Course Name: Digital Signal Processing Design

Course Number and Section: **14:332:447:01**

STFT and Phase Vocoder Time and Pitch Scale Modification

Submitted by: Lance Darragh

1. Introduction

Last time, we talked about the overlap-save and overlap-add methods of fast convolution, and how they depended on the theory of the DTFT, and computationally, the FFT. Here we investigate another application of the DTFT/FFT, the short-time-Fourier-transform (STFT), as well as a couple of its applications, time scaling and pitch shifting. The methods used in the STFT are very similar to what we discussed in the last report. The input is broken up into overlapping blocks of length N, and the length-N FFT will be taken of each block, just like in the overlap-save method. However, in the STFT, the blocks are windowed before taking the FFTs. This alters the signal. So, the reconstruction process has to take this into account. To do so, the inverse-short-time-Fourier-transform (ISTFT) windows the blocks again and recombines them in the same manner as in the overlap-add method. We'll explain further.

2. Short-time Fourier Transform (STFT)

The STFT is defined as first dividing the input signal into M + 1 overlapping blocks, each block being of length N, zero padding the last block if necessary. The portion of each block that does not overlap with the next block is referred to as the hop-size, and is denoted by R. The overlap of the blocks will then be N-R. Then, each block is windowed with a particular type of window that satisfies two important properties (to be described shortly), and the length-N DTFT of the each block is taken, resulting in the STFT. An illustration, taken from [0], is shown below.

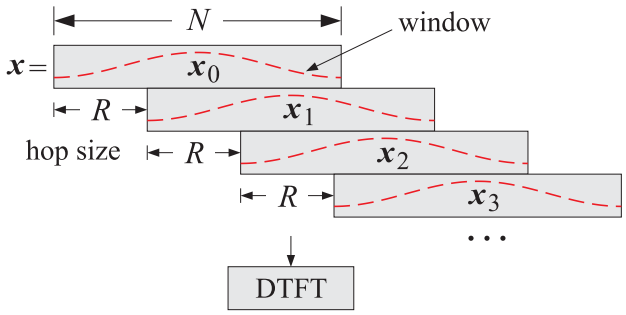

Fig. 1: Illustration of the STFT

This can be expressed mathematically by letting m be an integer from 0 to M that denotes each block, letting w(n) denote the window of length N, and applying the definition of the DTFT of the windowed signal:

(STFT)

(1) $$X_{k,m} = \sum_{n=0}^{N-1} x(mR+n)w(n)e^{-j\omega_k n}$$ k = 0, 1, ..., N - 1 (Frequency-domain Index)
m = 0, 1, ..., M (Time-domain Index)

Recall that the DTFT is the DFT evaluated at N points along the frequency axis, $\omega_k = 2\pi k/N$. The double subscripts indicate that the STFT is actually a two-dimensional, time-frequency representation of the signal. This means that the STFT allows us to make changes to the signal in both the time and frequency domains. We'll explore some of these features in the Applications section.

If we were to try to reconstruct this signal directly, the ISTFT would read,

(2) $$x(mR+n)w(n) = \frac{1}{N}\sum_{k=0}^{N-1} X_{k,m}e^{j\omega_k n}$$ n = 0, 1, ..., N-1
m = 0, 1, ..., M

Solving for the reconstruction of each block,

(3) $$x(mR+n) = x_m(n) = \frac{1}{Nw(n)}\sum_{k=0}^{N-1} X_{k,m}e^{j\omega_k n}$$ n = 0, 1, ..., N-1
m = 0, 1, ..., M

Where we let $x_m(n)$ denote a single block of the input signal. However, the values of w(n) may be close to zero, resulting in an extremely large, and inaccurate, value in the reconstructed signal. So, we use the overlap-add method instead.

(4) $$y(n) = \sum_{m=-\infty}^{\infty} x_m(n-mR)$$ (ISTFT, OLA Reconstruction)

3. STFT Window Properties

One may ask when we can reconstruct the exact signal from the STFT of the windowed version. We can do this by noting that y(n) = x(n)$\underline{w}$(n), where $\underline{w}$(n) is the overlapped-added version of the window.

(5) $$\underline{w}(n) = \sum_{m=-\infty}^{\infty} w(n-mR)$$

This gives us,

(6) $$y(n) = x(n)\underline{w}(n) = \sum_{m=-\infty}^{\infty} x_m(n - mR)$$

and

(7) $$x(n) = \frac{\sum_{m=-\infty}^{\infty} x_m(n - mR)}{\sum_{m=-\infty}^{\infty} w(n - mR)}$$

However, in practice we do not usually solve for x(n) exactly, but rather settle for a signal that is equal to x(n) up to a constant. To explain, a practical window is not ideal, but rather has a main lobe and side lobes in the frequency domain. Often, the window is chosen so that its Fourier transform has a narrow passband in frequency for its main lobe and has small side lobes. Recall that multiplication of the window in the time domain is equivalent to the convolution of the signal in the frequency domain. So, having a narrow passband allows the convolution to "sift" out more precisely the frequency components of the signal as the DTFT sum is taking place, but any side lobes of the window will cause spectral leakage to occur. So does the overlap of the blocks, but this overlap also helps to avoid the effects of the windows tapering to a low value towards the ends. The relative sizes of the main and side lobes of the window's Fourier transform determine its ability to reject adjacent frequencies when sifting. It is possible to choose the windows, and the hop size R, so that their contributions add up to a constant in the time and frequency domains, which preserves the information of each sample relative to each other. This is known as the constant-overlap-add (COLA) property. The following figure showing this type of overlap of Hamming windows is from [3].

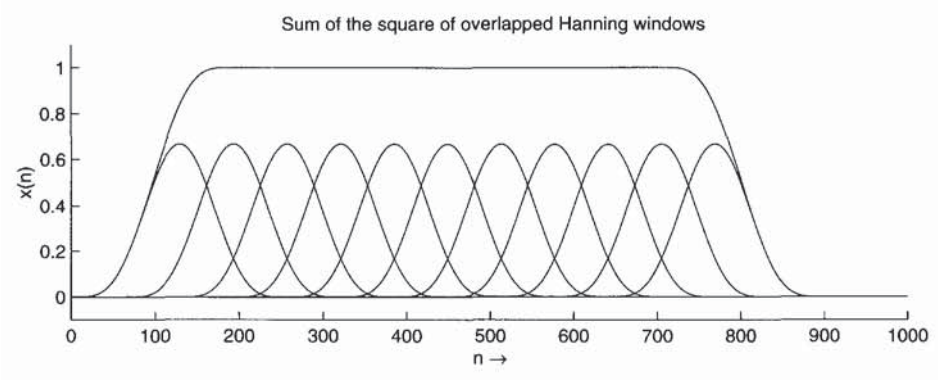

Fig. 2: Illustration of the COLA property

Since the window is periodic with period R, we can describe it in terms of an R-point discrete Fourier series.

(7) $$\underline{w}(n) = \sum_{m=-\infty}^{\infty} w(n - mR) = \frac{1}{R} \sum_{r=0}^{R-1} W(\omega_r) e^{j\omega_r n}, \qquad \omega_r = \frac{2\pi r}{R}$$

where

(8) $$W(\omega_r) = \sum_{n=0}^{N-1} w(n) e^{-j\omega_r n}$$

So, if we choose the window such that $W(\omega_r) = 0$ for r = 1, 2, ..., R - 1, we obtain a single value, corresponding to the r = 0 in equation (7), or $\omega_r = 0$ in equation (8), giving us a constant value for $\underline{w}(n)$, W(0)/R. We also require that $w(0) \neq 0$ so that there is no division by zero in equation (3).

4. Applications: The Phase Vocoder and Time and Pitch Scale Modifications

As mentioned earlier, the STFT can be used to apply time and frequency based effects to a signal. When doing this, we generalize our STFT and ISTFT definitions to have different hop sizes, Ra (analysis hop size) for the STFT, and Rs (synthesis hop size) for the ISTFT, with our time and/or frequency based signal processing occurring in between. The more general system is shown in the figure below, taken from [0].

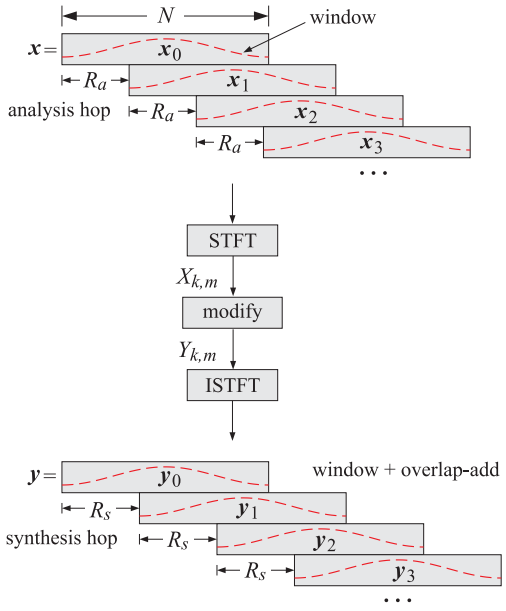

Fig. 3: An STFT Signal Processing System

One of the things this more general definition gives us the ability to do is change the duration of a signal, to stretch or compress it in time. Say for example that a musical piece was recorded for a film, but its length was a few seconds too long or too short to match the scene it was recorded for. Instead of rerecording the entire piece, an audio editor can compress or expand the signal in time to save the recording and make it work with the scene. By letting Ra be greater than Rs, the signal is played back faster in time, and conversely, if Ra < Rs, the signal will be played back at a slower speed. By taking the ratio of these two parameters, we define the speed-up factor, r.

(10) $$r = \frac{Ra}{Rs}$$ (speed-up factor)

If you've ever heard a tape or an album played faster or slower than the recorded speed, you've heard that doing so alters the frequency content of the signal. Playing the recording slower results in a time-stretched signal, but at a lower pitch, and playing the recording faster results in a time-compressed signal, but at a higher pitch, the familiar "chipmunk effect". There is a way to alter the duration of a signal without perceptively changing its frequency content, and it's done with what's known as a phase vocoder.

Within each block of the STFT, the phase information of the signal is essentially our information of when the frequencies present in that block occur relative to each other, within one period. This is our source of frequency-timing information. It is constrained within each block of the STFT, and we can access it by the index variable k within our N-point STFT blocks. We can construct a matrix consisting of the M + 1 STFT blocks as columns, accessing each block with the index m, where $1 \leq m \leq M + 1$. Each block has an overlap N-R, and the matrix can be constructed with the help of MATLAB's buffer function. Because we have some overlap between the blocks, we actually have some of the same phase information in adjacent blocks. What we need to do is map the frequency-timing information present in the STFT of the input signal, as it is broken down into blocks with a hop size Ra, to fit correctly with the absolute timing information when we recombine the blocks as they are put back together with a hop size Rs.

The STFT of the input signal x(n) can be written in the form

(9) $$X_{k,m} = \left|X_{k,m}\right| e^{j\Phi_{k,m}}$$

and the output can be written in the form

(10) $$Y_{k,m} = \left|Y_{k,m}\right| e^{j\Psi_{k,m}}$$

For the most straightforward version of the phase vocoder, the magnitudes of the samples are not changed.

(11) $$\left|X_{k,m}\right| = \left|Y_{k,m}\right|$$

For each frequency, $\omega_k = 2\pi k/N$, we cycle through all of the STFT blocks and compute the phase difference (mod 2π) between two adjacent blocks, which is inherently a function of the analysis hop size.

(12) $$\Delta\omega_{k,m} = \frac{1}{R_a}\left[\Phi_{k,m} - \Phi_{k,m-1} - R_a\omega_k\right]_{mod2\pi}$$

This is the average change in phase per change in time equal to the analysis hop size Ra. To determine the instantaneous frequency, we add the frequency being analyzed to it's change in frequency.

(13) $$\omega_{k,m} = \omega_k + \Delta\omega_{k,m}$$

Then we multiply this by Rs, and the product becomes the difference in phase of this particular frequency between two adjacent blocks of the output samples.

(13) $$R_s\omega_{k,m} = \Psi_{k,m} - \Psi_{k,m-1}$$

Solving for $\mathbf{\Psi}_{k,m}$ we have a recursive formula for the phase value for each frequency ω_k in each block, now time-aligned according to our synthesis hop size Rs.

(14) $$\Psi_{k,m} = \Psi_{k,m-1} + R_s\omega_{k,m}$$

Since this is a recursive function, the first phase value in each block is taken to be that of the original STFT. The previous steps have been put into a function in MATLAB code called phmap (phase map) and can be found in Appendix A.2. To stretch or compress a signal in time without changing its pitch, we decide on our speed-up factor r, analysis hop size Ra, and FFT block size N, run the algorithm for the STFT, which can be found in Appendix A.1, then run the output through phmap and then run the algorithm for the ISTFT, which can be found in is in Appendix A.3. These steps have been put together into a function called phvoc (phase vocoder), which can is in Appendix A.4. The synthesis hop size is calculated internally by

(15) Rs = round(Ra/r)

Now that we have the ability to stretch or compress a signal in time without changing its pitch, we can combine this with a resampling operation to increase or decrease the pitch of a signal without changing its duration. What we'll be doing is changing the length of the signal, which changes the pitch, but using the phase vocoder with time modification to restore the original length of the signal, preserving the change in pitch. First we'll look at increasing the pitch.

If we have a signal x(n) with a sampling rate f_s, and we want to increase the pitch, we can play it back at a rate of rf_s, with $r > 1$, but this will be shorter in duration by a factor of 1/r. So, we can resample the signal at f_s/r, extending its length by 1/r, and then play it back at a rate of rf_s. This will be the same signal with a pitch r times higher. To illustrate, let's take a closer look at the resampling operation.

In the digital world we have to account for each sample of the output when it's sped up or slowed down. In the case where we extend the signal, to slow it down, we first pass the signal through an upsampler that increases the number of samples per second by simply adding zeros between each sample of the input. An upsampler that increases the sampling rate by a factor of L inserts L - 1 zeros between each sample of the input. Then, an interpolation filter, sometimes called an oversampling digital filter, is used to change the zeros to values that match more closely the samples of the input signal. An illustration taken from [1] is shown below for an upsampler with L = 4.

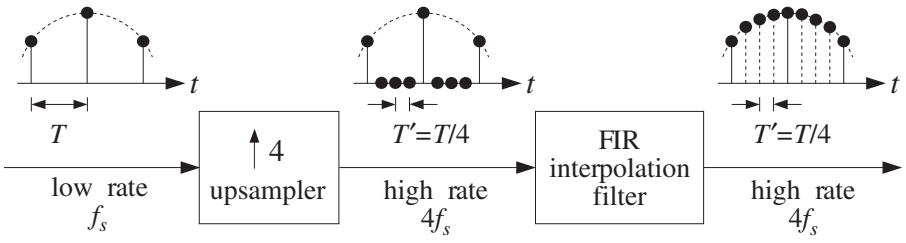

Fig. 4: Upsampling by a Factor of 4

This combined system is known as an interpolator. You may ask how we fill in these values. Well the answer is that there are a lot of different interpolation methods, some being better suited for specific applications, but the two most commonly used, and straightforward methods, are hold interpolation and linear interpolation. Hold interpolation fills in the L-1 zeros with the value of the sample that preceded them, effectively repeating the sample L times. A linear interpolation filter calculates the values so that there is a straight line in the graph of the time versus sample values.

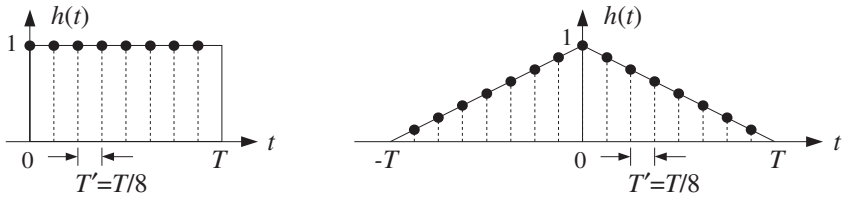

Fig 5: Hold and Linear Interpolators with L = 8, from [1]

The hold interpolator has an input/output relationship:

(14) $$y_{up}(nL+i) = x(n), \qquad i = 0, 1, ..., L-1$$

The linear interpolator uses a series of linear combinations of the two input signals, each weighted differently depending its relative location to the two samples. It has the input/output relationship:

(15) $$y_{up}(nL+i) = \left(1 - \tfrac{i}{L}\right) x(n) + \frac{i}{L} x(n+1), \qquad i = 0, 1, ..., L-1$$

An example for L = 4, taken from [1], is shown below. Say we want to calculate the L - 1 = 3 values between samples C and D; we'll call them samples X, Y, and Z

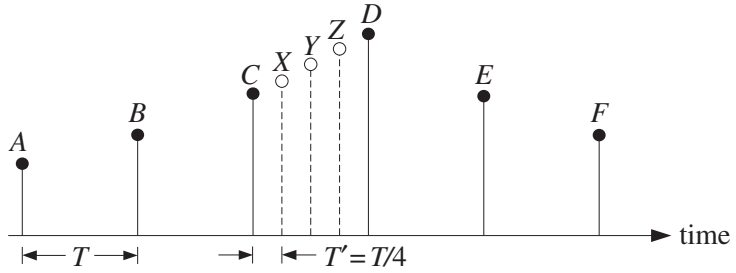

Fig. 6: Linear Interpolation Example, L = 4

To calculate these values, we use the following linear combinations of the samples C and D,

(16)
X = 0.75C + 0.25D
Y = 0.50C + 0.50D
Z = 0.25C + 0.75D

This takes care of upsampling in the time domain, but we also have to consider how these changes affect the frequency domain. Continuing with the same example from [1], we show the frequency spectrum of the original signal, sampled at the low rate f_s. Assuming that the signal was bandlimited to $f_s/2$ before sampling, its nonoverlapping spectral images are placed at multiples of f_s.

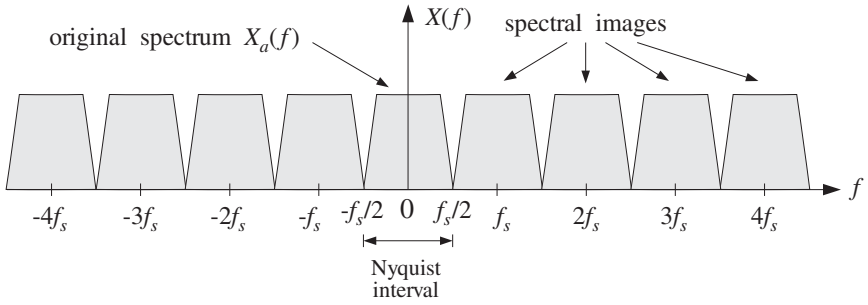

Fig: 7: Spectrum of Signal Sampled at Original Rate f_s, from [1]

Recall from sampling theory that the passband gain of the interpolation filter must be equal to L. These additional spectral images are normally removed by an analog anti-image lowpass postfilter, with a cutoff frequency of $f_s/2$, when being reconstructed into an analog signal, but they still exist while we're in the digital domain. When we increase the sampling rate to $4f_s$, which we'll call f_s', from the perspective of the new sampling rate we've moved the Nyquist interval to $f_s'/2$ with spectral images repeated at multiples of f_s'.

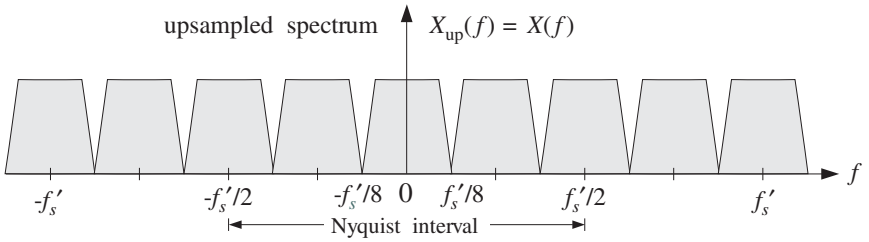

Fig: 8: Spectrum of Signal Upsampled at the Rate $f_s' = 4f_s$, from [1]

When upsampling by a factor of L, our original Nyquist interval now lies in the range $|f| \leq f_s'/(2L) = f_s'/8$, and we now have L - 1 replicas of the original spectrum between multiples of the new sampling rate f_s'. A digital FIR filter operating at the new rate f_s', with a cutoff at $f_s'/2$, will remove these

L - 1 spectral images between multiples of f_s', but it cannot remove those that occur at multiples of f_s' because it is a digital filter, which is periodic with period f_s'. So we use an analog postfilter to remove these during analog reconstruction.

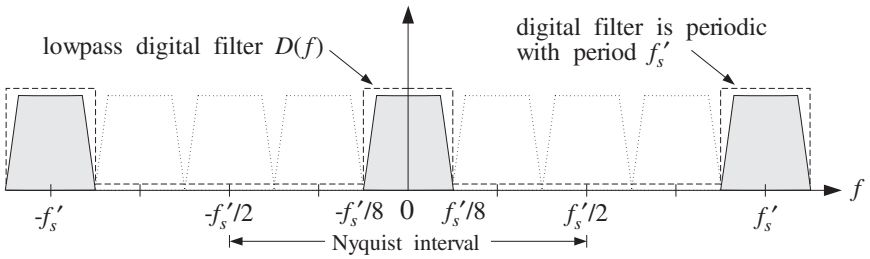

Fig: 8: High-rate digital filter removes intermediate spectral images, from [1]

Now when we reconstruct the signal in the analog domain we can use a less stringent analog anti-image lowpass postfilter. Instead of requiring an analog filter with a very steep cutoff, which must be of high order and therefore be relatively expensive, we can relax the specifications and use a much simpler, less expensive analog filter.

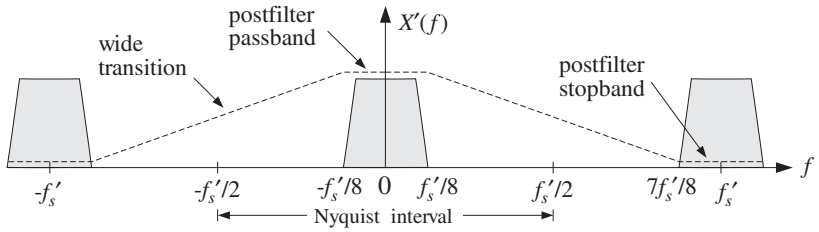

Fig: 9: More relaxed analog anti-imaging postfilter, from [1]

As a matter of fact, sometimes signals are intentionally sampled, or resampled, at a higher rate than needed, just to be able to take advantage of this lower order analog postfilter, and it is often a good design trade-off to take advantage of the sharper cutoff in the digital domain in order to relax the specifications needed in the analog domain.

To increase the pitch of a signal by shortening it in time, we use a process called decimation. It is the inverse process of interpolation. If we were to upsample a signal by a factor of L, inserting L - 1 zeros

between samples and interpolating their values, downsampling the signal by a factor of L would remove these L - 1 samples. If our upsampled signal is at the rate f_s', our downsampled signal will be at at rate $f_s = f_s'/L$. We also move the Nyquist interval from $|f| \leq f_s'/2$ to $|f| \leq f_s'/(2L)$, or $|f| \leq f_s/2$. This downshifts our spectral replicas so that they are closer together.

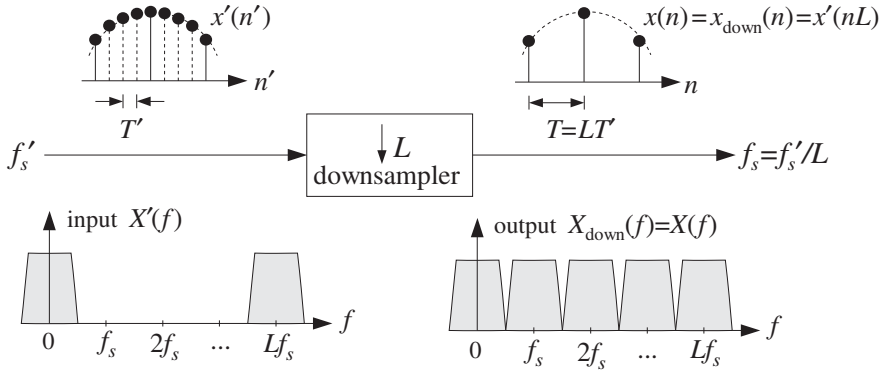

Fig: 10: The effects of downsampling in time and frequency domains, from [1]

In this example, our signal was already bandlimited to $f_s/2$, and we upsampled by the exact same amount we downsampled. More generally, this is not the case, and we need to ensure that our spectral replicas do not overlap each other when they are downshifted as a result of downsampling. We do this by bandlimiting the signal with a digital lowpass prefilter whose passband gain is 1. This is known as a decimation filter. The system of the downsampler followed by the decimation filter is known as a decimator, and a more general illustration is shown below.

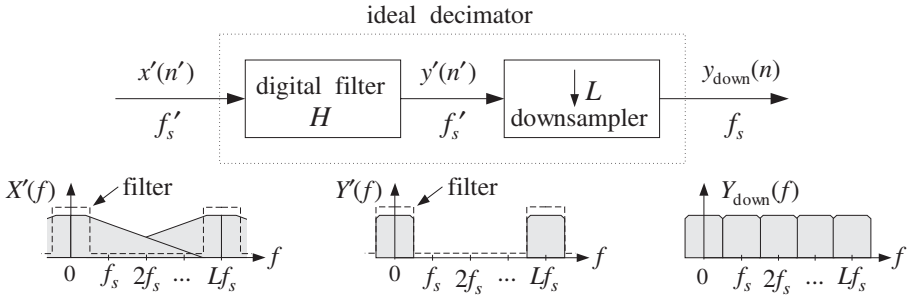

Fig: 11: Ideal digital decimator in the frequency domain, from [1]

We can generalize the combination of interpolation and decimation to a more general type of sampling rate conversion, that which multiplies the sampling frequency by a rational factor. We'll let L denote the upsampling factor and M denote the downsampling factor.

(16) $$f_s' = \frac{L}{M} f_s$$

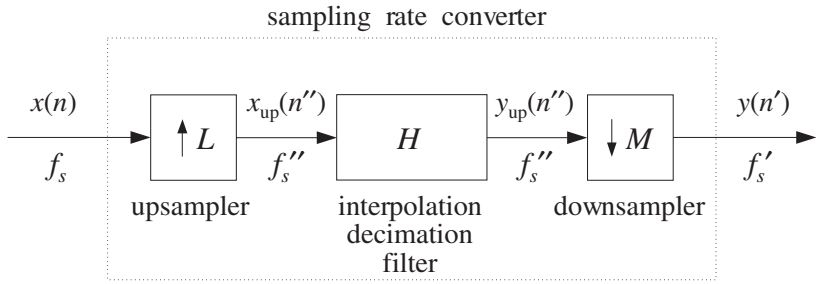

Fig: 12: A more general sampling rate converter, from [1]

Now that we have a more general sampling rate conversion and the tools to stretch and compress signals in time, we can put together our pitch modification system. We let r by our speed up factor in rational form, L/M.

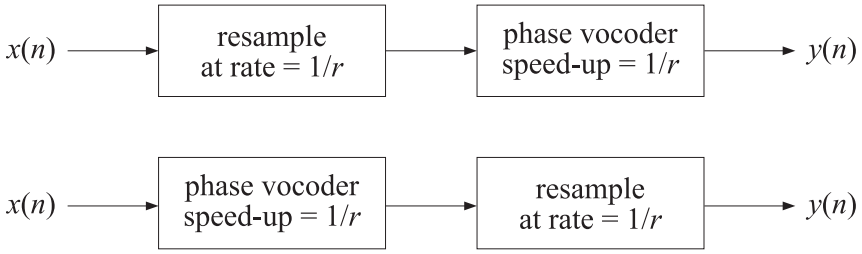

Fig: 13: Pitch shifting by a factor of r, from [1]

We can interchange the order of the operations, but better results are obtained from the top version if r > 1, and the bottom version when r < 1. To do the resampling operation, we'll use MATLAB's built in resample function, which takes as input the numerator and denominator of r. It offers us to use either the linear or hold interpolator, but we'll use the linear interpolator because it has a better frequency response for use in the STFT because it has a narrower main lobe and smaller side lobes.

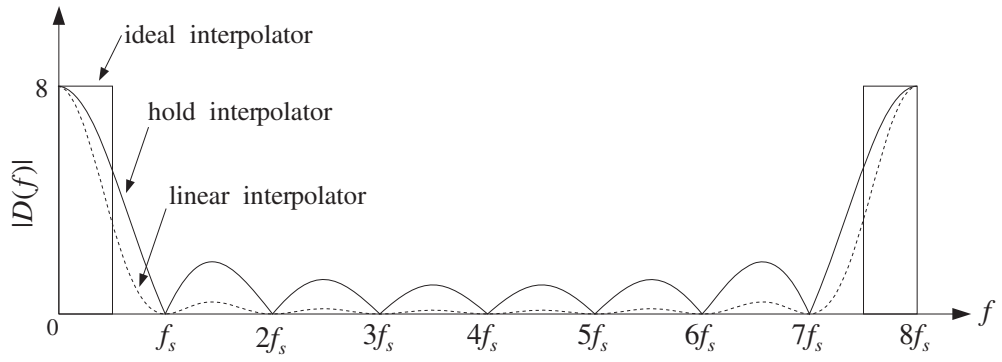

Fig. 14: Frequency Response of Ideal, Hold, and Linear Interpolators with L = 8, from [1]

We'll also make use of MATLAB's rat function, which takes r as a decimal value and outputs an estimate of r in rational form, p/q = L/M. Then, if r >1, we put these values into the resample function and pass the output through the phase vocoder, and if r < 1 we put them through in the opposite order.

The code to modify the pitch of a flute sound up and down by a factor of 2 can be found in Appendix A.5. A few observations about these processes, we note that the audio file being stretched actually loses data when it's slowed down. The number of columns in the matrices, X and Y will have to be modified in order to obtain the entire audio file being slowed down. This can be done with a conditional statement inside the stft function, zero padding the input signal to be the length of the final output signal if r < 1.

When expanding a signal in time, there's always an upper limit on how long we can stretch it before the effect becomes obviously unnatural, even when using a phase vocoder. This is based on the fact that the phase vocoder uses a type of averaging technique to transfer the phase information between time scales.

When trying different values of Ra and N, there is a noticeable difference in the quality of the output signal when changing the STFT-block size. The larger the value of the block size the better the results. Settling for a value to experiment with by keeping N fixed at 4096 and trying different analysis hop sizes, we can hear a practical upper limit on the ratio of the hop size to the window length. At a ratio of 1/2 (Ra = 2048), we hear noticeable chopping sounds in the output. At the other extreme, anything with a ratio of less than 1/32 (Ra = 128) can take impractically long to compute due to the large number of FFTs taken (M ≈ (length of input)/Ra), and the output sounds granulated too. Using higher values of Ra also resulted in an output that sounded a bit more granulated. So the outputted signals are computed using

Ra = 256. This resulting in a ratio that seemed to yeild the best results. More information on granulation can be found in [3].

These effects can be understood better by analyzing the frequency/time resolution tradeoff of the STFT. The distance between distinguishable frequencies in the transform domain is $2\pi/N$. So, the larger the windowed DTFT blocks (of size N), the better the frequency resolution, but each block is a snapshot across a window of time, and it will not give us information about how the frequency components of the signal change with time. By breaking the signal into blocks, we are able to get some resolution of how often the frequencies present in the signal change with time, which is why the STFT is also sometimes referred to as the time-dependent Fourier transform. The more frequently we process blocks the better resolution we get about how the frequencies in the signal change with time, but this means we have to take a smaller block size, compromising our frequency resolution, and conversely, the larger the block size the worse the time resolution.

To see these pitch modifications graphically, let's take an audio file consisting of a flute playing a single note. First, we'll plot the magnitude of its frequency spectrum.

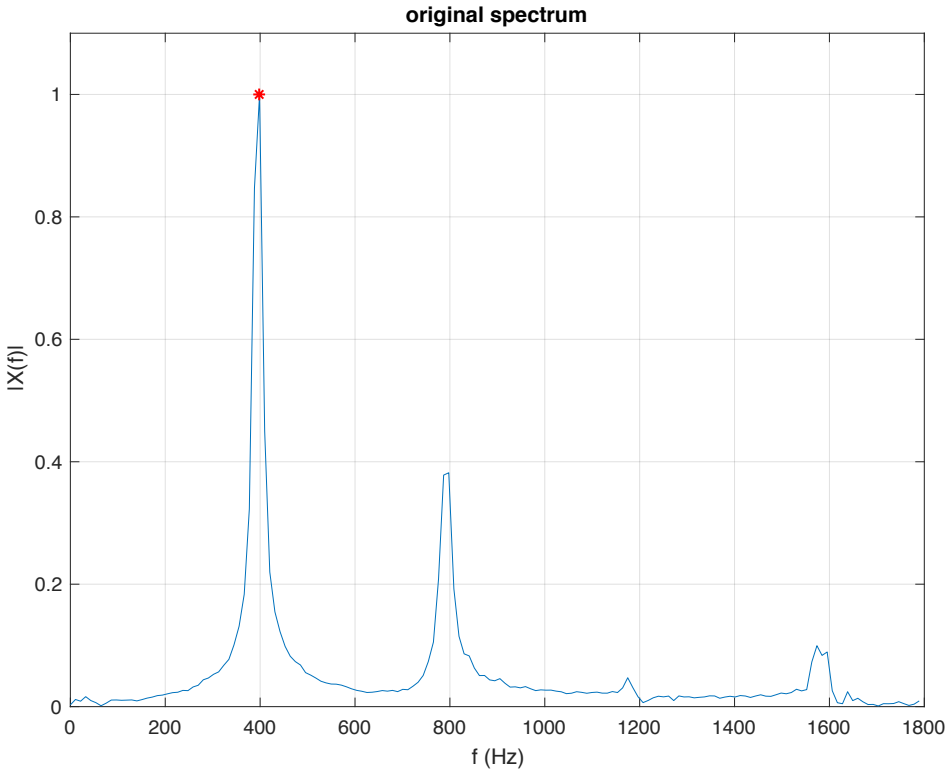

Fig. 15: Frequency spectrum of a flute playing a single note

Then, we'll use our STFT algorithm to create a spectrogram, a plot of the time-frequency information with time on the horizontal axis, frequency on the vertical axis, and the magnitude of the frequency components, in dB, depicted by color.

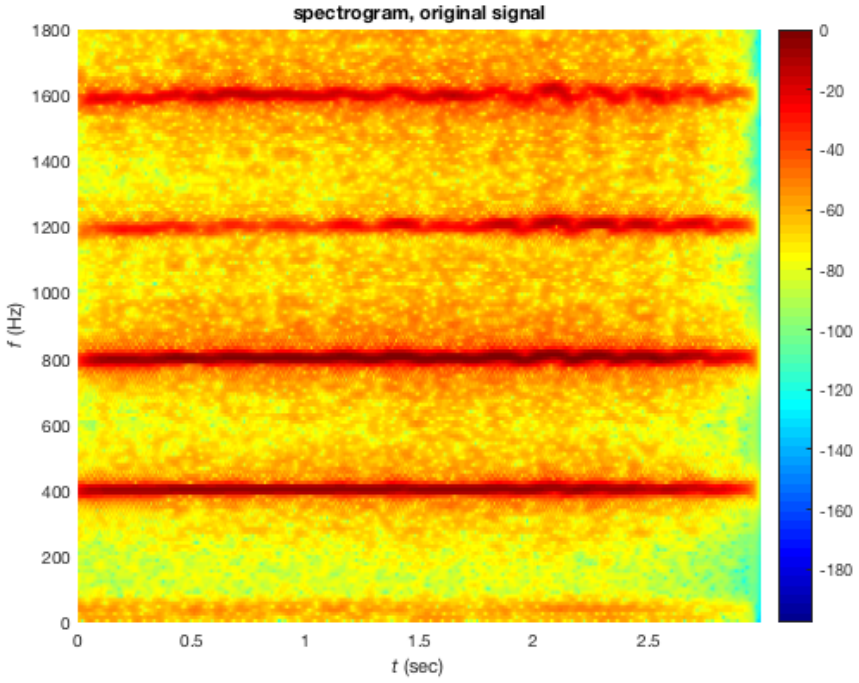

Fig. 16: Spectrogram of a flute playing a single note

The darker red spots on the spectrogram correspond to the peaks in the plot of the frequency spectrum. What we see is that there are dominant frequencies at around 400 Hz, 800 Hz, 1200 Hz, and 1600 Hz. Knowledge of the harmonic series tells us that a musical note, or practically any sound produced by a vibrating body, consists of a fundamental frequency and a harmonic series, frequencies that are integer multiples of the fundamental frequency, each being of a lesser magnitude than the fundamental frequency. What we see here is a fundamental frequency around 400 Hz. This could correspond to a G_4, which is 392 Hz, or a G_4#, which is 415.30 Hz. Looking at the higher frequency harmonics, we can see that they are slightly below 1200 Hz and 1600 Hz, indicating that our fundamental frequency is slightly below 400 Hz, so our note is a G_4.

Now we'll use our pitch shifting algorithm to scale the frequencies by a factor of two, which will scale our musical note by one octave, and graph its frequency spectrum plot and spectrogram. The code for running these signals through the pitch modification and creating the spectrograms can be found in Appendix A.6

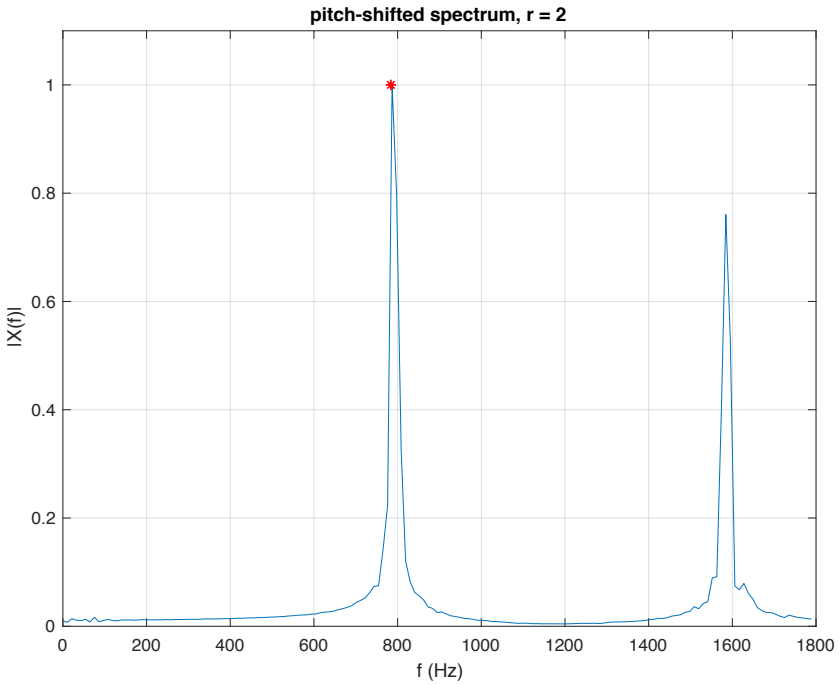

Fig. 17: Frequency spectrum of pitch-shifted signal, r = 2

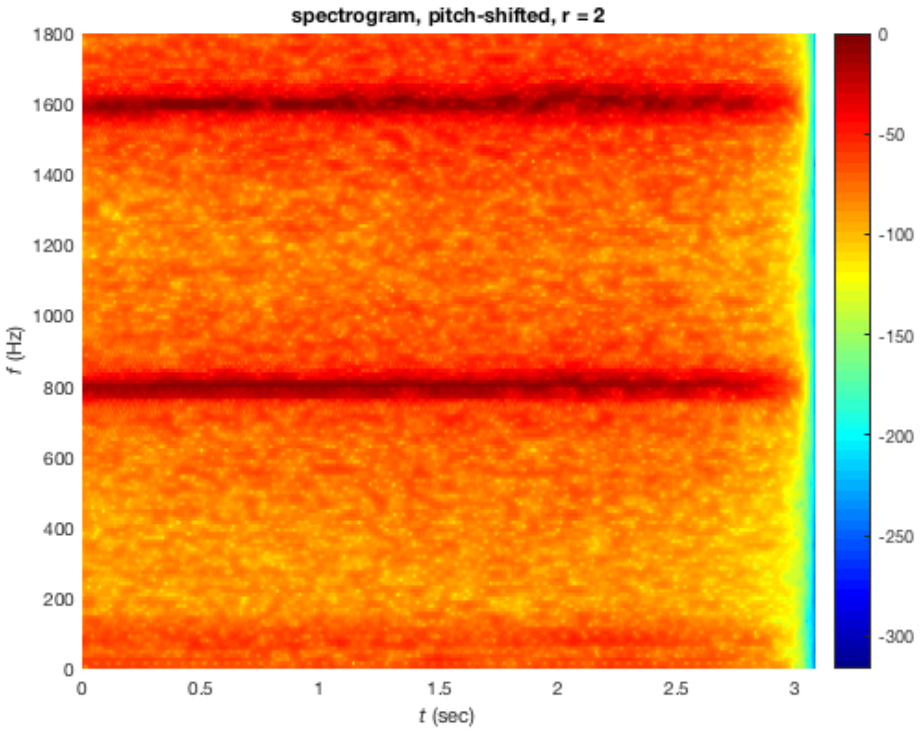

Fig. 17: Spectrogram of pitch-shifted signal, r = 2

We can see that the fundamental frequency has been moved to twice the original frequency, and its harmonics have been scaled by the same factor. This corresponds to a G_5 in our musical system. It's still a G, just one octave higher. Similarly, we can play this note one octave lower by using a speed-up factor of r = 1/2. Doing so results in the following graphs.

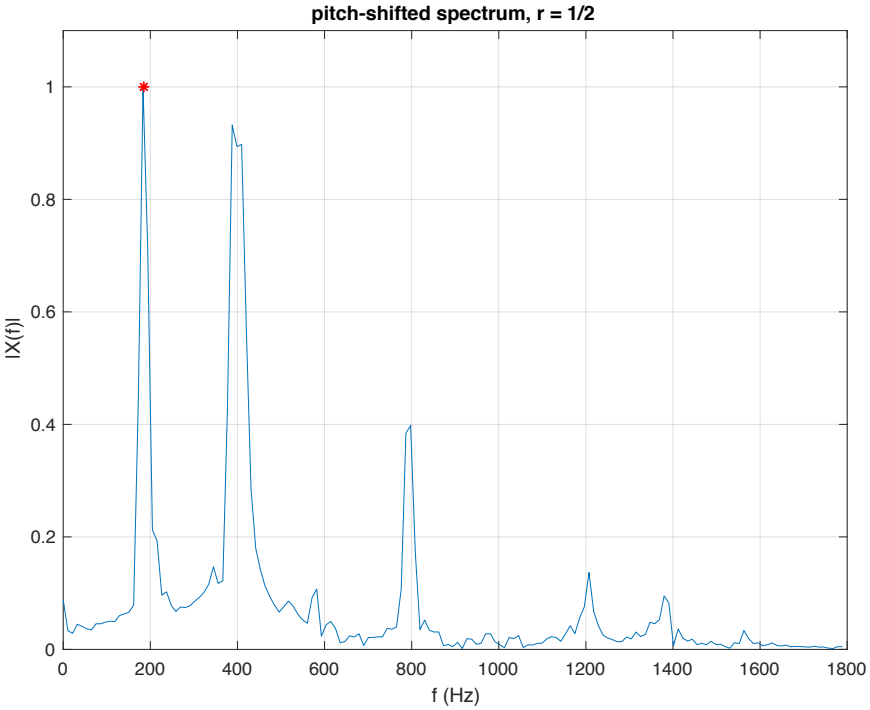

Fig. 18: Frequency spectrum of pitch-shifted signal, r = 1/2

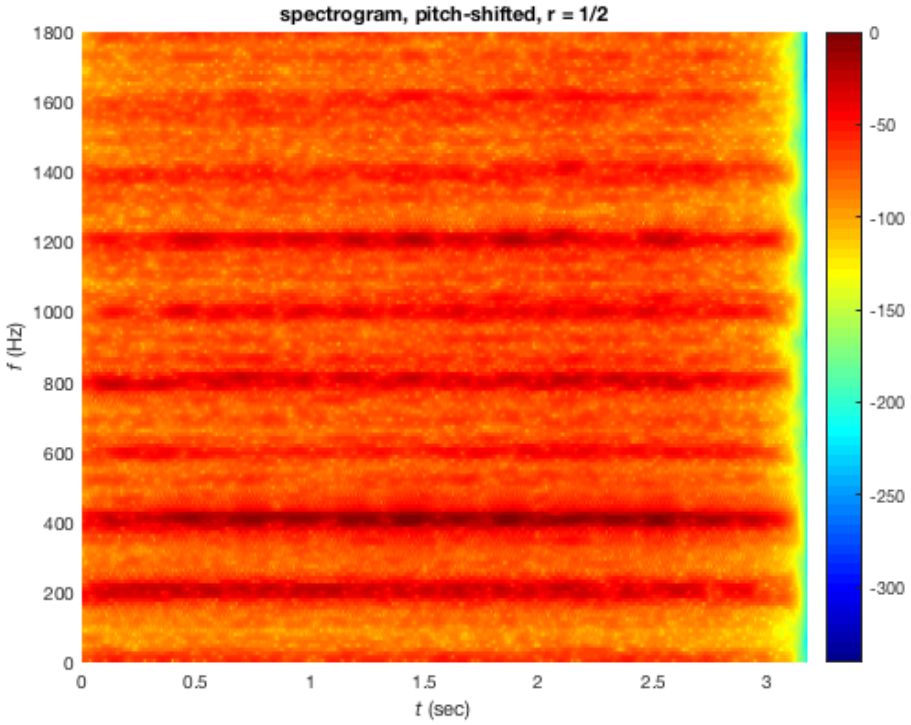

Fig. 19: Spectrogram of pitch-shifted signal, r = 1/2

Note that these are multiplicative changes in frequency, not translations in frequency, which are additive shifts in frequency. Musical transposition is actually multiplicative, as the notes that make up our musical scales are derived from the harmonic (overtone) series, which themselves were derived by Pythagoras by using strings of different lengths that were multiplicative rational factors of each other. A musical transposition by a semitone can be accomplished by using a speed-up factor of the form

(17) $r = 2^{(k/12)}$, where k is any integer.

The useful values of k are restricted by the range of human hearing, which is about 20Hz to 20 kHz, so the bounds of useful values of k will be different depending on what frequency you start with.

In order to better visualize what's happening at the sample level, lets take a four millisecond clip of this flute's note and repeat the steps above.

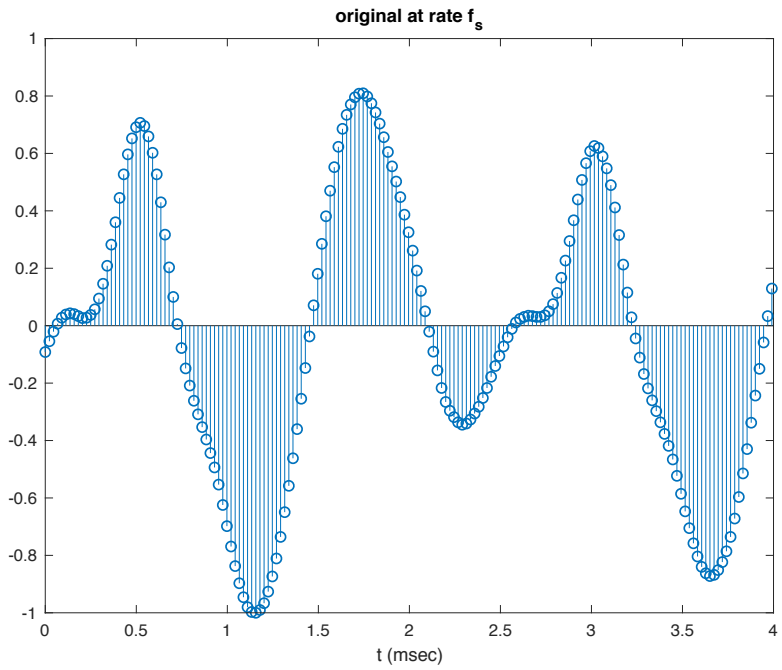

Fig. 20: Four millisecond clip of the original signal

Now, we'll resample it at a rate of f_s/r, with r = 2

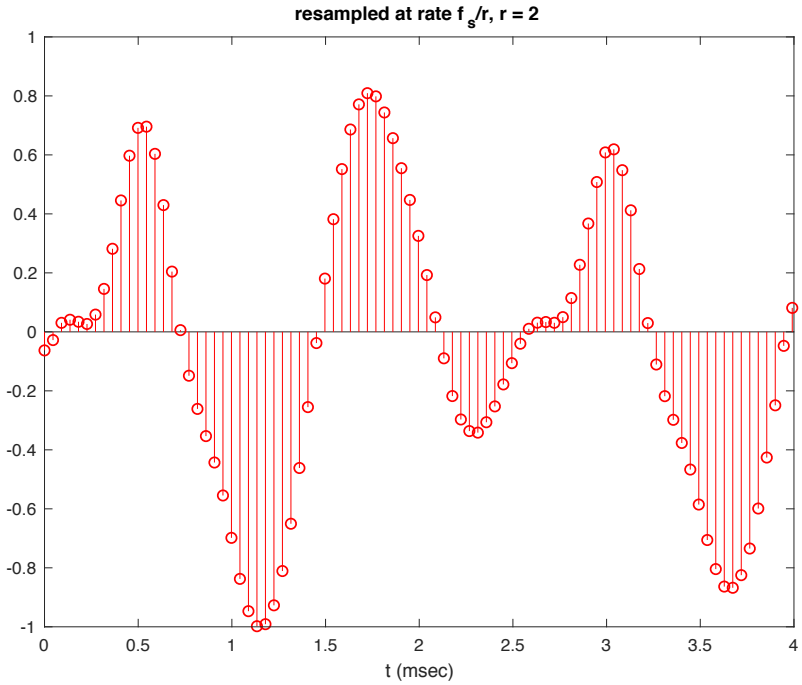

Fig. 21: Four millisecond clip of the resampled signal, f_s/r, with r = 2

Notice that it's the same signal as the original but with every other sample removed. Now, we'll replay it at the normal sampling rate f_s.

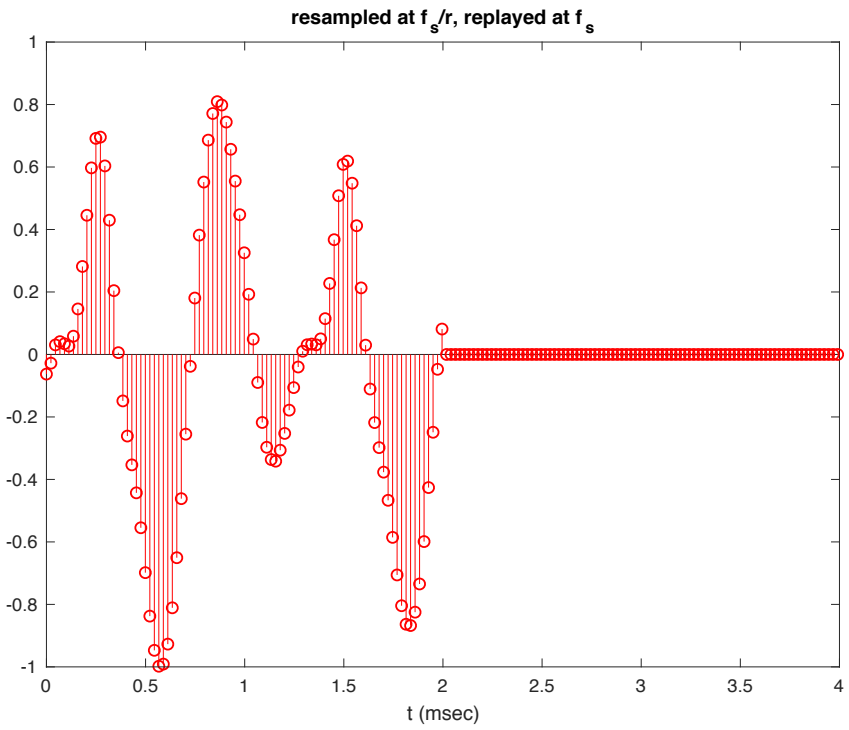

Fig. 21: Four millisecond clip of the resampled signal, f_s/r, with r = 2, played back at the rate f_s

This is the familiar "chipmunk" effect, where we here the same sound, but at a higher pitch, due solely to it bing played back faster. Now, we'll stretch it back out in time with our phase vocoder.

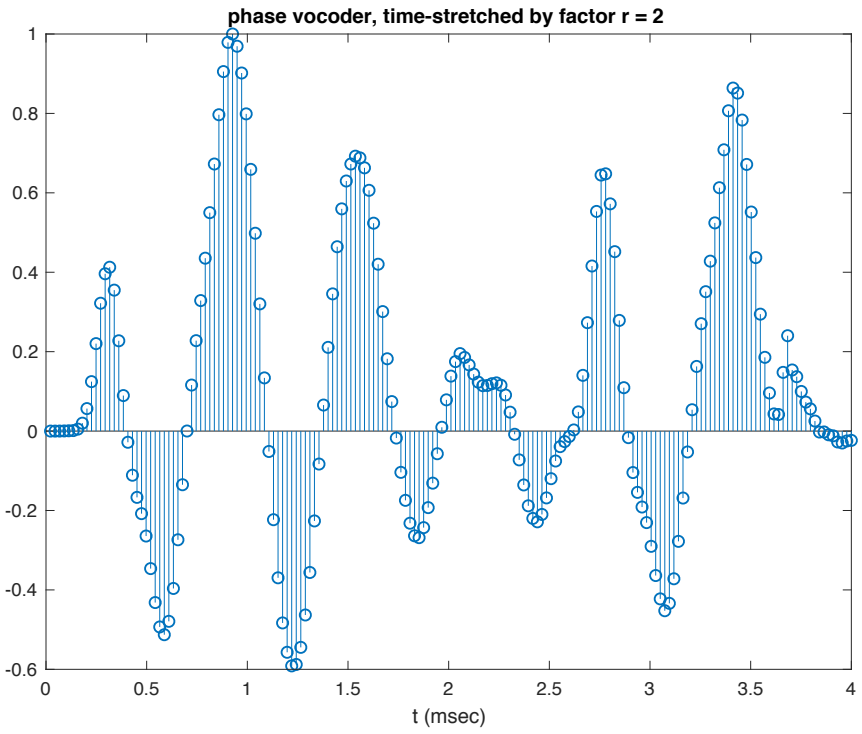

Fig. 22: Signal resampled at f_s/r and stretched by a factor of r, with r = 2

We can see from the graph that the duration of the signal is the same as that of the original, and the frequency of oscillation is twice as high. (Note to Professor: The plot is different from that in the resources. I've spent a good bit of time trying to find the source of the error, but I'm not able to find why there's a difference in these plots. All of the audio signals, frequency plots, and spectrograms sound/look correct to me. If you have time, and only if you have time, please let me know what the error is because I'd like to be solid on what's going on in the phase vocoder.)

Finally, let's look closer at the COLA property of the window. We'll use a hanning window with length N = 128 and its overlapped version, $\underline{w}(n)$, with M = 10.

(18) $$\underline{w}(n) = \sum_{m=0}^{10} w(n - mR)$$

Where the hanning window, w(n) is defined as

(19) $$w(n) = 0.5 - 0.5cos(\frac{2\pi n}{N-1}), \qquad n = 0, 1, ..., N-1$$

We'll consider four values of the hop size, R = N/2, 3N/8, N/3, and N/4. For each, we'll plot the magnitudes in both the time and frequency domains and see which have constant values and which don't. The magnitudes in the frequency domain will be normalized to unity.

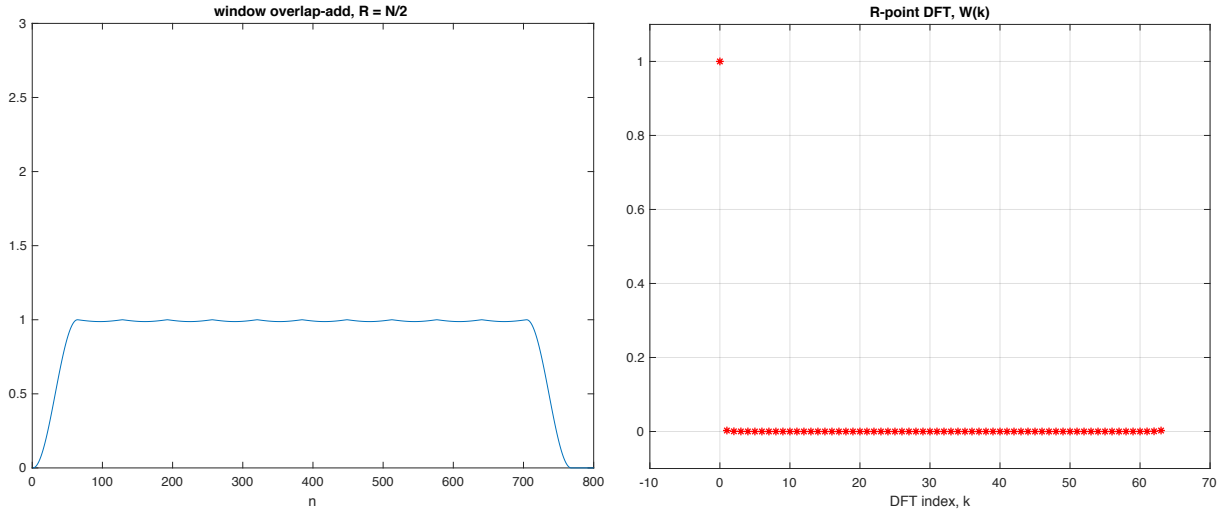

Fig. 23: COLA property satisfied for hanning window with R = N/2

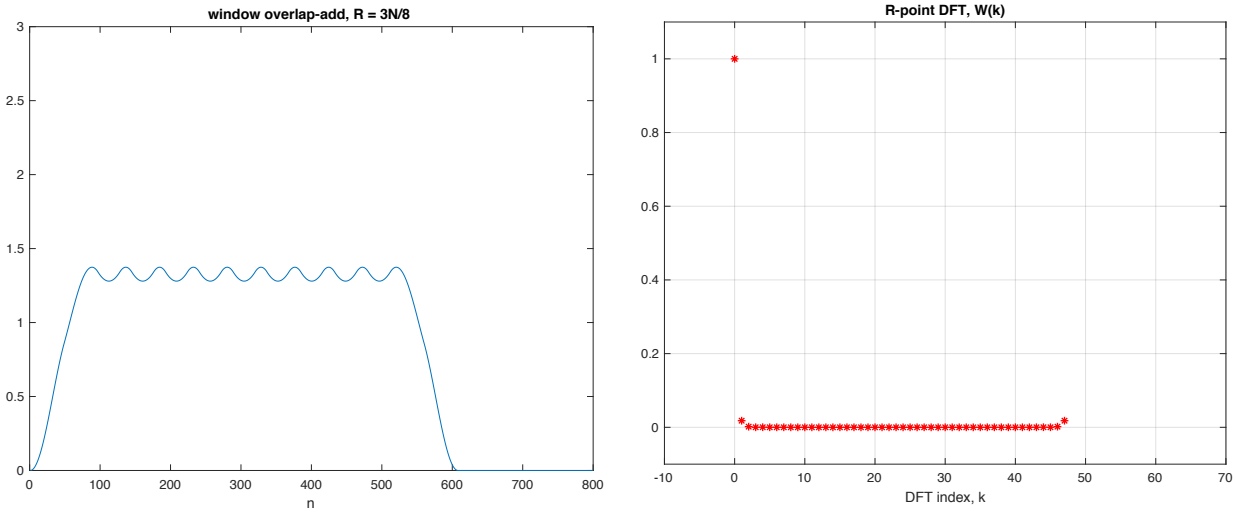

Fig. 24: COLA property (practically) satisfied for hanning window with R = 3N/8

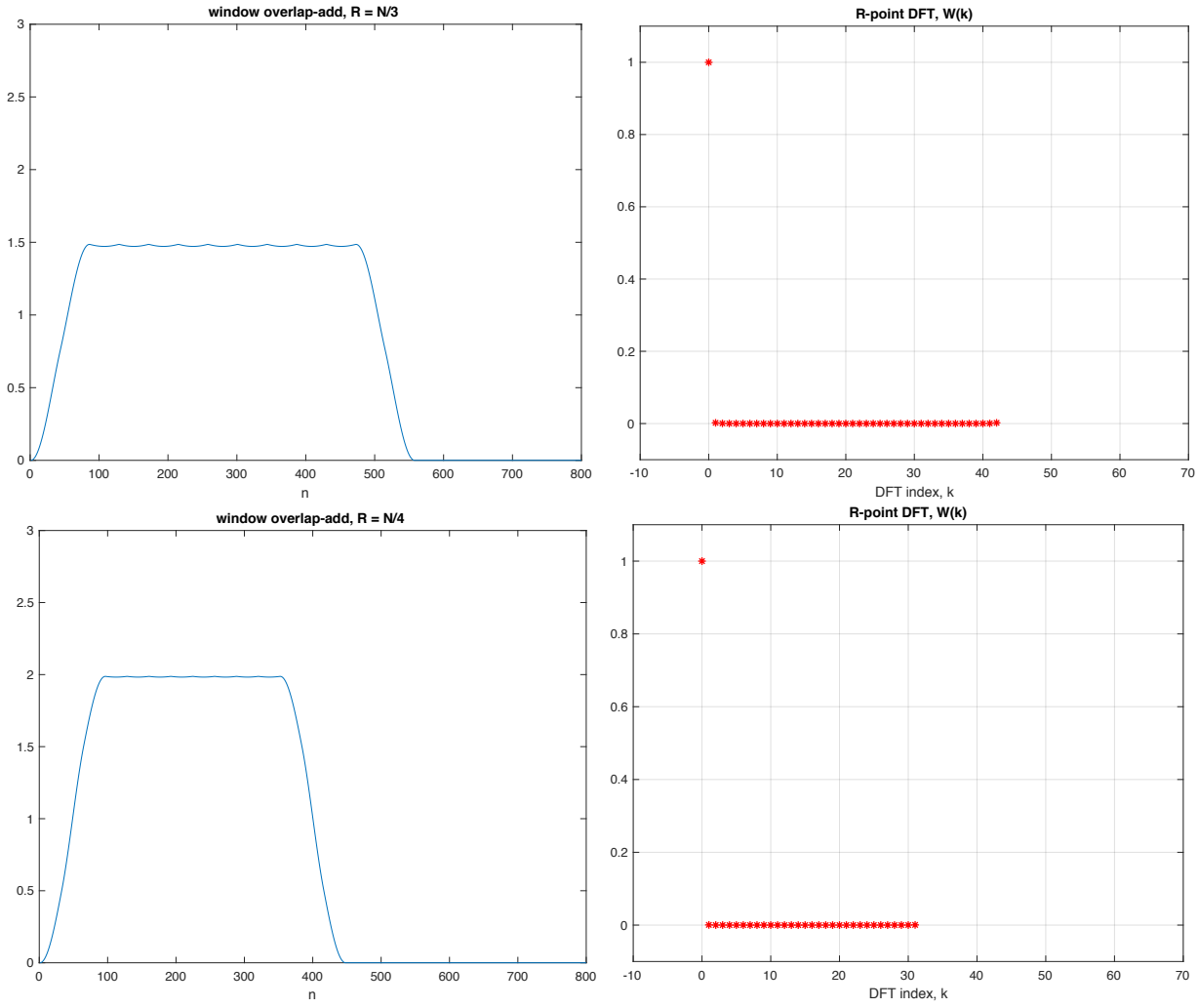

Fig. 25: COLA property satisfied for hanning window with R = N/3

F**ig. 26**: COLA property satisfied for hanning window with R = N/4

We note that in all of these examples, the condition that $w(0) \neq 0$ is satisfied.

5. References

[0] Sophocles J. Orfanidis, DSP Design Project Resources, Rutgers University, 2019.

[1] Sophocles J. Orfanidis, Introduction to Signal Processing, Rutgers University, 2010.

[2] A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing, 3/e, Pearson, Upper Saddle River, NJ, 2010.

[3] Udo Z¨olzer, ed., DAFX – Dig

Table of Contents

1. Phase Vocoder (phvoc function)

```
% Import original audio
[x, fs] = audioread('04 Finger Eleven - One Thing.mp3', [44100*25,
 44100*35 - 1]); % 10 sec audio sample
x = x(:,1); % as a column

% Listen to original audio
soundsc(x,fs)
```

1.a Phase Vocoder, r > 1

```
% speed up factor
r = 1.4;
% analysis hop size
Ra = 256;
% STFT block size
N = 4096;
% Compute output
y = phvoc(x,r,Ra,N);
% Normalize to prevent clipping
y = y/max(abs(y));

% Listen to results
```

```
soundsc(y,fs);

% Output Results
audiowrite('phvoc_r_gt_1.wav', [x', zeros(1,fs), y'],fs);
```

1.b Phase Vocoder, r < 1

```
% speed up factor
r = 0.7;
% analysis hop size
Ra = 256;
% STFT block size
N = 4096;
% Compute output
y = phvoc(x,r,Ra,N);
% Normalize to prevent clipping
y = y/max(abs(y));

% Listen to results
soundsc(y,fs);

% Output Results
audiowrite('phvoc_r_lt_1.wav', [x', zeros(1,fs),y'],fs);
```

2. Pitch Modification (pitchmod function)

```
[x,fs] = audioread('flute2.wav');
x = x(:,1);

% Listen to original audio
soundsc(x,fs)
```

2.I Spectrum Plot, Original Signal

```
N = 4096;
Xmag = abs(fft(x,N));                % Take the FFT of x
Xmag = fftshift(Xmag'/max(Xmag));    % Normalize and FFT shift
Xmag(1:2048) = [];                   % Only look at positive
 frequencies
fmax = 1800;                         % Only look up to 1800 Hz.
kmax = round(fmax*N/fs);             % DTF index at around 1800 Hz
f = (0:1/kmax:(1-1/kmax))*fmax;      % frequencies from 0 to 1800 Hz
Xmag(kmax + 1:end) = [];             % frequencies from 0 to 1800 Hz
close all
plot(f,Xmag,398,1,'r*'); grid on; xlabel('f (Hz)'); ylabel('|X(f)|');
title('original spectrum'); axis([0,fmax,0,1.1])
```

2.II Spectrogram, Original Signal

```
R = 256;        % Hop size
N = 4096;       % FFT length
```

```
X = stft(x,R,N);    % STFT of x
M = size(X,2) - 1;
k = 0:N/2;      % positive-frequency half of Nyquist interval
f = k*fs/N;     % f from 0 to fs/2
t = (0:M)*R/fs; % M + 1 time frames spaced by R*Ts = R/fs

Xmag = abs(X(k+1,:)); % extract positive frequency half
Xmag = Xmag/max(max(Xmag));

S = 20*log10(Xmag);      % convert to dB

close all
surf(t,f,S,'edgecolor','none') % Create spectrogram with time on the
% horizontal axis and frequency on the vertical axis
colormap(jet); colorbar; axis tight; view(0,90); % View from above
xlabel('{\itt} (sec)'); ylabel('{\itf} (Hz)');
ylim([0,1800]);             % Show frequency range from 0 1800 Hz
title('spectrogram, original signal')
```

2.a Pitch Modification, r = 2

speed up factor

```
r = 2;
% analysis hop size
Ra = 256;
% STFT block size
N = 2048;
% compute output
y = pitchmod(x, r, Ra, N);
% normalize results to prevent clipping
y = y/max(abs(y));

% Compare Original and Results
soundsc([x', zeros(1,fs),y'],fs);

% Output Results
audiowrite('pitchmod_r_equals_2.wav', [x', zeros(1,fs),y'],fs);
```

2.a.I Spectrum Plot, r = 2

```
N = 4096;
Ymag = abs(fft(y,N));                % Take the FFT of x
Ymag = fftshift(Ymag'/max(Ymag));    % Normalize and FFT shift
Ymag(1:2048) = [];                   % Only look at positive
 frequencies
fmax = 1800;                         % Only look up to 1800 Hz.
kmax = round(fmax*N/fs);             % DTF index at around 1800 Hz
f = (0:1/kmax:(1-1/kmax))*fmax;      % frequencies from 0 to 1800 Hz
Ymag(kmax + 1:end) = [];             % frequencies from 0 to 1800 Hz
close all
plot(f,Ymag,784,1,'r*'); grid on; xlabel('f (Hz)'); ylabel('|X(f)|');
title('pitch-shifted spectrum, r = 2'); axis([0,fmax,0,1.1])
```

2.a.II Spectrogram, r = 2

```
R = 256;        % Hop size
N = 4096;       % FFT length
Y = stft(y,R,N);    % STFT of y
M = size(Y,2) - 1;
k = 0:N/2;      % positive-frequency half of Nyquist interval
f = k*fs/N;     % f from 0 to fs/2
t = (0:M)*R/fs; % M + 1 time frames spaced by R*Ts = R/fs

Ymag = abs(Y(k+1,:)); % extract positive frequency half
Ymag = Ymag/max(max(Ymag));

S = 20*log10(Ymag);      % convert to dB

close all
surf(t,f,S,'edgecolor','none') % Create spectrogram with time on the
% horizontal axis and frequency on the vertical axis
colormap(jet); colorbar; axis tight; view(0,90); % View from above
xlabel('{\itt} (sec)'); ylabel('{\itf} (Hz)');
ylim([0,1800]);             % Show frequency range from 0 1800 Hz
title('spectrogram, pitch-shifted, r = 2')
```

2.b Pitch Modification, r = 1/2

```
% speed up factor
r = 1/2;
% analysis hop size
Ra = 256;
% STFT block size
N = 2048;
% compute output
y = pitchmod(x, r, Ra, N);
% normalize results to prevent clipping
y = y/max(abs(y));

% Compare Original and Results
soundsc([x', zeros(1,fs),y'],fs);

% Output Results
audiowrite('pitchmod_r_equals_0.5.wav', [x', zeros(1,fs),y'],fs);
```

2.b.I Spectrum Plot, r = 1/2

```
N = 4096;
Ymag = abs(fft(y,N));                % Take the FFT of y
Ymag = fftshift(Ymag'/max(Ymag));    % Normalize and FFT shift
Ymag(1:2048) = [];                   % Only look at positive
 frequencies
fmax = 1800;                         % Only look up to 1800 Hz.
kmax = round(fmax*N/fs);             % DTF index at around 1800 Hz
f = (0:1/kmax:(1-1/kmax))*fmax;      % frequencies from 0 to 1800 Hz
```

```
Ymag(kmax + 1:end) = [];            % frequencies from 0 to 1800 Hz
close all
plot(f,Ymag,185,1,'r*'); grid on; xlabel('f (Hz)'); ylabel('|X(f)|');
title('pitch-shifted spectrum, r = 1/2'); axis([0,fmax,0,1.1])
```

2.b.II Spectrogram, r = 1/2

```
R = 256;        % Hop size
N = 4096;       % FFT length
Y = stft(y,R,N);    % STFT of y
M = size(Y,2) - 1;
k = 0:N/2;      % positive-frequency half of Nyquist interval
f = k*fs/N;     % f from 0 to fs/2
t = (0:M)*R/fs; % M + 1 time frames spaced by R*Ts = R/fs

Ymag = abs(Y(k+1,:)); % extract positive frequency half
Ymag = Ymag/max(max(Ymag));

S = 20*log10(Ymag);      % convert to dB

close all
surf(t,f,S,'edgecolor','none') % Create spectrogram with time on the
% horizontal axis and frequency on the vertical axis
colormap(jet); colorbar; axis tight; view(0,90); % View from above
xlabel('{\itt} (sec)'); ylabel('{\itf} (Hz)');
ylim([0,1800]);             % Show frequency range from 0 1800 Hz
title('spectrogram, pitch-shifted, r = 1/2')
```

2.c A Closer Look at the Sample Level

```
fs = 44100; % Sampling Rate
load x4.mat;    % 4 msec sample of flute playing

t = (0:fs*0.004)*1000/ fs; % 4 msec in units of msec

close all
stem(t,x4); xlabel('t (msec)'), title('original at rate f_s')

% Speed up by r =2
r = 2;
[p,q] = rat(1/r);

y2 = resample(x4,p,q);          % resample at fs/r, r = 2
t2 = (0:fs/r*0.004)*r*1000/fs; % 4 msec in units of msec

close all
stem(t2,y2,'r'); xlabel('t (msec)'),
title('resampled at rate f_s/r, r = 2')

% Resample at fs/r and playback at fs

y3 = [y2,zeros(1,length(y2)-1)]; % resmpled signal is contained in the
% first 2 milliseconds
```

```
close all
stem(t,y3,'r'); xlabel('t (msec)'),
title('resampled at f_s/r, replayed at f_s')

% Extend in time by r with phase vocoder at 1/r
Ra = 4;
N = 32;
y4 = phvoc(y2,1/r,Ra,N);
y4 = y4/max(abs(y4));       % Normalize
y4(178:end) = [];           % Make duration equal to that of original
% by removing the zeros padded

close all
stem(t,y4); xlabel('t (msec)'),
title('phase vocoder, time-stretched by factor r = 2')
```

3. COLA Property

```
N = 128;    % Length of window
M = 10;     % Number of blocks
n = 0:N-1;
w = 0.5 - 0.5*cos(2*pi*n/(N-1)); % Hanning window
t = 1:800;
```

3.a R = N/2

```
R = N/2;

wbuff = repmat(w',1,M+1);   % Matrix of w(n) repeated M+1 times

wOla = zeros(1,800);        % Overlap-added window

for m = 0:M                     % Number of blocks (columns)
    y = wbuff(:,m+1);
    for n = 1:N                             % Overlap-add each block
        wOla(m*R + n) = wOla(m*R + n) + y(n);
    end
end

close all
plot(t,wOla),axis([0,800,0,3]), title('window overlap-add, R = N/2'),
xlabel('n');

r = 0:R-1;                  % DFT index
wr = 2*pi*r/R;              % DFT frequencies

% Calculate DFTs according to equation (7)
wDFT = zeros(1,R);
for r = 0:R-1
    for n = 0:N-1
        wDFT(r+1) = wDFT(r+1) + w(n+1)*exp(-1i*wr(r+1)*n);
    end
end
```

```
wDFT = abs(wDFT)/max(abs(wDFT));  % Normalize to unity

r = 0:R-1;                    % DFT index

close all
plot(r,wDFT, 'r*'); grid on; axis([-10,70,-0.1,1.1]);
xlabel('DFT index, k'), title('R-point DFT, W(k)'),
```

3.b R = 3N/8

```
R = 3*N/8;

wbuff = repmat(w',1,M+1);    % Matrix of w(n) repeated M+1 times

wOla = zeros(1,800);         % Overlap-added window

for m = 0:M                       % Number of blocks (columns)
    y = wbuff(:,m+1);
    for n = 1:N                               % Overlap-add each block
        wOla(m*R + n) = wOla(m*R + n) + y(n);
    end
end

close all
plot(t,wOla),axis([0,800,0,3]), title('window overlap-add, R = 3N/8'),
xlabel('n');

r = 0:R-1;                    % DFT index
wr = 2*pi*r/R;                % DFT frequencies

% Calculate DFTs according to equation (7)
wDFT = zeros(1,R);
for r = 0:R-1
    for n = 0:N-1
        wDFT(r+1) = wDFT(r+1) + w(n+1)*exp(-1i*wr(r+1)*n);
    end
end

wDFT = abs(wDFT)/max(abs(wDFT)); % Normalize to unity

r = 0:R-1;                    % DFT index

close all
plot(r,wDFT, 'r*'); grid on; axis([-10,70,-0.1,1.1]);
xlabel('DFT index, k'), title('R-point DFT, W(k)'),
```

3.c R = N/3

```
R = round(N/3);

wbuff = repmat(w',1,M+1);    % Matrix of w(n) repeated M+1 times
```

```
wOla = zeros(1,800);        % Overlap-added window

for m = 0:M                     % Number of blocks (columns)
    y = wbuff(:,m+1);
    for n = 1:N                             % Overlap-add each block
        wOla(m*R + n) = wOla(m*R + n) + y(n);
    end
end

close all
plot(t,wOla),axis([0,800,0,3]), title('window overlap-add, R = N/3'),
xlabel('n');

r = 0:R-1;                  % DFT index
wr = 2*pi*r/R;              % DFT frequencies

% Calculate DFTs according to equation (7)
wDFT = zeros(1,R);
for r = 0:R-1
    for n = 0:N-1
        wDFT(r+1) = wDFT(r+1) + w(n+1)*exp(-1i*wr(r+1)*n);
    end
end

wDFT = abs(wDFT)/max(abs(wDFT)); % Normalize to unity

r = 0:R-1;                  % DFT index

close all
plot(r,wDFT, 'r*'); grid on; axis([-10,70,-0.1,1.1]);
xlabel('DFT index, k'), title('R-point DFT, W(k)'),
```

3.d R = N/4

```
R = N/4;

wbuff = repmat(w',1,M+1);   % Matrix of w(n) repeated M+1 times

wOla = zeros(1,800);        % Overlap-added window

for m = 0:M                     % Number of blocks (columns)
    y = wbuff(:,m+1);
    for n = 1:N                             % Overlap-add each block
        wOla(m*R + n) = wOla(m*R + n) + y(n);
    end
end

close all
plot(t,wOla),axis([0,800,0,3]), title('window overlap-add, R = N/4'),
xlabel('n');

r = 0:R-1;                  % DFT index
wr = 2*pi*r/R;              % DFT frequencies
```

```
% Calculate DFTs according to equation (7)
wDFT = zeros(1,R);
for r = 0:R-1
    for n = 0:N-1
        wDFT(r+1) = wDFT(r+1) + w(n+1)*exp(-1i*wr(r+1)*n);
    end
end

wDFT = abs(wDFT)/max(abs(wDFT)); % Normalize to unity

r = 0:R-1;                  % DFT index

close all
plot(r,wDFT, 'r*'); grid on; axis([-10,70,-0.1,1.1]);
xlabel('DFT index, k'), title('R-point DFT, W(k)'),
```

Appendix A.1: STFT

```
function X = stft(x, Ra, N)
% Calculates the STFT of the input signal x with analysis hop size Ra
 and
% FFT length N

M = floor(length(x)/Ra);        % Determine how many blocks, (will be
 M+1)
Lext = M*Ra + N;                % Length of extended input signal
x(length(x)+1:Lext) = 0;        % Zero pad input signal
Xbuff = buffer(x, N, N-Ra,'nodelay'); % Separate signal into size N
 blocks with overlap N-Ra
n = 1:N;                                % For window of length N
w = 0.5 - 0.5*cos(2*pi*n/(N-1));        % Hanning window
w = w';                                 % Transpose window
W = repmat(w,1,M+1);                    % Window each block
X = fft(W.*Xbuff,N);                    % Take the FFT of windowed blocks
end
```

Appendix A.2: Phase Map (phmap function)

```
function Y = phmap(X,r,Ra,N)
% Maintains correct phase under time scale modifications
% N = STFT block size, X = N x m STFT, r = speed up factor
% Ra = analysis hop size

mod2pi = @(x) mod(x + pi, 2*pi) - pi;

[~,M] = size(X);        % Determine how many columns, M+1, in X
M = M-1;                % Determine the value of M
Rs = round(Ra/r);       % Determine synthesis hop size

Ymag = abs(X);          % Preserve magnitude of input
Phi = angle(X);         % Transfer phase angles of input
```

```
k = 0:(N-1);
wk = 2*pi*k'/N;

Psi(:,1) = Phi(:,1);     % Psi(k,0) = Phi(k,0)

    for m = 1:M
        dw = (1/Ra)*mod2pi(Phi(:,m+1) - Phi(:,m) - Ra*wk); % Bring
 omega within the Nyquist interval
        Psi(:,m+1) = Psi(:,m) + Rs*(wk + dw);
    end
    Y = Ymag.*exp(1i.*Psi);
end
```

Appendix: A.3 ISTFT

```
function y = istft(Y, Rs, N)

% Takes the inverse STFT of Y with synthesis hop size Rs and
% FFT length N

[~,M] = size(Y);         % Determine how many columns, M+1, in Y
M = M-1;                 % Determine the value of M
Ly = M*Rs + N;           % Determine length of the output signal
y = zeros(Ly,1);         % Initialize y(n) to zeros
n = 0:(N-1);                          % For window of length N
w = 0.5 - 0.5*cos(2*pi*n/(N-1)); % Hanning window
w = w';                              % Transpose window
```

Appendix A.4: Phase Vocoder (phvoc function)

```
function y = phvoc(x, r, Ra, N)
% Applies the phase vocoder effect to the input signal
% N = STFT block size, x = input signal, r = speed up factor
% Ra = analysis hop size

Rs = round(Ra/r);        % Determine synthesis hop size

% This is necessary to determine the correct output length when
 expanding
% the signal in time
if r < 1
    M = floor(length(x)/Ra);
    Ly = M*Rs + N;
    x(end + 1:Ly) = 0;  % Zero pad input signal to be the length of
 the output signal
end

X = stft(x, Ra, N);      % Calculate the STFT
Y = phmap(X, r, Ra, N); % Map the phases to preserve them
y = istft(Y, Rs, N);     % Calculate the ISTFT

end
Ybuff = real(ifft(Y,N));              % Take the IFFT of Y
```

```
for m = 0:M                         % Number of blocks (columns)
    ym = Ybuff(:,m+1);              % Vectorize windowing
    ym = ym.*w;                     % Window each block of Ybuff
    for n = 1:N                                 % Overlap-add each block
        y(m*Rs + n) = y(m*Rs + n) + ym(n);  % ym(n) is already
 windowed
    end
end

end
```

Appendix A.5: Pitch Modification (pitchmod function)

```
function y = pitchmod(x, r, Ra, N)
% Modifies pitch without changing duration
% N = STFT block size, x = input signal, r = speed up factor
% Ra = analysis hop size

[p,q] = rat(1/r);           % Rationalize spead up factor for
 resampling

if p <= q                   % If r >= 1 resample at 1/r first, then
 phvoc
    x = resample(x, p, q);  % Resample at 1/r
    y = phvoc(x, 1/r, Ra, N);   % Map phases to 1/r
else                            % If r < 1 phvoc, then resample at 1/r
    x = phvoc(x, 1/r, Ra, N);   % Map phases to 1/r
    y = resample(x, p, q);      % Resample at 1/r
end

end
```

Published with MATLAB® R2017b