frequency content of the signal at a particular moment in time. By dividing the signal into a series of discrete frames of samples and performing the STFT on each frame, we get a picture of how the frequency content of the signal evolves over time. By modifying the frequency content in each frame, many new effects are possible which are not easily implemented in the time domain.

shows a diagram of the phase vocoder process. The following steps are common to every phase vocoder effect. Each step is discussed in detail in a subsequent section.

1. Given an input signal of arbitrary length, choose a frame of *N* consecutive samples. The value *N* is known as the frame size or window size.
2. Multiply the signal a window function of length *N*. The window function is defined to have a nonzero value for *N* consecutive samples and a value of zero everywhere else. By multiplying the input signal and the window function, only the *N* samples in the frame will remain; the rest will be set to zero.
3. After the signal has been windowed, apply a fast Fourier transform (FFT). Since the windowed signal contains *N* points, an FFT of size *M* ≥ *N* must be used. Typically the window size and FFT size are identical, however in some cases shorter windows can be used with larger FFTs. In practice, the FFT size *M* is almost always a power of 2 for reasons of computational efficiency. The combination of window function and FFT constitutes the short-time Fourier transform.
4. The output of the FFT will be a collection of *M* frequency-domain bins containing magnitude and phase information for each frequency. Each phase vocoder effect will apply a different type of processing in this step, as discussed below in the Phase Vocoder Effects section.
5. Apply the inverse fast Fourier transform (IFFT) to the output of step 4. This will once again produce *M* samples in the time domain.
6. Add the *M* samples from step 5 to the output buffer which holds the output signal from the effect.
7. Move on to the next frame: move forward *H* samples in the input signal and return to step 1. The output in step 6 will also move forward *H* samples. The value H is called the hop size. The hop size is sometimes equal to the window size but is frequently a fraction of it (e.g. H = N/2 or H = N/4).

This process is known as overlap-add; it works by analyzing a set of overlapping frames within the input signal, and the output of each frame after processing, properly aligned in time, is added to form the output signal. The following sections discuss each aspect of the overlap-add process.

### Windowing

It is important to note that the phase vocoder does not analyze the frequency content of the entire signal at once. Instead, the STFT analyzes only the frequency content present in a particular frame located at a specific time within the signal. A window function of length *N* will contain *N* nonzero samples starting from n = 0, with a value of 0 at all other samples. The simplest type is the rectangular window:



Many types of windows are possible. Other common variants are the Bartlett (triangular) window, the Hann window and the Hamming window ():



Intuitively, the motivation for choosing different types of windows relates to smoothing the edges of the windowed signal. As shows, the rectangular window cuts off abruptly at the start and end, and these discontinuities will create undesirable effects known as sidelobes in the frequency domain, where energy at one frequency will bleed into other frequency bins. The triangular, Hann and Hamming windows produce more gradual transitions at the edges which reduce the sidelobe amplitude. However, they do so at the cost of reducing the precision with which any given frequency component can be identified. In signal processing, the trade-off is described as main lobe width versus sidelobe height; a more complete discussion can be found in [7]. Though there is no single best solution for all cases, in phase vocoder audio effects Hann and Hamming windows are frequently used.

To apply a window function, the input signal is multiplied by the window. The window function can be offset in time to isolate different regions of the input signal:

,

where m is the offset of the window. The samples from m to m+N-1 will be isolated by the window function, with all other samples set to zero.

### Analysis: Fast Fourier Transform

The fast Fourier transform (FFT) is an algorithm for calculating the discrete Fourier transform (DFT) of a signal. In turn, the DFT takes *M* time-domain points as input and produces *M* regularly-spaced frequency domain bins as output. Converting from the time domain to the frequency domain in this way is known as the analysis step of the phase vocoder (in contrast to the synthesis step going from frequency to time domain). It is important that the size *M* of the FFT be at least as large as the length of the input signal. Since real-world audio signals can be of arbitrary length, this illustrates the importance of the window function in isolating only a small number of points for the FFT.

We can represent the combination of window and FFT (together, the short-time Fourier transform) as follows:



where n, k represent the short time frame start and frequency bins respectively. Here k ranges from 0 to M - 1, representing evenly-spaced frequency bins between 0 and 2π.

### Interpreting Frequency-Domain Data

Audio signals are always real numbers, but the values of the frequency bins are complex, containing a real and an imaginary component, *x+jy*. Both components are crucial to making sense of the frequency data, but their meaning is easier to understand in polar (or magnitude-phase) representation, *Aej*. We can convert the polar representation and the cartesian real-imaginary form,



where atan2 is a version of the arctangent function, available in most programming languages, which produces values in the range (-π, π] depending on the quadrant of x and y. So, the time frequency bins in Eq. can be written as:



One important use of phase information is to improve the resolution of frequency detection. Recall that the FFT produces only a fixed number of frequency bins, but signals may contain frequencies that fall between the bin frequencies. The following section shows how comparing phase from two consecutive frames can be used to recover exact frequencies between the bins.

Target Phase, Phase Deviation and Instantaneous Frequency

For frequency bin *k*, a ‘target phase’, *t*, can be calculated using the bin frequency, and unwrapped bin phase of the previous hop. This target phase is the sum of the previous unwrapped phase and the expected phase increment. The expected increment is the frequency of the sinusoid multiplied by the hop size. So we have,



where *k* is the frequency of bin *k*, and *h* is the hop size. This target phase represents the perfect case of a steady state sinusoid fitting exactly into a frequency bin, i.e., no spectral leakage. Since this is almost never the case, we have some phase deviation *d*, as shown in Figure 0.4 and given by:

.

The corresponding value in the range (-:], known as the principal argument, of this deviation phase is then used to calculate the unwrapped phase increment per hop, *h*:



The instantaneous frequency, *fi*, can then be calculated:



where fs is the sample frequency. This instantaneous frequency can then be viewed as a more accurate frequency measurement than the frequency resolution offered by the filterbank or STFT.

### Synthesis: Inverse Fast Fourier Transform

The specific processing done in the frequency domain depends on the particular phase vocoder effect. Once this processing is complete, however, all phase vocoder effects will convert the signal back to the time domain using the inverse fast Fourier transform (IFFT):



This process is known as synthesis or reconstruction. It produces *M* points in the time domain which can be added to the output buffer. Depending on the type of frequency-domain processing performed, it is possible that the output frame may not have smooth edges even if a Hamming window (or similar) was used for analysis. Occasionally then, applying a second window before adding the result to the output buffer will help reduce audible artifacts.

Overlap-Add

The purpose of the phase vocoder process is to analyze the frequency content of each short frame, modify it and reconstruct it in the time domain. The manner in which the frames advance from one to the next is important to the proper operation of phase vocoder. Once a frame has been analyzed, processed and synthesized, the effect should advance by the hop size. Suppose the hop size is given by H. Then if the window in frame i previously began at sample m, the window in frame i+1 will begin at sample m+H. The hop size is always less than or equal to the window size; if it were greater than the window size, there would be a gap between successive windows. Smaller hop sizes (more overlap) will often produce better-quality output in many phase vocoder effects, however they require more computation.

Hop sizes of one half, one fourth or one eighth the window size are common. A useful guideline for choosing hop size is the constant overlap-add or COLA criterion. Suppose that in the steps listed above for the phase vocoder theory overview, no processing at all is done in the frequency domain (step 4), and that instead, the FFT is immediately followed by the inverse FFT. For windows and hop sizes meeting the COLA criterion, the output signal at the end of the overlap-add process will be identical to the input signal, with no distortion or modulation introduced by the windowing process. Essentially, this requires that the window functions, when overlapped, add to a constant value, as depicted in . Rectangular windows meet the COLA criterion for a hop size equal to the window size; Bartlett, Hann and Hamming windows require a hop size of at most half the window size. Integer divisions of this maximum hop size are also possible: for example, a hop size of 1/8 the window size will meet the COLA criterion for all four windows.

To summarize, it is important to choose a window and hop size such that the windowing process itself does not introduce unnecessary artifacts into the output. The exact choice of window type, length and hop size depends in large part on the specific effect, several of which are discussed below.

### Filterbank Analysis Variant

Implementation of the phase vocoder is also possible using a filter bank approach. This leads to a computationally more expensive implementation but it can be shown to be theoretically equivalent, and it is often straightforward to consider the phase vocoder as a filter bank.

Returning to the STFT equation, it is clear that:



where  is now used to represent 2k/N.

This can be seen as a demodulation of the signal components at frequency  down to baseband, followed by a low pass filtering of the signal using the filter *h*[*n*]. This is known as the complex baseband filter bank implementation, and is illustrated in , top.

If the variables are switched such that m🡪n-m, we obtain:



Now we can view the filter as being band pass at the frequency . After filtering, the result is then demodulated back down to baseband. This implementation is known as the complex band pass implementation. It is illustrated in , bottom.

For the signal to be reconstructed from the magnitude and phase values of *X*(*n*,*w*), each baseband signal must be modulated back to the frequency .



The signal *y*[*n*] is reconstructed by summing these terms for each frequency bin.

### Oscillator Bank Reconstruction Variant

Since we can assume that the time-domain signal x[n] is real, frequency bins which are symmetric about the Nyquist frequency will be conjugate pairs.



These two signals may then be summed to simplify the analysis. This also results in a more meaningful interpretation:



Oscillator bank reconstruction is clearly computationally expensive. Despite this, it is generally a musically intuitive way of viewing the synthesis stage, especially for musicians. Using an IFFT overlap and add approach is much more efficient. The STFT represents a downsampled version of the filterbank outputs. This is possible due to the filtering step bandlimiting the channel signals.

## Phase Vocoder Effects

The phase vocoder and its underlying theory have a wide range of applications. In the fields of machine listening and music informatics, these include signal content analysis, pitch detection and steady-state/transient separation. In the domain of audio effects, important applications include time-scaling and pitch-shifting. Normally, when an audio file is played at a higher sample rate than it was recorded at, faster playback rate and higher pitch go together. The phase vocoder allows playback speed and pitch to be varied independently of one another. Other common phase vocoder effects include robotization and whisperization voice effects.

### Robotization

The robotization effect is most commonly used on voice signals. It applies a constant pitch to the signal while preserving the vocal formants that determine vowel and consonant sounds, resulting in a robot-like monotone voice that is nonetheless very intelligible.

Within the context of the overlap-add phase vocoder procedure described in Phase Vocoder Theory, , the robotization effect itself is very simple. Following the FFT at each frame, the phase of every frequency bin is set to zero, while the magnitude is left unchanged. Setting the phase to zero is equivalent to making each frequency bin a real number (imaginary component set to 0). Suppose the value of frequency bin k is ak + jbk. Then the output value following robotization will be .

This procedure should be repeated identically for each bin of each frame. By preserving the magnitude, the overall shape of the spectrum remains the same, preserving the vocal formants. But by regularizing the phase information, each frequency component will effectively restart from zero phase on each hop rather than connecting smoothly from one hop to the next. This causes a constant audible pitch which depends on the hop size. In general, the pitch of the robot voice can be determined by:

,

where fs is the sample rate and H is the hop size in samples. The sound of the robotization effect also depends on the window size. In general, moderately-sized windows (roughly 256 to 1024 samples) produce the most striking effect. Very small windows reduce the clarity of the output, where longer windows attenuate the robot-like quality by passing through more of the signal’s original pitch. Intuitively, we can understand this by noting that phase information is used to resolve small frequency differences between bins. If there are a large number of bins, each pitch will be closely determined even without phase information, but if fewer bins are used, then regularizing the phase will better obscure the original frequencies.

Note that for both robotization and whisperization, the constant overlap-add criterion for choosing window and hop size is not an important consideration, since both use deliberate artifacts of the phase vocoder analysis and resynthesis process to achieve their signature sounds.

#### Robotization Code Example

The following C++ code fragment is adapted from the materials that accompany this book. This example implements robotization using the phase vocoder. Details of calculating the FFT and inverse FFT are not included here, as the goal is to show the overall architecture.

int inwritepos; // Write pointer into the input buffer

int outwritepos; // Write pointer into the output (overlap-add) buffer

int outreadpos; // Read pointer into the output buffer

int inputBufferLength; // Length of the input buffer (in samples)

int outputBufferLength; // Length of the output buffer (in samples)

int sampsincefft; // Counter of how many samples have elapsed since last FFT

float \*windowBuffer; // Buffer that holds the (pre-calculated) window function

int windowBufferLength; // Length of the window function

float \*fftTimeDomain; // Buffer that holds time-domain samples for the FFT calculation

float \*fftFrequencyDomain; // Buffer that holds frequency-domain samples from FFT

float fftScaleFactor; // Scaling factor to normalize output level; depends on window/hop sizes

int fftTransformSize\_; // Size of the FFT calculation (in samples); normally equals

// window size but could be longer

int hopSize\_; // Hop size parameter (in samples)

// Collect the audio samples in the input buffer. When we've reached the next

// hop interval, calculate the FFT and process the pitch shift.

for (int i = 0; i < numSamples; ++i)

{

const float in = channelData[i];

// Store the next buffered sample in the output. Do this first before anything

// changes the output buffer-- we will have at least one FFT size worth of data

// stored and ready to go. Set the result to 0 when finished in preparation for the

// next overlap/add procedure.

channelData[i] = outputBufferData[outreadpos];

outputBufferData[outreadpos] = 0.0;

if(++outreadpos >= outputBufferLength)

outreadpos = 0;

// Store the current sample in the input buffer, incrementing the write pointer. Also

// increment how many samples we've stored since the last transform. If it reaches

// the hop size, perform an FFT and any frequency-domain processing.

inputBufferData[inwritepos] = in;

if (++inwritepos >= inputBufferLength)

inwritepos = 0;

if (++sampsincefft >= hopSize\_)

{

sampsincefft = 0;

// Find the index of the starting sample in the buffer. When the buffer length

// is equal to the transform size, this will be the current write position but

// this code is more general for larger buffers.

int inputBufferStartPosition = (inwritepos + inputBufferLength

- fftTransformSize\_) % inputBufferLength;

// Window the buffer and copy it into the FFT input

int inputBufferIndex = inputBufferStartPosition;

for(int fftBufferIndex = 0; fftBufferIndex < fftTransformSize\_; fftBufferIndex++)

{

// Set real part to windowed signal; imaginary part to 0.

fftTimeDomain[fftBufferIndex][1] = 0.0;

if(fftBufferIndex >= windowBufferLength) // Safety check, in case window

// isn't ready

fftTimeDomain[fftBufferIndex][0] = 0.0;

else

fftTimeDomain[fftBufferIndex][0] = windowBuffer[fftBufferIndex]

\* inputBufferData[inputBufferIndex];

inputBufferIndex++;

if(inputBufferIndex >= inputBufferLength)

inputBufferIndex = 0;

}

// Perform the FFT on the windowed data, going into the frequency domain.

// Result will be in fftFrequencyDomain

fftw\_execute(fftForwardPlan\_);

// \*\*\*\*\*\*\*\*\*\* PHASE VOCODER PROCESSING GOES HERE \*\*\*\*\*\*\*\*\*\*\*\*\*\*

// This is the place where frequency-domain calculations are made

// on the transformed signal. Put the result back into fftFrequencyDomain

// before transforming back.

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for(int bin = 0; bin < fftTransformSize\_; bin++)

{

float amplitude = sqrt(fftFrequencyDomain[bin][0]\*fftFrequencyDomain[bin][0]

+ fftFrequencyDomain[bin][1]\*fftFrequencyDomain[bin][1]);

// Set the phase of each bin to 0. phase = 0 means the signal is entirely

// positive-real, but the overall amplitude is the same as before.

fftFrequencyDomain[bin][0] = amplitude;

fftFrequencyDomain[bin][1] = 0.0;

}

// Perform the inverse FFT to get back to the time domain. Result wll be

// in fftTimeDomain. If we've done it right (kept the frequency domain

// symmetric), the time domain resuld should be strictly real allowing us

// to ignore the imaginary part.

fftw\_execute(fftBackwardPlan\_);

// Add the result to the output buffer, starting at the current write position

// (Output buffer will have been zeroed after reading the last time around)

// Output needs to be scaled by the transform size to get back to original

// amplitude: this is a property of how fftw is implemented. Scaling will also

// need to be adjusted based on hop size to get the same output level (smaller

// hop size produces more overlap and hence higher signal level)

int outputBufferIndex = outwritepos;

for(int fftBufferIndex = 0; fftBufferIndex < fftTransformSize\_; fftBufferIndex++)

{

outputBufferData[outputBufferIndex] += fftTimeDomain[fftBufferIndex][0] \*

fftScaleFactor;

if(++outputBufferIndex >= outputBufferLength)

outputBufferIndex = 0;

}

// Advance the write position within the buffer by the hop size

outwritepos = (outwritepos + hopSize\_) % outputBufferLength;

}

}

Most of the code is devoted to the mechanics of the overlap-add phase vocoder process, including accumulating enough samples to fill a window, applying the window function, and transforming it to the frequency domain. The specific robotization code goes through each frequency bin and sets the phase to 0. The remainder of the code performs an inverse FFT and adds the result to the output buffer, with the multiplier fftScaleFactor (dependent on the combination of window size and hop size) needed for the input and output levels to match. The function fftw\_execute() performs the FFT or IFFT using the fftw library. Other FFT libraries could be used with minor changes to the code.

This code fragment does not include the initialization, which includes several important steps such as calculating the window, calculating fftScaleFactor, setting initial values for the read and write pointers, and preparing the FFT library.

### Whisperization

The whisperization effect, like robotization, is also most commonly used on voice signals. It maintains the vocal formants while completely eliminating any sense of pitch. The resulting effect sounds similar to a person whispering: the contents of speech remain clear while any sense of voicing is lost.

Whisperization is implemented similarly to robotization, but instead of setting the phase of each frequency bin to zero at every frame, the phase is set to a random value in the range φ = [0, 2π). Assuming the original contents of bin k can be written ak + i\*bk and φk holds the random phase value for bin k, the output is:

.

Notice that a different random value is used for each frequency bin, each frame. As with robotization, the magnitude of each bin is preserved which maintains the overall shape of the spectrum. But scrambling the phase erases any sense of periodicity from one frame to the next.

The most important parameter determining the whisperization effect is window size. In this case, shorter windows (e.g. 64 to 256 samples) are typically most effective at eliminating any pitch content. If the window is too short, the effect becomes unintelligible, but for longer windows, the original pitch of the signal remains apparent. Hop size is a less important parameter that can be adjusted experimentally, though generally it should be no longer than the window length or other artifacts will result.

#### Whisperization Code Example

To adapt the previous robotization code example to instead perform whisperization, the basic overlap-add structure stays the same and we only need to change the lines following the FFT calculation:

for(int bin = 0; bin <= fftTransformSize\_ / 2; bin++)

{

float amplitude = sqrt(fftFrequencyDomain[bin][0]\*fftFrequencyDomain[bin][0] +

fftFrequencyDomain[bin][1]\*fftFrequencyDomain[bin][1]);

// This is what we would use to exactly reconstruct the signal:

// float phase = atan2(fftFrequencyDomain[bin][1], fftFrequencyDomain[bin][0]);

// But instead, this is what we use to scramble the phase:

float phase = 2.0 \* M\_PI \* (float)rand() / (float)RAND\_MAX;

// Set the phase of each bin to 0. phase = 0 means the signal is entirely

// positive-real, but the overall amplitude is the same as before.

fftFrequencyDomain[bin][0] = amplitude \* cos(phase);

fftFrequencyDomain[bin][1] = amplitude \* sin(phase);

// FFTs of real signals are conjugate-symmetric. We need to maintain that symmetry

// to produce a real output, even as we randomize the phase.

if(bin > 0 && bin < fftTransformSize\_ / 2) {

fftFrequencyDomain[fftTransformSize\_ - bin][0] = amplitude \* cos(phase);

fftFrequencyDomain[fftTransformSize\_ - bin][1] = -amplitude \* sin(phase);

}

}

This code calculates the amplitude and phase of each bin based on the real and imaginary components. However instead of using the phase of the original signal, the phase is replaced with a random number between 0 and 2π. Amplitude and phase are converted back to real and imaginary values using sin and cos.

Another change between robotization and whisperization code occurs in the first line of the for() loop. Notice that in this example, the loop only goes through the first half of the bins. This is because the FFT of a real-valued signal is always conjugate-symmetric: F(k) = F(N - k)\* where k is the bin and N is the total transform size. If we want the output to remain as a real signal, we need to maintain conjugate symmetry even as we randomize the phase. The code inside the last if() statement fills in the second half of the frequency bins using the complex conjugates of the first half.

### Time Scaling

The phase vocoder allows a signal to be rescaled in time without any change in pitch. To perform time scaling, the basic idea is to take advantage of the basic relationship between time, frequency and phase, =t. Looking at this equation, we can see that frequency can be preserved, and time can be varied, at a cost of the original analysis phase being lost. Despite this loss of phase preservation, results are surprisingly good for many phase vocoder applications, especially when a short analysis hop size, such as 1/8 of the window length, is used.

When time scaling using the phase vocoder, the analysis stage (steps 1-3 in Phase Vocoder Theory, ) is performed identically to any other phase vocoder application. The synthesis stage (steps 5-7) is then varied to facilitate time compression and expansion. The analysis and synthesis hop sizes are no longer equal.

If R is the stretching ratio (e.g. for 20% expansion, R equals 1.2) applied to the signal, the synthesis hop size, hs, is related to the analysis hop size, ha, by hs, is =Rha

Recall that the phase vocoder analysis offers a method for calculation of the instantaneous frequency using phase information. This instantaneous frequency is proportional to the phase increment over a single analysis hop. This phase increment is the parameter which is used in this time scaling application. If we consider a single frequency bin *k,* representing a sinusoidal track over time, the phase and amplitude increment per sample within hop n, are given by:

,

where ha is the analysis hop size. Refer to Eq.s to for explanation of the terms.

Time Scaling Resynthesis

These values may now be used in the resynthesis stage. For the same sinusoidal track, the phase is calculated at each sample, m, using:

.

In this case, q is used to denote synthesis phase. The synthesis phase is then incremented by the same amount as was calculated in the analysis, for the number of samples in the synthesis hop size. This leads to a difference in phase by the end of the hop (recall the phase deviation mentioned earlier), as long as the synthesis hop size differs from that of the analysis hop size, which is a definition of time scaling. In the case of amplitude, the analysis increment cannot be used in the same way, as the same cumulative errors applied to amplitude would lead to severe artifacts compared to phase.

The amplitude increment is calculated instead using the synthesis hop size, to maintain identical analysis and synthesis amplitudes at both the beginning and end of the hop.



The amplitude can then be calculated for the duration of the synthesis hop size using:

.

These sinusoidal components are then summed to synthesize the time scaled audio:



Clear problems arise from time scaling using a spectral method such as this, discussed in a later section. Also, by definition, time-scaling cannot be a real-time audio effect except for a limited period of time. If the output is compressed in time compared to the input (i.e. plays faster), then at some point the effect would become non-causal since the output would depend on future input samples. On the other hand, if the output is stretched in time (plays slower), the difference in time between input and output will steadily accumulate and an ever-increasing amount of memory will be needed to buffer the audio waiting to play. However, the converse case of pitch-shifting without time-scaling, discussed in the following section, is a common real-time audio effect.

### Pitch Shifting

There are several ways to shift the pitch of a signal without changing its speed using the phase vocoder. One of the most straightforward approaches uses the time-scaling algorithm presented in the previous section. Suppose we apply a time stretch factor of R; then every block of N input samples will produce R\*N output samples after time-scaling. If we then changed the sample rate to play R times faster, then the output would have the same speed as the input but with a pitch R times higher.

In practice, we don’t actually change the sample rate at the output but we achieve a similar effect using interpolation to fit R·N samples in the space of N. If x[n], 0 < n < R·N-1, represents the output of a single frame of the phase vocoder, then we can calculate the pitch-shifted output using linear interpolation:

.

This interpolation is performed for each frame of the phase vocoder output. With time-scaling, the synthesis hop size was R\*h samples, but in the pitch-shifting, each of the interpolated buffers advances by the analysis hop size h, ensuring that input and output signals remain at the same speed despite the pitch shift.

To summarize, raising the pitch of the output requires first stretching the signal in time (R > 1), then compressing the longer output buffer with interpolation to fit in the original length. Lowering the pitch requires compressing the signal in time, producing fewer output samples which are then stretched with interpolation to fit the original length.

Code Example

The following C++ code fragment uses much of the same phase vocoder structure as the robotization and whisperization effects, but instead performs real-time pitch shifting. Some initial code which is identical to the robotization example has been omitted.

int outwritepos; // Temporary write pointer

float \*inputBufferData; // Buffered input samples awaiting FFT

int inputBufferLength; // Length of the input buffer (in samples)

float \*outputBufferData; // Buffered output samples for overlap-add

int outputBufferLength; // Length of the output buffer (in samples)

int sampsincefft; // Counter of how many samples have elapsed since last FFT

float \*windowBuffer; // Buffer that holds the analysis window function

int windowBufferLength; // Length of the analysis window function

float \*synthWindowBuffer; // Buffer that holds the synthesis window function

int synthesisWindowLength;// Length of the synthesis window

float \*fftTimeDomain; // Buffer that holds time-domain samples for the FFT calculation

float \*fftFrequencyDomain; // Buffer that holds frequency-domain samples from FFT

float fftScaleFactor; // Scaling factor to normalize output level; depends on window/hop sizes

float \*resampledOutput; // Buffer holding resampled (interpolated) output from FFT

float \*\*lastPhase; // Previous phase values for each bin and channel

float \*\*psi; // Adjusted phase values for each bin and channel

int fftTransformSize\_; // Size of the FFT calculation (in samples); normally equals

// window size but could be longer

double pitchRatio\_; // Ratio of output to input frequency

int analysisHopSize\_; // Hop size parameter for input (in samples)

int synthesisHopSize\_; // Hop size parameter for output (in samples)

// synthesis / analysis size should match pitchRatio\_

// [...]

// Up until this point, the code is the same as the robotization

// example earlier in the chapter. The code below executes the pitch

// shift for one (overlapped) FFT window.

// [...]

fftw\_execute(fftForwardPlan\_);

for (int i = 0; i < fftTransformSize\_; i++) {

// Convert the bin into magnitude-phase representation

double magnitude = sqrt(fftFrequencyDomain[i][0] \* fftFrequencyDomain[i][0]

+fftFrequencyDomain[i][1] \* fftFrequencyDomain[i][1]);

double phase = atan2(fftFrequencyDomain[i][1], fftFrequencyDomain[i][0]);

// Calculate frequency for this bin

double frequency = 2.0 \* M\_PI \* (double)i / fftTransformSize\_;

// Increment the phase based on frequency and hop sizes

double deltaPhi = (frequency \* analysisHopSize\_) +

princArg(phase - lastPhase[i][channel] -

(frequency \* analysisHopSize\_));

lastPhase[i][channel] = phase;

psi[i][channel] = princArg(psi[i][channel] + deltaPhi \* synthesisHopSize\_);

// Convert back to real-imaginary form

fftFrequencyDomain[i][0] = magnitude \* cos(psi[i][channel]);

fftFrequencyDomain[i][1] = magnitude \* sin(psi[i][channel]);

}

// Perform the inverse FFT

fftw\_execute(fftBackwardPlan\_);

// Resample output using linear interpolation to stretch it

double outputLength = floor(fftTransformSize\_ / pitchRatio\_);

for(int i = 0; i < outputLength; i++) {

x = i \* fftTransformSize\_ / outputLength;

ix = floor(x);

dx = x - (double)ix;

resampleOutput[i] = fftTimeDomain[ix]\*(1.0 - dx) +

fftTimeDomain[(ix+1)%fftTransformSize\_]\*dx;

}

// Add the result to the output buffer, starting at the current write position

int outputBufferIndex = outwritepos;

for(int fftBufferIndex = 0; fftBufferIndex < outputLength; fftBufferIndex++) {

if (fftBufferIndex > synthesisWindowLength)

outputBufferData[outputBufferIndex] += 0;

else

outputBufferData[outputBufferIndex] += resampleOutput[fftBufferIndex]

\* fftScaleFactor \* synthesisWindowBuffer[fftBufferIndex];

if(++outputBufferIndex >= outputBufferLength)

outputBufferIndex = 0;

}

// Advance the write position within the buffer by the hop size

// (Use the original hop size since we have resampled the output back to the

// expected length)

outwritepos = (outwritepos + analysisHopSize\_) % outputBufferLength\_;

In this example, princArg() implements the principal argument function (princ). After performing the FFT, each bin is converted to magnitude-phase representation. At that point, the phase is updated based on Eq. so the windows can be overlapped at a different synthesis hop size. Normally, making analysis and synthesis hop sizes different would result in a time stretch. However, the output of the inverse FFT is resampled and then overlapped at the original (analysis) hop size. This results in a pitch shift without changing the timing of the signal.

### Phase Vocoder Artifacts

Amplitude estimation is not easily achieved with linear interpolation, especially when using a long synthesis window. Consider the idea of a piano piece being played more slowly: the attacks would have much the same temporal envelope, but steady state and decay regions would be longer before the start of the next note although the decay rate would be the same. Linear interpolation implies that all regions of the envelope stretched equally.

Also, problems arise in phase relationships, as the phase error is cumulative. If phase coherence is lost at transients (note attacks), transient smearing occurs which softens the attacks and loses the natural sound of these regions. Loss of phase coherence across harmonic partials of the same note can lead to loss of natural sounding steady state regions.

Finally, a ‘phasiness’ is introduced, which adds a reverb-like quality to the sound. One reason for this ‘phasiness’ is the loss of coherence of phase within the same sinusoidal component. If the phase is not maintained across all components which correspond to a windowed sinusoid, although frequency is maintained, severe amplitude distortions can occur. This produces a ‘robot-like’ effect when applied to signals such as voice.

Further problems arise when chirp-like signals are applied to phase vocoders. When the signal crosses over to the next frequency bin, it takes the phase of the new frequency track, leading to discontinuities. This is the kind of problem that can be solved by using higher level peak picking and sinusoidal tracking.

## Problems

1. Write pseudo-code to perform a short-time-fourier transform on a sampled signal. You can call another function to perform the FFT.

2. Consider the block diagram of the phase vocoder given by Figure 0.1. If a Hamming window of 1024 samples is used, what is the maximum hop size that will allow perfect reconstruction of the signal? Why?

3. Suppose the sampling frequency is 32.768kHz and the FFT size is 1024 points. What is the width of one frequency bin (i.e. how far apart in frequency are two adjacent bins)?

4. We can perfectly reconstruct any signal under the right conditions, but the FFT has limited frequency resolution. Suppose the input is a sine wave which doesn't exactly align with any bin frequency. Explain how phase information can be used to determine the exact frequency..

5. Explain the operation of the robotisation effect, and explain what parameter determines the frequency of the robot voice.

6. Explain the operation of the whisperisation effect. What is the restriction on the window length and why is it necessary?

7. Explain the steps involved in a pitch shifter effect (changing pitch without changing speed).



Figure .1. Overview of the overlap-add process for phase vocoder effects.



Figure .2. Four popular window functions.



Figure .3. The effect of a rectangular window and a Hann window applied to a signal.



Figure .4. Relationship between actual phase and target phase. The instantaneous frequency is represented by the gradient of the dashed line, and the bin frequency is the gradient of the solid line.



Figure .5. The constant overlap-add criterion requires that the window functions, when overlapped, add to a constant value. It holds in the top plot, but not the bottom plot.



Figure .6. The complex baseband filterbank implementation (top) where h[n] is a low pass filter, and the complex band pass implementation (bottom). The filters for the first three bins are shown only.

[1] J. L. Flanagan and R. M. Golden, "Phase Vocoder," *Bell System Technical Journal,* pp. 1493-1509, Nov. 1966.

[2] M. Portnoff, "Implementation of the digital phase vocoder using the fast Fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* vol. 24, pp. 243- 248, 1976.

[3] J. Laroche and M. Dolson, "Improved phase vocoder timescale modification of audio," *IEEE Trans. Speech and Audio Processing,* vol. 7, pp. 323 -332, 1999.

[4] M. Dolson, "The phase vocoder: A tutorial " *Computer Music Journal,* vol. 10, pp. 14-27, 1986.

[5] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing*, 3rd ed.: Prentice Hall, 2009.

[6] S. J. Orfanidis, *Introduction to Signal Processing* Orfanidis (prev. Prentice Hall), 2010.

[7] J. O. Smith, *Spectral Audio Signal Processing*: W3K Publishing 2011.