On-line article published on www.cycling74.com, October, 2006.

The Phase Vocoder – Part I

Richard Dudas and Cort Lippe

Introduction

The phase vocoder is a tool used to perform time-stretching and pitch-shifting on recorded sounds. Its name derives from the early "vocoders" (contraction from "voice encoders"), which used a set of bandpass filters in parallel over many frequency bands, to crudely analyze and reconstruct speech. The infamous hardware vocoders of the 1960s and 1970s (as used, for example, by Wendy Carlos in the soundtrack of Kubrick's film "A Clockwork Orange") were based on this technology, and allowed the spectrum of one sound (in the Carlos example a synthesizer) to be controlled by that of another (a voice). In the Max/MSP examples folder, there is an example by Marcel Wierckx called "classic_vocoder.pat" (located in the "effects" sub-folder) which shows how this traditional vocoder works. Unlike the classic vocoder, which is based on bandpass filters, the phase vocoder is based on a Short-Term Fourier Transform (STFT) – a Fourier Transform performed sequentially on short segments of a longer sound – and in practical use has little to do with the hardware vocoders of the 1960s and 1970s. The phase vocoder can, however, be considered a type of vocoder because the Fourier Transform returns a set of amplitude values for a set of frequency bands spaced evenly across the sound's spectrum, similar to the older vocoder's set of bandpass filters. Of course the phase vocoder, as it's name suggests, not only takes into account the amplitude of these frequency bands, but also the phase of each band.

If you are not familiar with the Fast Fourier Transform (FFT) as it is used in MSP, we suggest you review MSP Tutorials 25 and 26, which deal with the fft~/ifft~ objects and the pfft~ object set, respectively.

Our Starting Point

In MSP's Tutorial 26 on the pfft~ object, we are shown a simple phase vocoder patch which analyzes an incoming sound and records the FFT frames into a buffer~ object. The data in the buffer~ object is then reconstructed so that a basic sort of time-stretching (and compression) may be performed on the recorded data. Although it works as advertised and introduces the basic concepts, it lacks some of the flexibility of a more standardly-designed phase vocoder, which lets us transpose our original sound, as well as start at any point in the original sound.

Let's review the STFT and basic phase vocoder design. A Short Term Fourier Transform (STFT) is a series of Fourier Transforms, usually spaced evenly in time:

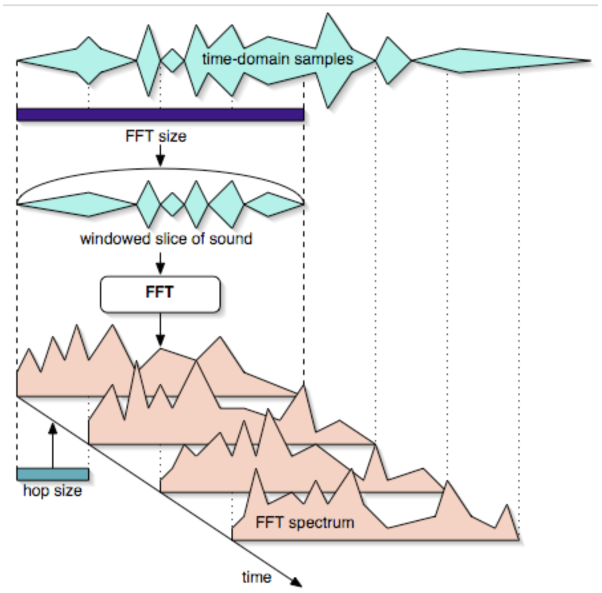


Fig. 1 – Diagram of the Short Term Fourier Transform (STFT).

Whereas the old fft~ and ifft~ objects just perform the FFT, the pfft~ object actually goes one step further and performs an STFT, as shown above. The input signal is sliced into frames, which start at a regular offset known as the hop size. The hop size determines the overlap, which can be defined as the number of frames superimposed on each other at any given time. If the hop size is half the FFT frame size, the overlap is 2, and if it is one-fourth the frame size, the overlap is 4. Theoretically, an STFT needs a hop size of at least 4 and although a hop size of 2 can be made to work for many musical purposes, the phase vocoder will sound better with an overlap of 4.

If our FFT size is 1024, our FFT will give us 1024 frequency bins (these would be called bands in filter terminology) from DC (0Hz) to the sampling frequency. Because the data above the Nyquist Frequency (half the sampling rate) is a mirrored copy of the data below the Nyquist Frequency, the pfft~ object by default eliminates this redundant data for efficiency's sake, and reconstructs it for us before performing the inverse FFT. (In our phase vocoder patch we will need to tell the pfft~ object to override this elimination of data for reasons that will become clear later in this article).

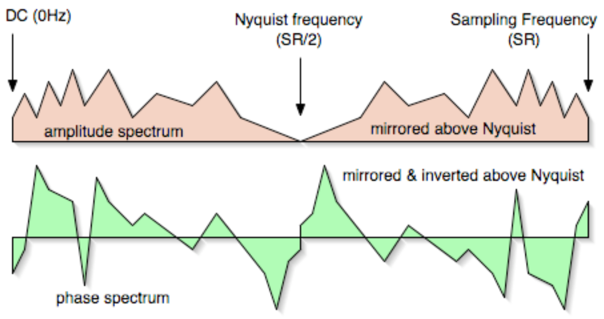


Fig. 2 – Diagram of the FFT Spectrum

For each bin, the FFT provides us with coordinates which can be converted to amplitude and phase values. (This is covered in MSP's FFT tutorials.) The difference between the phase values of successive FFT frames for a given bin determines the exact frequency of the energy centered in that bin. This is often known as the phase difference (and sometimes also referred to as phase derivative or instantaneous frequency if it's been subjected to a few additional calculations).

If we record a succession of FFT frames, then play them out of order, the differences between the phase values in the bins will no longer produce the correct frequency content for each bin. Therefore, in order to "reconstruct" a plausible playback of re-ordered FFT frames, we need to calculate the phase difference between successive frames and use it to construct a "running phase" (by simply summing the successive differences) for the output FFT frames.

Getting Sound into the Phase Vocoder

The first thing we need to do is to be able to is to access our sound file directly in our pfft~ sub-patch and perform an FFT directly on the sound, without sending it into any fftin~ inlets of the pfft~. Why do we need to do it this way? Since the whole point of a phase vocoder is to perform time stretching and compression on an existing sound, we need to be able to access that sound directly via a buffer~ object and perform the FFT at any given playback speed and overlap. In order to have independent transposition and playback speed, a phase vocoder needs independent playback of the sound to be transformed for each FFT overlap (in our case 4). Each playback frame needs to be synchronized with its respective FFT frame. Therefore we cannot send a single copy of the sound we wish to transform into the pfft~, but need to play a slice of the sound into each of the four overlap frames. Since we cannot send one slice of the sound into the pfft~ object (it keeps track of its input history and considers its input to be continuous sound, not individual slices of sound), we cannot use the fftin~ object inside the pfft~, but must do our FFT processing using the fft~ object. The fft~ object performs a full-spectrum FFT (i.e. mirrored), so we consequently need to make the pfft~ object behave the same way, so the fft~ can work in sync with the FFT frames processed inside the pfft~ object. We need to make a few changes to the default way that pfft~ deals with the STFT.

First, we first need to tell the pfft~ object to process full-spectrum FFT frames, instead of the default “spectral frame” which is half the FFT size (up to half the Nyquist). This is easily accomplished by adding a non-zero fifth argument to the pfft~ object. Because the full-spectrum argument is the fifth argument, we must supply all the other arguments before it, including the fourth argument, the start onset, which will be set to the default value of zero.

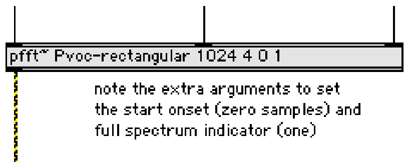

pfft~ Pvoc-rectangular 1024 4 0 1
note the extra arguments to set the start onset (zero samples) and full spectrum indicator (one)

Fig. 3 – Additional Full-Spectrum Argument to the pfft~ Object

Next, because the fftin~ and fftout~ objects perform the FFT calculation at zero phase with respect to the FFT (the first sample in the windowed frame sent to the FFT is the middle of the window), and the traditional fft~ and ifft~ objects perform the FFT 180 degrees out of phase, we need to make sure any fftin~ and fftout~ objects in our patch have the same FFT phase offset used in the fft~ objects. We do this by specifying a phase offset to the fftin~ and fftout~ objects. A phase value of 0.5 means 180 degrees out of phase, so this is the value we want. While we do not need the fftin~ object in our pfft~, we can still make use of the convenience of the fftout~ object in order to get the properly windowed and overlap-added result out of our pfft~. The automatic windowing in the fftout~ object should behave like our manual windowing with the fft~ objects.

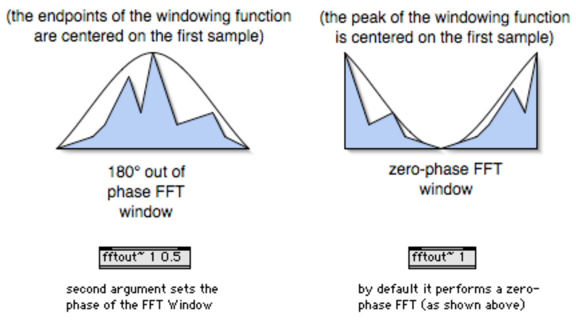
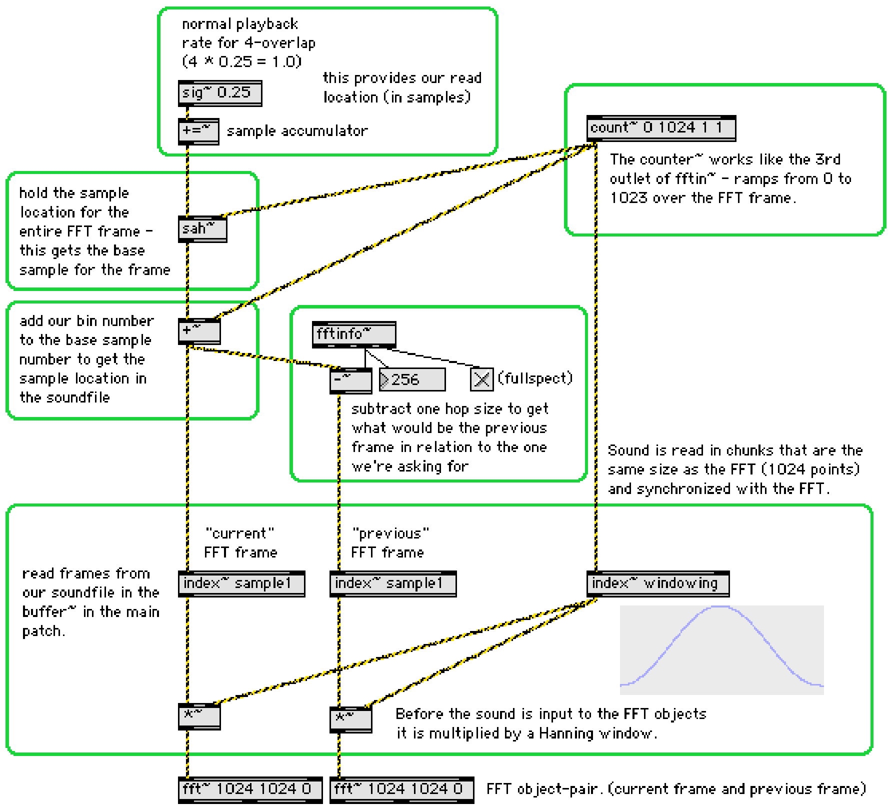

(the endpoints of the windowing function are centered on the first sample)
(the peak of the windowing function is centered on the first sample)
180° out of phase FFT window
zero-phase FFT window
fftout~ 1 0.5
fftout~ 1
second argument sets the phase of the FFT Window
by default it performs a zero-phase FFT (as shown above)

Fig. 4 – Zero-Phase FFT Window

Now we are ready to start constructing our phase vocoder sub-patch.

Accessing our Pre-Recorded buffer~

We always need to access our buffer~ in two places – at the current FFT frame location, and at the location of the previous FFT frame of the buffer. We can use the index~ object to access the buffer~, just as we might use the index~ object in a simple playback patch. (Note that we are accumulating samples at the “normal” playback rate for now.) And because we’re manually designing the input part of our STFT ourselves, using the fft~ object, we need to window the signal we read from the buffer~ before sending it to the fft~ objects. The patch uses a Hanning window (the default window used by pfft~).



normal playback rate for 4-overlap (4 * 0.25 = 1.0)

this provides our read location (in samples)

sample accumulator

hold the sample location for the entire FFT frame - this gets the base sample for the frame

The counter~ works like the 3rd outlet of fftin~ - ramps from 0 to 1023 over the FFT frame.

add our bin number to the base sample number to get the sample location in the soundfile

subtract one hop size to get what would be the previous frame in relation to the one we're asking for

Sound is read in chunks that are the same size as the FFT (1024 points) and synchronized with the FFT.

read frames from our soundfile in the buffer~ in the main patch.

"current" FFT frame

"previous" FFT frame

Before the sound is input to the FFT objects it is multiplied by a Hanning window.

FFT object-pair. (current frame and previous frame)

Fig. 5 – Reading Frames from a buffer~ inside a pfft~

Using two fft~ objects that are a quarter frame apart from each other, we calculate two FFT frames (the present frame and the previous). Using cartopol~, we convert from cartesian coordinates to polar coordinates to get amplitudes and phases, and then simply subtract the phases of the previous frame from the present frame to get phase differences for each bin. Just as in the simplified phase vocoder in Tutorial 26, we use the frameaccum~ object to accumulate the phase differences to construct the "running phase" for output.

Additionally, at the top of the patch we no longer have a fixed playback rate (it was set to 0.25 in the previous example image), but have added the capability to time-stretch (or compress) the sound by accepting a (variable) sample increment from the main patch.

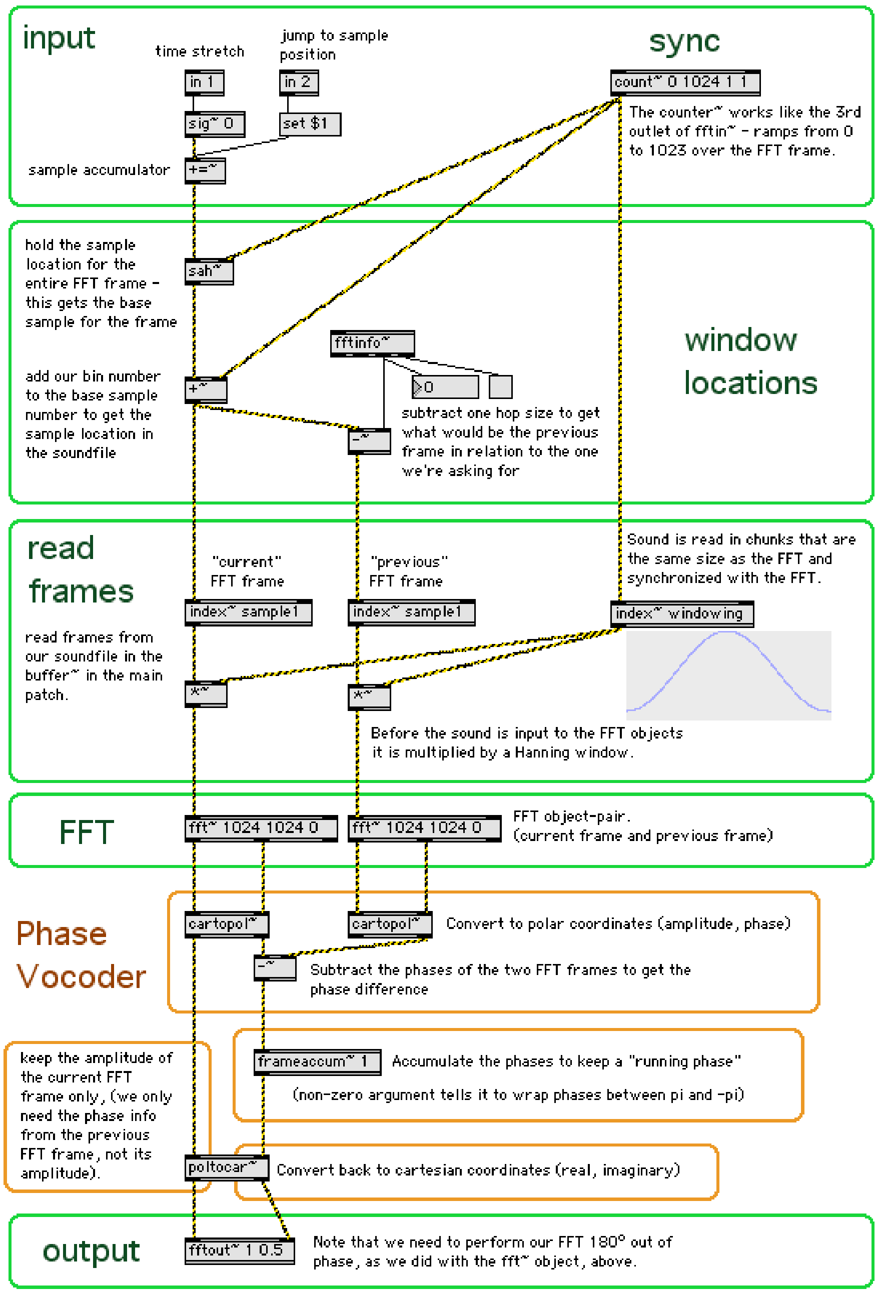

input
time stretch
jump to sample position
in 1
in 2
sig~ 0
set $1
sample accumulator
+=~
sync
count~ 0 1024 1 1
The counter~ works like the 3rd outlet of fftin~ - ramps from 0 to 1023 over the FFT frame.
hold the sample location for the entire FFT frame - this gets the base sample for the frame
sah~
add our bin number to the base sample number to get the sample location in the soundfile
+~
fftinfo~
>0
-~
subtract one hop size to get what would be the previous frame in relation to the one we're asking for
window locations
read frames
"current" FFT frame
"previous" FFT frame
Sound is read in chunks that are the same size as the FFT and synchronized with the FFT.
index~ sample1
index~ sample1
index~ windowing
read frames from our soundfile in the buffer~ in the main patch.
*~
*~
Before the sound is input to the FFT objects it is multiplied by a Hanning window.
FFT
fft~ 1024 1024 0
fft~ 1024 1024 0
FFT object-pair. (current frame and previous frame)
Phase Vocoder
cartopol~
cartopol~
Convert to polar coordinates (amplitude, phase)
-~
Subtract the phases of the two FFT frames to get the phase difference
keep the amplitude of the current FFT frame only, (we only need the phase info from the previous FFT frame, not its amplitude).
frameaccum~ 1
Accumulate the phases to keep a "running phase"
(non-zero argument tells it to wrap phases between pi and -pi)
poltocar~
Convert back to cartesian coordinates (real, imaginary)
output
fftout~ 1 0.5
Note that we need to perform our FFT 180° out of phase, as we did with the fft~ object, above.

Fig. 6 – The Phase Vocoder pfft~ Sub-Patch

Notice that now the "previous" frame that we read from the buffer~ might not actually be the frame that we previously read as the "current" frame! This is the whole point of the phase vocoder – we are able to read frames in any location in the buffer and at any speed, and by simultaneously reading the frame one hop-size before the "current" frame (regardless of the speed at which we're reading the buffer~) we can obtain the "correct" phase difference for the "current" FFT frame!

Cartesian and Polar Coordinates

The FFT algorithm outputs transformed data in cartesian (x, y) coordinates. These coordinates are often referred to as the real and imaginary parts of a complex number. The amplitude and phase values that we normally associate with the FFT are the polar coordinates of the (x,y) values. The polar coordinates of cartopol~ conveniently give us amplitude and phase information. While working in polar coordinates is convenient, Cartesian to polar conversion of phase information uses the trigonometric math function arctan~, which is computationally very expensive. Avoiding the arctan calculation, which must be calculated for each of the eight

FFTs used in a single phase vocoder, by using less intuitive Cartesian math means working with complex math (the complex multiply and divide instead of the simple addition, subtraction, multiplication, and division needed in the polar domain). In addition to the issue of computational efficiency, for reasons of accuracy, converting to and from polar coordinates can introduce small amounts of error which slowly accumulate over time and probably should be avoided. Finally, there are some additional features that improve the quality of the phase vocoder which we will see in Part II of this article (scheduled in two months' time), for which it is preferable to use complex math on cartesian coordinates instead of calculating on the polar coordinates derived from them. So we need to learn a little complex math and how it relates to the polar calculations we're performing on the amplitude and phase values.

A complex multiply multiplies the amplitudes and adds the phases of two complex signals (i.e. signals which have a real and imaginary part – such as the two outputs of the fft~ or fftin~ objects).

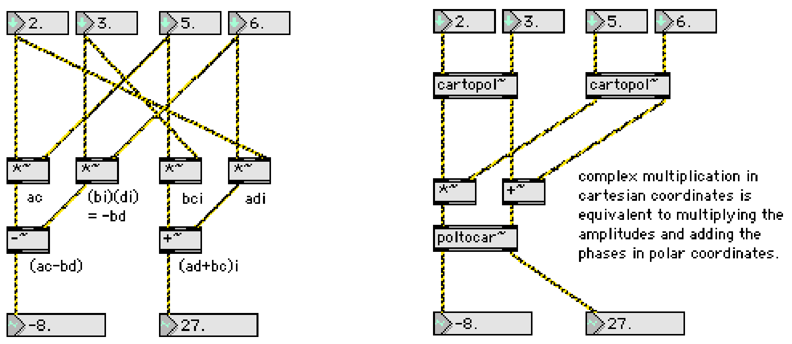

2.
3.
5.
6.
*~
ac
(bi)(di)
= -bd
bci
adi
-~
(ac-bd)
+~
(ad+bc)i
-8.
27.
cartopol~
poltocar~
complex multiplication in cartesian coordinates is equivalent to multiplying the amplitudes and adding the phases in polar coordinates.

Fig. 7. – Complex Multiplication

A complex divide divides the amplitudes and subtracts the phases.

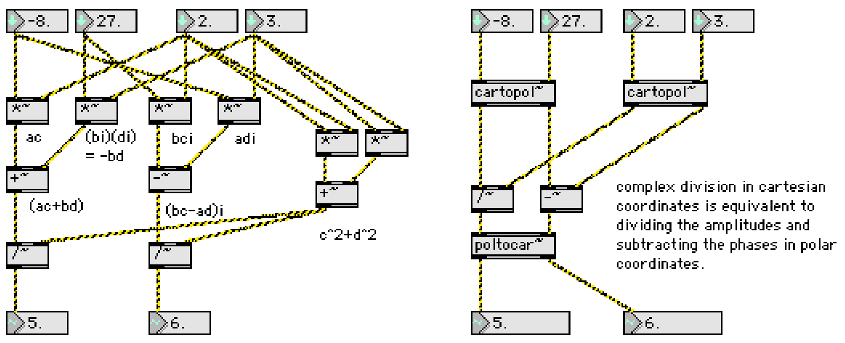

-8.
27.
2.
3.
*~
ac
(bi)(di)
= -bd
bci
adi
+~
(ac+bd)
-~
(bc-ad)i
c^2+d^2
/~
5.
6.
cartopol~
poltocar~
complex division in cartesian coordinates is equivalent to dividing the amplitudes and subtracting the phases in polar coordinates.

Fig. 8. – Complex Division

What is important about these complex multiplication and division sub-patches is what it does to the phases of our two complex signals. Complex multiplication adds phases, whereas complex division subtracts phases. Therefore we can calculate both the phase difference as well as accumulate the running phase of our output signal using these complex math operators. Because we only care about the phase in our phase vocoder patch (remember in the polar version shown previously we did not modify the amplitudes), we can make a further optimization to the complex division and remove the denominator by which both real and imaginary parts are scaled:

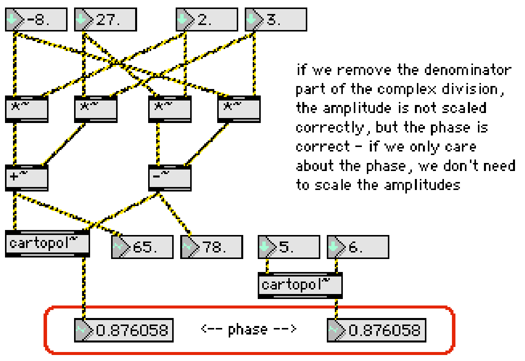

-8.
27.
2.
3.
*~
*~
*~
*~
+~
-~
cartopol~
65.
78.
5.
6.
cartopol~
0.876058
<-- phase -->
0.876058
if we remove the denominator part of the complex division, the amplitude is not scaled correctly, but the phase is correct - if we only care about the phase, we don't need to scale the amplitudes

Fig. 9. – Complex Phase Subtraction Based on Division

The Phase Vocoder Using Complex Math

Now we're ready to use these to construct a phase vocoder which uses complex multiplication and division. The first part of the patch will remain the same – we are only changing what happens between the fft~ objects that read the buffer~, and the fftout~ at the bottom of the patch.

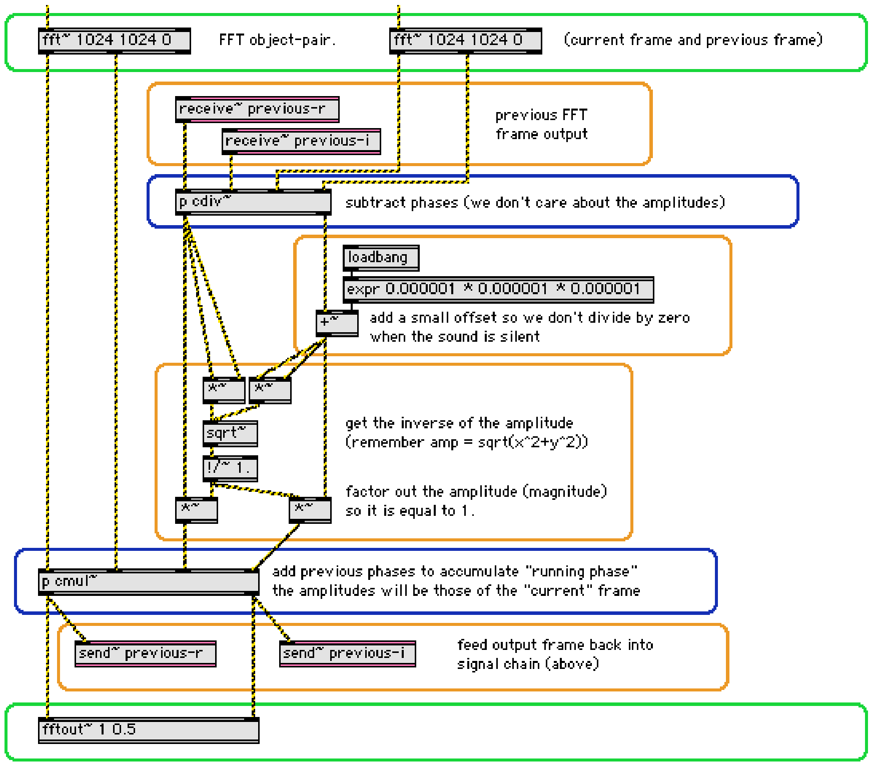

fft~ 1024 1024 0
FFT object-pair.
fft~ 1024 1024 0
(current frame and previous frame)
receive~ previous-r
receive~ previous-i
previous FFT frame output
p cdiv~
subtract phases (we don't care about the amplitudes)
loadbang
expr 0.000001 * 0.000001 * 0.000001
+~
add a small offset so we don't divide by zero when the sound is silent
*~
*~
sqrt~
!/~ 1.
*~
*~
get the inverse of the amplitude (remember amp = sqrt(x^2+y^2))
factor out the amplitude (magnitude) so it is equal to 1.
p cmul~
add previous phases to accumulate "running phase" the amplitudes will be those of the "current" frame
send~ previous-r
send~ previous-i
feed output frame back into signal chain (above)
fftout~ 1 0.5

Fig. 10. – Phase Vocoder using Complex Math

Notice how we have to use the send~ and receive~ objects to manually feed-back the previous output frame so we can use it to accumulate the running phase. Remember that a receive~ object "receives" signal data from its corresponding send~ object with a delay of one signal vector in order to avoid signal loops. The signal vector of a pfft~ is the size of the FFT frame, and since we are running a full-spectrum pfft~, the delay is 1024 samples, or one FFT frame.

Also notice how we invert the amplitude values so we can produce a frame which contains JUST the phases, with the amplitudes set to 1.0. (The inverted amplitude, or magnitude, multiplied by the real and imaginary values essentially cancels out the amplitude.) This is so we can use our complex divide and multiply and not affect the amplitude values of the "current" frame. In order to make sure we don't divide by zero, we add a very small offset to the complex number before calculating the inverse amplitude. This does not affect the sound perceptibly.

At this point you might want to compare both the polar and cartesian versions of the phase vocoder patch. The polar version which we first showed you is conceptually clearer at first glance. However, note the difference in CPU usage of the two patches. You may well decide that the extra effort in designing the phase vocoder using complex arithmetic is worth the payoff!

In listening to the phase vocoder patch, you may have noticed a somewhat metallic or slightly reverberated quality to the phase vocoder's sound as it runs. This is caused by phase inconsistencies from bin to bin. Since any given FFT bin might be affected by more than one component of a sound—for instance, a non-harmonic sound with two very close inharmonically related frequencies will struggle for precedence in a bin that they share—this can create a slightly reverberated quality to the sound. In Part II of the article we will learn some tricks to minimize this effect, as well as look at a buffer~ reader that allows for flexible and independent transposition and stretching at any location in the buffer~.

Conclusion

To sum up what we've just covered, the basic stucture of the phase vocoder requires four overlapping *pairs* of buffer~ reads and four overlapping *pairs* of FFTs. The buffer/FFT pairs are exactly one quarter frame apart (one hop). The FFT pairs allow us to calculate the phase differences between where we are in a sound and where we were a quarter frame ago (one hop). Then, for each of the four pairs, we simply add their respective phase difference to the previous phase from a full frame before, accumulating our running phase. We do all this with Cartesian coordinates (real and imaginary values making use of complex arithmetic) using fft~ objects inside a pfft~ that is running in full-spectrum mode with a 180-degree phase offset for the fftout~ object so that it runs in phase with the fft~ objects.

Acknowledgements

The authors would like to thank Miller Puckette for developing what was more than likely the first real-time phase vocoder, and most certainly the first real-time phase vocoder using Cartesian coordinates!