Real-time Pitch Correction

Nikolas Borrel-Jensen s081782
Nis Nordby Wegmann s082079
Andreas Hjortgaard Danielsen s081783

Department of Informatics and Mathematical Modelling,
Technical University of Denmark

Copenhagen, June 8, 2009

Abstract

In this report we describe a method for pitch correction: three methods for pitch detection and two methods used for pitch shifting. The pitch detection presented has a correct detection in more that 90 percent of the time. The pitch shifting algorithm presented can be used in a real-time environment, though with some reverberation introduced. The end product is an AudioUnit plugin that can be used with a commercial sequencer program on the Mac platform.

Keywords: pitch correction, pitch detection, pitch shifting, voice signal, real-time, harmonic product spectrum, cepstrum, phase vocoder, Bernsee, Audio Unit

Contents

1 Introduction

In the music industry the AutoTune technology has become a standard in the professional studio. This technology is mainly used in the post-processing stages of song creation to correct notes that are slightly out of pitch or to create some sort of vocal effect. A prominent example of such an effect is heard on Cher's Believe from 1998.

Since pitch correction is usually done in the post-processing stage, no serious constraints are set with respect to performance. This report will discuss and implement a pitch correction method for use in a real-time setting. This imposes strict performance constraints of a whole other scale than the non-real-time implementation. First of all, we need to split the signal up into frames that are small enough to be processed without any notable delay. Secondly, the pitch must be shifted on the fly, not knowing how the rest of the signal will evolve. These contraints prove to be disastrous when trying to convert the standard pitch correction methods into this real-time environment.

This report will describe different approaches to the real-time pitch correction problem. Pitch correction consists of the following two steps:

1. Pitch detection – detect the fundamental frequency of the pitch being sung
2. Pitch shifting – shift the pitch to the desired tone

We will split this report into five parts: Section 2 will describe how the overall processing of the incoming signal is done, i.e. how to split the signal into frames and how to categorize them. Section 3 will describe three methods for pitch detection. It will go through theory and implementation along with testing of all three algorithms. Section 4 will describe two methods for pitch shifting: One using the phase vocoder, which is described both in terms of theory and implementation and one using a method by Stephan Bernsee of dspdimension.com [2]. This method will be documented and tested. In Section 6, the AudioUnit implementation will be described. Finally, we will test the precision of the pitch detection and make a perceptual test of the pitch shifting and the AudioUnit implementation.

We would like to thank Stephan Bernsee for letting us use his pitch shifter and for quick responses to our questions. Thanks to Mette Tvarnø and Peter (Munter) Müller for lending us their voices during the test phase. And last, but definitely not least, thanks should go to our supervisor Jan Larsen for his encouragement and technical counseling.

2 Outer shell

Correcting the pitch of a voice, can be divided into two well-known problems in signal processing: Pitch detection and pitch shifting. In order to correct a signal, we have to determine the fundamental frequency of the actual pitch being sung, and the do the pitch shifting according to a given factor. We have constructed a shell in our MATLAB implementation for simulating a real-time environment. This outer shell reads in a WAV-file encoded with a sampling rate of 44.1 kHz and splits it into small frames of size 1024. This gives a latency of 23 ms, since a whole frame has to be read in before it can be processed. Real-time application often requires a latency of less than 10 ms[7], so our 23 ms is a bit too much latency to be considered real-time, but because of the quality of our methods we have estimated that a good trade-off between quality of sound and latency have been reached with 1024 samples per frame, supported by[7]. We assume that a 23 ms frame can be considered stationary, because we have done an empirical test, and concluded that a singer do not have time to change pitch in the course of 23 ms.

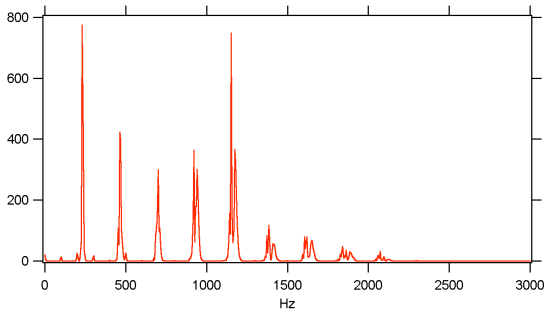

Figure 1: A frequency spectrum for a human voice signal; the formants appears as sharp peaks occuring at integer multiples of the fundamental frequency

Detecting and shifting a pitch in a frame of course requires the existence of pitch information in the frame. In other words, if the frame is silent or is governed by noise, there's no way to detect any pitch in that frame. Hence, for making our pitch corrector work correctly, we will define criteria for which frames to correct. We define three types of frames: silent, unvoiced and voiced.

Let $x_u(n)$ be the u'th frame of a larger signal $x(n)$ and where N is the frame size. Then we define the absolute average of the frame as

$$\text{abs}_{\text{average}} = \frac{1}{N}\sum_{n=1}^{N}|x_u(n)| \quad (1)$$

We define a silent frame as a frame that has an absolute average less than a given threshold.

An unvoiced frame is determined according to a method called zero-crossings that determines how many times the signal amplitude changes signs during the frame. If the a sign change occurs more than 45% of the time we have chosen to classify the frame as unvoiced.

Frames that does not have an absolute average less than a given threshold and has a low percentage of fluctuations will be classified as voiced frames which we can process in our pitch corrector.

3 Pitch detection

In this paper we will consider two basic methods for detecting the fundamental frequency of a human voice signal:

- the Harmonic Product Spectrum (HPS) pitch detection algorithm
- the Cepstrum pitch detection algorithm

Further, we present a method which combines the above two methods:

- the CBHPS pitch detection algorithm

In Section 5 we analyse, evaluate and compare the algorithms to each other, based on their capability in detecting the fundamental frequency of male and female voice signals.

3.1 Frequency domain analysis of human voice signals

In human voice signals the energy of the signal appears in the frequency domain as distinct peaks, or *formants*, occurring at integer multiples of the fundamental frequency. Each formant is formed by a resonance in the human vocal tract. In figure 1 is shown a frequency spectrum of a human voice signal.

Both the HPS and the Cepstrum pitch detection algorithms are based on a frequency domain analysis of the voice signal, and exploit the above mentioned trait. As our signal arrives in small frames of 23 ms, of which we have to determine the fundamental frequency individually, we use a short-time Fourier transformation (stFT) on each frame.

3.2 Harmonic Product Spectrum

The HPS pitch-detection algorithm [11] is a simple and robust method for determining the fundamental frequency of a signal. The HPS algorithm measures the coincidences of the formants by downsampling the frequency spectrum with integer factors, and then by multiplying each of the downsampled spectrums pointwise together with the original spectrum. The result of this process is called Harmonic Product Spectrum. In that way, peaks in the original spectrum occuring at integer multiples of each other, will reinforce and cause a peak in the harmonic product spectrum. This is depicted in figure 2.

The mathematical definition of the HPS algorithm is given by the two following equations:

$$\begin{aligned} \mathrm{HPS(k)} &= \prod_{r=1}^{R} |X(kr)|^2, \quad 0 \le k < N \qquad (2) \\ \hat{k} &= k_i, \quad HPS(k_i) = \max(HPS) \qquad (3) \end{aligned}$$

where $X(k)$ is the N-point DFT of the input signal $x(t)$, R is the selected number of harmonics to be considered and $\hat{k}$ is the index determined by searching the harmonic product spectrum for the frequency k_i which maximizes the value of HPS(k_i); thus k_i has to be in the range of harmonics corrosponding to potential fundamental frequencies (e.g., 50 - 2000hz).

The main advantages of the HPS algorithm is its simplicity and superior resistance to additive noise [11]. As stochastic noise has an incoherent structure in the frequency domain, the contributions of the noise will add randomly to the peaks of HPS(k) in (2). A drawback of the HPS algorithm is that in order to estimate the fundamental frequency at a proper resolution the signal must be zero padded before its transformation into the frequency domain. This increases the computational cost of the algoritm considerably compared to other pitch detection algorithms.

3.2.1 Logarithmic HPS

Taking the logarithm of $HPS(k)$ yields the *Logarithmic Harmonic Product Spectrum*

$$\begin{aligned} \mathrm{LHPS(k)} &= \log \mathrm{HPS(k)} \\ &= \log(\prod_{r=1}^{R} |X(kr)|^2) \\ &= 2\sum_{r=1}^{R} \log |X(kr)| \end{aligned}$$

Typically, the peak amplitudes in HPS are very sharp due to reinforcement at coinciding frequencies. Taking the logarithm of HPS flattens these peaks, which makes the HPS algorithm more suitable for being used in combination with other pitch-detection algorithms.

3.3 Cepstrum

Determining the pitch of a speech signal from the frequency spectrum by locating the bin with the greatest magnitude is not possible, since the speech consists of an excitation sequence convolved with the impulse responses of the vocal system. Instead, we would like to extract the excitation from the signal, but since the Fourier transform obey the superposition principle, we can't use it directly to separate the two signals. Though, viewing the spectrum $|S(\omega)|$ of the speech $s(x)$ as consisting of a fast varying part $|E(\omega)|$, and a slowly varying part $|\Theta(\omega)|$, where the first is the excitation and the latter is the impulse response of the vocal system, we can take the logarithm to get the multiplicative terms into additive terms:

$$\begin{aligned} C_s(\omega) &= \log |S(\omega)| \\ &= \log |E(\omega)\Theta(\omega)| \\ &= \log |E(\omega)| + \log |\Theta(\omega)| \\ &= C_e(\omega) + C_\theta(\omega) \end{aligned}$$

Because the Fourier transform works individually on the two pieces because of the linearity, and therefore separates the fast varying piece and the slowly varying piece, we can compute Fourier series coefficients for the rahmonics[1] of the "signal" by

$$a_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} C_s(\omega) e^{-j\omega n} d\omega \qquad (4)$$

Because $C_s(\omega)$ is a real, even function of ω, we can write (4) as

$$c_s(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} C_s(\omega) e^{j\omega n} d\omega$$

[1] equivalent to harmonics in the spectrum

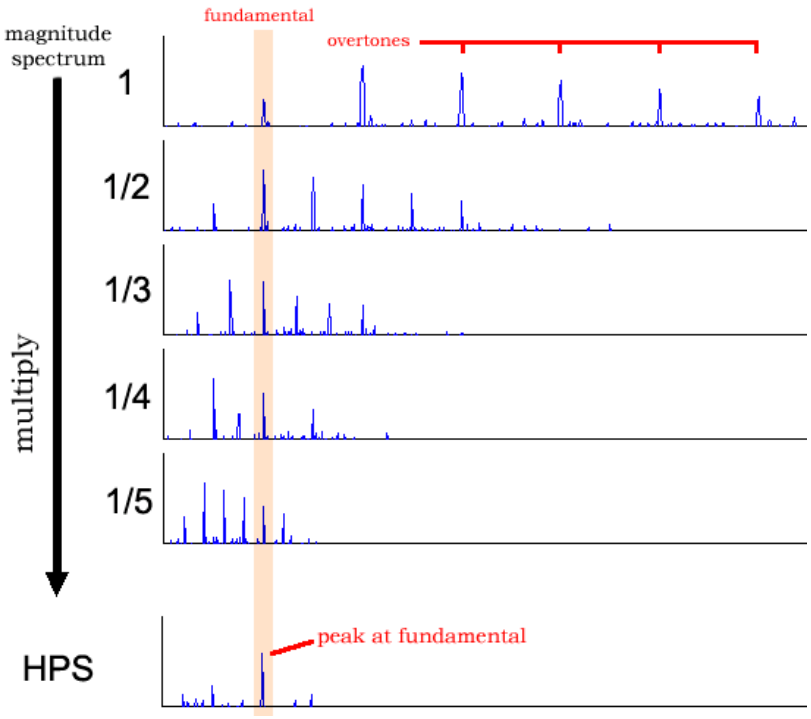

Figure 2: The HPS algorithm

where $c_s(n)$ is the inverse Fourier series coefficients of $C_s(\omega)$, and we can write

$$c_s(n) = c_e(n) + c_\omega(n) \tag{5}$$

where $c_e(n)$ is the IFT of C_e and $c_\theta(n)$ is the IFT of $C_\theta(n)$. This new "spectrum" is called ceptrum, and from that it is possible to examine the excitation c_e and get the pitch. To summarize, what we do is simply

$$\text{Ceps}(n) = |\text{IDFT}[\log|S(n)|]| \tag{6}$$

Searching the cepstrum for the peak with the greatest magnitude in a given interval, we have found the fundamental frequency of the signal.

3.3.1 Pitch estimation in the Cepstrum

We will now show more formally, that it is possible to extract C_e and from that estimate the pitch. As mentioned above, the voiced speech signal is modeled in the time domain as the convolution of the excitation and the impulse response of the vocal system

$$s(n) = e(n) * \Theta(n)$$

Since we are working on frames instead of the entire speech signal, we will use the notation from [4] for the Short-Time processing of signals. We define a frame of speech to be the product of a shifted window with the speech sequence

$$f(n;m) \triangleq s(n)w(m-n)$$

where w is the window and s is the signal. The frame is just a new sequence on n, which is zero outside $n \in [m-N+1, m]$, where N is the size of the frame. First, by [4] it has been shown, that a frame of speech ending at time m

$$f_s(n;m) = s(n)w(m-n) = [e(n) * \Theta(n)]w(m-n)$$

can be written as

$$\begin{aligned} f_s(n;m) &\approx e(n)w(m-n) * \Theta(n) &(7)\\ &= f_e(n;m) * \Theta(n) &(8)\end{aligned}$$

where $f_e(n;m)$ is the frame of $e(n)$ determined by the window ending at time m. Using this approximation, we want to find the Real Cepstrum (RC) of the speech, which are the RC of the sum of the two convolved components, $f_e(n;m)$ and $\Theta(n)$

$$c_{f_s}(\omega) = c_{f_e}(\omega) + c_e(\omega)$$

In fact, we want to find the short-term Real Cepstrum (stRC) of the excitation and the RC of the vocal tract impulse response, but by [4] the statement is correctly written. Let us first look at the stRC of the excitation. Assume that we have Q periods of the excitation $e(n)$ inside the frame $f_e(n;m)$, and let those periods correspond to indices $q = q_0, q_0+1, \ldots, q_0+Q-1$, then

$$f_e(n;m) = \sum_{q=q_0}^{q_0+Q-1} w(m-qP)\delta(n-qP)$$

Computing the Fourier transform of this discrete-time aperiodic signal, we get

$$E(\omega;m) = \sum_{q=q_0}^{q_0+Q-1} w(m-qP)e^{-j\omega qP} \tag{9}$$

Inspired by [4], we define the sequence

$$\tilde{w}(q) = \begin{cases} w(m - qP), & q = q_0, \ldots, q_0 + Q - 1 \\ 0, & \text{other} \end{cases}$$

We insert that into (9), and we get

$$\begin{aligned} E(\omega; m) &= \sum_{q=q_0}^{q_0+Q-1} \tilde{w}(q) e^{-j\omega qP} \\ &= \tilde{W}(\omega P) \end{aligned}$$

where

$$\tilde{W}(\omega) = \sum_{q=-\infty}^{\infty} \tilde{w}(n) e^{-j\omega q}$$

is the the DTFT of the sequence $\tilde{w}(q)$. Since

$$E(\omega; m) = \tilde{W}(\omega P)$$

we can conclude that $\log|E(\omega, M)|$ will be a periodic function[2] of ω with period $2\pi/P$.

Now taking the IDTFT of $\log|E(\omega; m)|$ we get

$$c_e(n; m) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log|E(\omega; m)| e^{j\omega n} d\omega \tag{10}$$

Since the period of $\log|E(\omega; m)|$ is $2\pi P$, the rahmonics occur at times $n = \frac{i2\pi}{2\pi/P} = iP$, for $i = 1, 2, \ldots$. Hence

$$c_e(n; m) = \sum_{i=-\infty}^{\infty} \alpha_i \delta(n - iP)$$

where the numbers α_i depends on $\log|E(\omega; m)|$ and can be computed by (10). This means, that c_e is a weighted train of impulse responses equally spaced by P, hence the pitch is equal to P, decreasing $1/i$ due to the property of the Fourier coefficients.

Now let's look at the vocal system impulse response. The RC for $\theta(n)$ is

$$c_\theta(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log|\theta(\omega)| e^{j\omega n} d\omega$$

By [4], the RC will decay as $1/n$ due to the property of the Fourier coefficients, and this envelope will usually decay quickly with respect to the typical values of the pitch period P. We now have the result sketched in (5)

$$c_s(n; m) = c_e(n; m) + c_\omega(n)$$

As shown in the experimental subsection, $c_\theta(n)$ decays very quickly, and we can therefore do the following assumption

$$c_s(n; m) = \begin{cases} c_e(0; m) + c_\theta(0), & n = 0 \\ c_\theta(n), & 0 < n < P \\ c_e(n; m), & n \geq P \end{cases}$$

[2] since $w(x)$ is an aperiodic signal

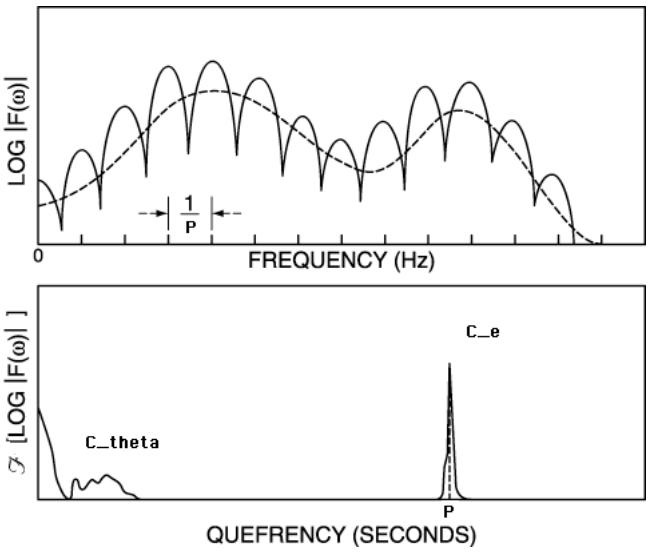

Figure 3: The frequency domain and the cepstrum domain, showing the excitation (solid line) and the vocal tract (dotted line) impulse responses

It is now possible to distinct the vocal tract and the excitation, and we only have to locate the initial peak of the $c_e(n; m), n \neq 0$ (not the DC coefficient).

This is depicted in figure 3. Here we see, that the first cepstrum peak is found at index P. The next peaks will appear at multiples of P, but is not shown in the figure. We also see how the vocal tract impulse response c_θ is decaying fast with respect to the excitation c_e.

3.4 Cepstrum-Biased HPS

In order to combine the Cepstrum and the HPS pitch-detection algorithm, the cepstrum must be converted to the frequency domain. To do this we observe that for the HPS

$$f = k \cdot \frac{f_s}{N}$$

where f is the frequency component in hertz corresponding to the k'th harmonic in the HPS, N is the resolution of the HPS and f_s is the samplerate.

For the Cepstrum we have that

$$f = \frac{f_s}{n}$$

and here f is the frequency component in hertz corresponding to the n'th harmonic in the cepstrum. Hence, the following relation exists among frequency components in the HPS and the cepstrum

$$k \cdot \frac{f_s}{N} = \frac{f_s}{n} \Rightarrow \frac{k}{N} = \frac{1}{n} \Rightarrow \frac{N}{k} = n$$

From these observations we get that the *Frequency Indexed Cepstrum* (FIC) can be calculated by writting all values in the cepstrum at index n to the index $\frac{N}{n}$ in the FIC. Obviously for some values of n, we will have that they point at

coinciding indices in the FIC; in such cases we choose the largest of the possible values. Thus FIC can be defined as follows

$$\mathrm{FIC}(k) = \max\{\texttt{ceps}(n) \mid 0 \leq n < N, k = \mathrm{floor}(\frac{N}{n})\}$$

After calculating the FIC the *Cepstrum-Biased HPS* (CBHPS) can be defined by multiplying the FIC pointwise with LHPS

$$\mathrm{CBHPS}(k) = \mathrm{FIC}(k) \cdot \mathrm{LHPS}(k)$$

In the end the fundamental frequency is determined by searching the CBHPS for the index $\hat{k}$ which maximizes the value of $\mathrm{CBHPS}(\hat{k})$.

3.5 Resolution errors

In this section, we will look at the resolution errors in the Cepstrum and the HPS method. First, in the cepstrum method, the fundamental frequency for a bin in the quefrency domain is calculated by

$$\mathrm{freq} \quad = \quad \frac{f_s}{n} \tag{11}$$

where n is the index of the bin and f_s is the sampling rate. From the formula, we see that the resolution will get worse when the index gets smaller, meaning that the frequency gets higher. Contrary to the Cepstrum method, the HPS method has a constant resolution error, given by

$$\Delta_{\mathrm{HPS}} \quad = \quad \frac{f_s}{N} \tag{12}$$

where N resolution of the HPS. With a frame size of 8192 (can be achieved by zero padding) and a sampling rate of 44100 Hz, we get the resolution error

$$\Delta_{\mathrm{HPS}} \quad = \quad \frac{44100}{8192} = 5.38 \text{ samples} \tag{13}$$

Since the frequency mapped to musical notes are logarithmic, i.e. the difference in frequency from A (49) to Bb (50) is 26.16 where the difference between A (61) to Bb (62) is 52.33, a given resolution error is worse in the lower frequencies than in the higher frequencies, and on the basis of that we can calculate the relative resolution error. Generally, the resolution error can be expressed as

$$\text{resolution error} \quad = \quad \frac{f_{q_{i+1}} - f_i}{f_{i+1} - f_i} \tag{14}$$

Here, f_i is the fundamental frequency of a given tone in the 12-tone scale, f_{i+1} is the fundamental frequency of next half tone from f_i in the 12-tone scale and $f_{q_{i+1}}$ is the fundamental frequency of the next bin with the frequency f_i.

We will now find the explicit expressions for both the Cepstrum- and the HPS method. Let's look at the Cepstrum first. By (11), we have

$$\Delta_{\mathrm{cep\_log}} \quad = \quad \frac{\frac{44100 \text{ Hz}}{44100 \text{ Hz}/f_i} - \frac{44100 \text{ Hz}}{44100 \text{ Hz}/f_i - 1}}{f_{i+1} - f_i} \tag{15}$$

To calculate the denominator, we will use the widely used MIDI standard[3] to map fundamental frequency f to a real number p as follows

$$p \quad = \quad 69 + 12 \cdot \log_2(\frac{f}{440\text{Hz}}) \tag{16}$$

This creates a linear pitch space in which octaves have size 12, semitones (the distance between adjacent keys on the piano keyboard) have size 1, and A440 is assigned the number 69. Expressing (16) with respect to f, we get

$$f \quad = \quad 440 \cdot 2^{(p-69)/12} \tag{17}$$

Now, combining (15), (16) and (17), we get the desired result

$$\Delta_{\mathrm{cep\_log}}(f) \quad = \quad \frac{\frac{44100 \text{ Hz}}{44100 \text{ Hz}/f} - \frac{44100 \text{ Hz}}{44100 \text{ Hz}/f - 1}}{440 \cdot 2^{69+12\log_2(f/440+1-69)/12} - f} \tag{18}$$

$$= \quad -\frac{f}{(-44100 + f)(2^{1/12} - 1)} \tag{19}$$

The resolution error for the HPS is constant, and from (12) and (14), with a sampling rate of 44100 Hz, we get

$$\Delta_{\mathrm{HPS\_log}}(f) \quad = \quad \frac{\frac{44100 \text{ Hz}}{\text{frame size}}}{440 \cdot 2^{69+12\log_2(f/440+1-69)/12} - f} \tag{20}$$

$$= \quad -\frac{11025}{f \cdot \text{frame size} \cdot (2^{1/12} - 1)} \tag{21}$$

$\Delta_{\mathrm{cep\_log}}(f)$ and $\Delta_{\mathrm{HPS\_log\_err}}(f)$ is depicted in figure 4 for at frame size of 8192 samples.

3.5.1 Selecting the right algorithm

From figure 4 it is obvious that for low frequencies the Cepstrum, in terms of resolution error, is the most precise, while for higher frequencies the CBHPS is more accurate. As the CBHPS method in general is better at selecting the right fundamental frequency than the Cepstrum or the HPS alone, the proper thing to do is to initially let the CBHPS determine the fundamental frequency f_{cbhps}. Afterwards if f_{cbhps} is a *high frequency* we keep it; but if instead it is a *low frequency*, we search in the cepstrum for the highest peak in a small delta around the index corresponding to the frequency f_{cbhps}.

To determine which frequencies are high and which are low, we have to find the frequency f for which

$$\Delta_{ceps}(f) = \Delta_{cbhps}(f) \tag{22}$$

where $\Delta_{hps}(f)$ and $\Delta_{ceps}(f)$ is the resolution error at frequency f for the HPS and the cepstrum respectively. As mentioned the resolution error for the CBHPS is constant for all frequencies and can be defined as (12) for the HPS. The resolution error for the cepstrum is

$$\Delta_{ceps}(f) = \frac{fs}{k} - \frac{fs}{k-1}, \quad k = \frac{fs}{f} \tag{23}$$

[3] http://en.wikipedia.org/wiki/Pitch_(music)

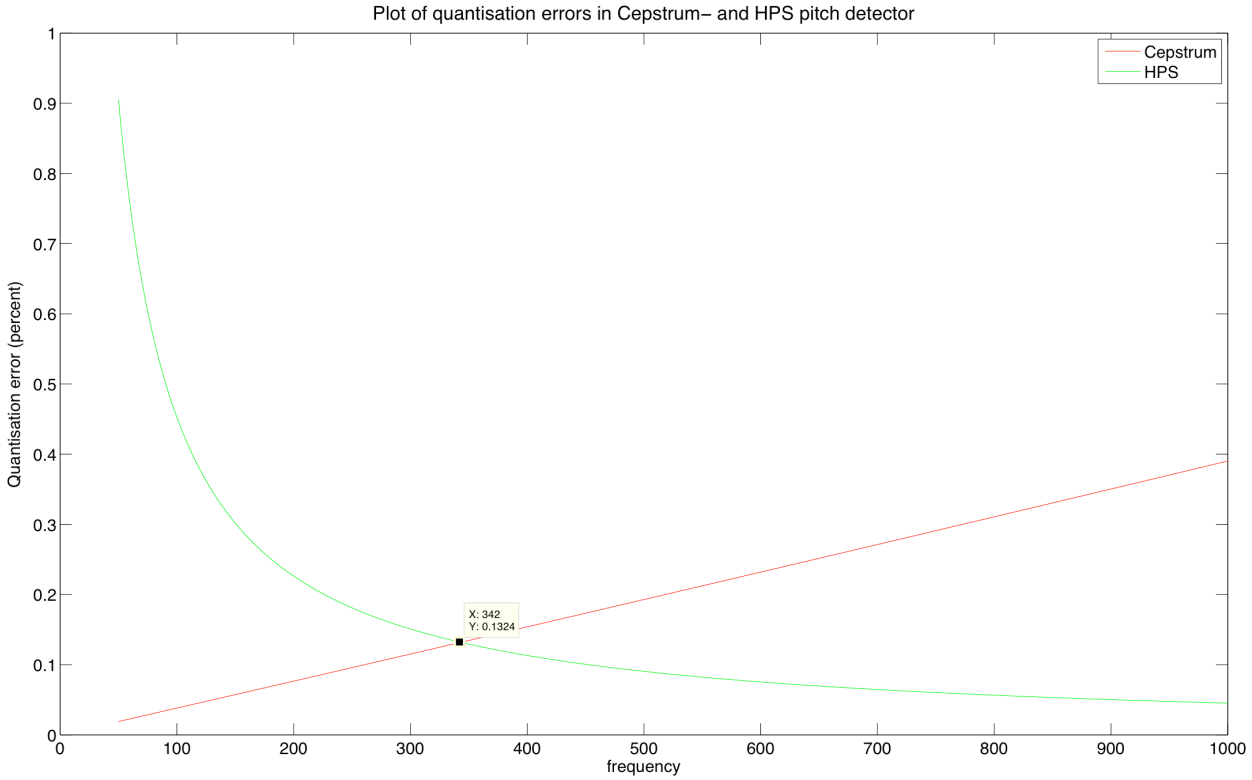

Figure 4: The relative resolution error for both the Ceptrum-and the HPS method, with a frame size of 8192

In order to find the frequency f which satisfies (22), we must solve the following equation

$$\frac{f_s}{k} - \frac{f_s}{k-1} = \Delta_{cbhps}$$

$$\Downarrow$$

$$\Delta_{cbhps}k^2 + \Delta_{cbhps}k - fs = 0$$

$$\Downarrow$$

$$k = \frac{-\Delta_{cbhps} + \sqrt{\Delta_{cbhps}^2 + 4\Delta_{cbhps}fs}}{2\Delta_{cbhps}}$$

After having determined k, its frequency-value can be calculated due to (23):

$$f = \frac{f_s}{k}$$

4 Pitch shifting

We now present two algorithms for pitch shifting. The first algorithm is based on the phase vocoder and the second algorithm is based on a method derived by Stephan Bernsee [2]. Both algorithms are analysis/synthesis-systems and they have very similar analysis parts. The actual processing (pitch shifting) is what differentiates the two methods.

The phase vocoder uses the results from the analysis to time-scale the signal without altering the frequency content and then resample to achieve the pitch shifting. The idea of Bernsee's method is simply to shift the bins in the frequency domain by multiplying the index of the bins by a given factor, resulting in a pitch shifted signal. Though, doing this directly would introduce some serious artifacts. Having a frequency component that is not a multiple of f_s/N, where f_s is the sample rate and the N is the frame size, will create leakage in the frequency spectrum and by this introduce some artificial frequencies, and a shift of these frequencies will therefore destroy the new signal. If the signal is not pitch shifted, i.e. shifted by a factor 1, the smearing is canceled out when we take the IDFT, but we can not expect this after shifting the bins. Therefore both methods use overlapping frames, which increase the resolution. Using a suitable number of overlapping frames, we will achieve all sinusoids (indicated by the bins) with great influence, i.e. relative big magnitude, to have about the same frequency, and the error can be considered negligible [2]. An example can be seen in table 1. Both methods can be divided into an analysis step, where the true frequency and phase information is found, a processing step, where the actual pitch shifting is done, and a synthesis step, where we use the information gathered in the analysis step to retrieve a synthesised time-signal that has been pitch shifted.

Specifically, formula (29), (30) and (31) from the phase vocoder are equivalent to formula (43), (44) and (45), respectively, of Bernsee's method.

4.1 Phase Vocoder

A vocoder is a way of coding voice signals for transmission over channels with limited bandwidth. The word vocoder is actually a contraction of voice coder. There are several kinds vocoders but the one we're interested in is the phase vocoder, which was described by Flanagan and Golden in

Bin number	Bin frequency [Hertz]	Bin magnetude	Bin phase difference	Estimated true frequency
110	2368.652344	0.024252	-2.356196	2336.352516
111	**2390.185547**	**0.169765**	**2.356194**	**2422.485348**
112	**2411.718750**	**0.848826**	**0.785398**	**2422.485352**
113	**2433.251953**	**0.848826**	**-0.785398**	**2422.485351**
114	**2454.785156**	**0.169765**	**-2.356194**	**2422.485355**
115	2476.318359	0.024252	2.356196	2508.618186

Table 1: Shows how the overlap increases the resolution of the frequency domain. Here we use an overlap factor of 4 (or an overlap of 3/4) *http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/*

1966 [5]. It has the special property of time-scaling that is the key to the pitch shifting algorithm.

Before we discuss how the phase vocoder is capable of shifting the pitch, let's take a detailed look at how the standard phase vocoder works.

4.1.1 The analysis/synthesis system

An analysis/synthesis system is used to efficiently encode speech signals for transmission over a channel. It does so by doing an analysis of the signal to extract vital features according to an underlying model. These features are then encoded and transmitted to the receiver whose synthesis system decodes the features and reconstructs the signal according to the underlying model. The phase vocoder is a special case of such a system that uses the short-time Fourier Transform (stFT) to split the signal into smaller frames.

Flanagan [5] describes the phase vocoder in terms of band-pass filters and phase derivatives. By passing the input signal through a number of contiguous band-pass filters and then recombining them there shouldn't be too much information loss. In other words let $x(n)$ be the input signal and $x_u(n)$ be the output signal of the u'th band-pass filter. Then the reconstructed signal is

$$y(n) = \sum_{u=1}^{M} x_u(n) \tag{24}$$

We then have $y(n) \approx x(n)$.

The flavor of phase vocoder we use isn't based on the rather dated original article but is based on a more recent description by Jean Laroche and Mark Dolson [8]. The following will be heavily based on this article, except for some notational differences.

4.1.2 Analysis

The analysis stage uniformly splits the signal into smaller frames according to the formula $t_a(u) = R_a u$, where $t_a(u)$ indicates the time instant that splits two frames with indices $u-1$ and u. R_a is called the analysis hop factor. An illustration of this is seen below.

R_a	R_a	R_a	$\cdots$	R_a

Each frame is then Fourier-transformed according the *non-heterodyned* stFT. Assuming the transform is calculated as a DFT, we write the stFT of the entire signal $x(n)$ as

$$X(u,k) = \sum_{n=-\infty}^{\infty} h(n)x(t_a(u)+n)e^{-j\Omega_k n} \tag{25}$$

where $h(n)$ is the analysis window (e.g. a Hamming or a Hanning window), $\Omega_k = \frac{2\pi k}{N}$ is the center frequency of the k'th frequency bin, where N is the size of the DFT.

As we will see further below, we recombine the frame using a synthesis hop factor R_s. If we set $R_s = R_a$ we will get an (almost) unaltered signal at the receiving end. But we would like to pitch shift the signal, so we want the signal to be time-scaled. The next section describes how this is done.

4.1.3 Time-scaling

By setting $R_s \neq R_a$ we will get a time-scaled signal, but because we alter the time instants for the frames, the phase of the short-time signals will no longer be aligned. In [8] there is a derivation of the phase-propagation formula, here we will only give a short explanation for it and list the important formulas we use in our implementation.

We assume an underlying model which states that the input signal is a sum of sinusoids:

$$x(t) = \sum_{i=1}^{I(t)} A_i(t)e^{j\phi_i(t)} \tag{26}$$

where

$$\phi_i(t) = \phi_i(0) + \int_0^t \omega_i(\tau)d\tau \tag{27}$$

and ω_i is the instantaneous frequency of the i'th sinusoid. Let $\theta_u^a(k)$ denote the phase of the k'th bin i the u'th analysis frame, that is

$$\theta_u^a(k) = \arctan\left(\frac{\Im(x_u^a(k))}{\sqrt{\Re(x_u(k))^2 + \Im(x_u(k))^2} + \Re(x_u(k))}\right) \tag{28}$$

We calculate the phase-propagation thusly:

$$\begin{aligned}
\Delta\Phi_u(k) &= \theta_u^a(k) - \theta_{u-1}^a(k) - R_a\Omega_k && (29)\\
\Delta_p\Phi_u(k) &= (\Delta\Phi_u(k) + \pi \mod 2\pi) - \pi && (30)\\
\hat{\omega}_u(k) &= \Omega_k + \frac{1}{R_a}\Delta_p\Phi_u(k) && (31)\\
\theta_u^s(k) &= \theta_{u-1}^s(k) + R_s\hat{\omega}_u(k) && (32)
\end{aligned}$$

Let's go through the details of these four formulas, as they reappear in Bernsee's method.

We call $\Delta\Phi_u(k)$ the *heterodyned* phase increment, which is the phase difference between frames $u-1$ and u. $\Delta_p\Phi_u(k)$ is its *principal determination*, i.e. in the interval $[-\pi, \pi]$. This will ensure that the phase difference is centered around 0. Using this phase difference and the analysis hop-factor R_a we can calculate the true frequency $\hat{\omega}_u(k)$ of a nearby sinusoid, which will be a more accurate estimate for the true frequency than reading directly off the bins. Equation (32) is the phase-propagation formula, which is used to calculate the phase for all k in the u'th frame. We use this to construct the output frame $Y(u,k)$ as

$$Y(u,k) \quad = \quad |X(u,k)|e^{j\theta_u^s(k)} \tag{33}$$

which the synthesis steps will convert to the u'th time frame.

4.1.4 Synthesis

At the receiver's end we need to reassemble the signal, which is done by setting new time instants $t_s(u) = R_s u$ where we want to reassemble the signal. For each frame we take the inverse Fourier transform of $Y(u,k)$ to obtain short-time signals

$$y_u(n) = \frac{1}{N}\sum_{k=0}^{N-1} Y(u,k)e^{j\Omega_k n} \tag{34}$$

which we combine to obtain the synthesized signal

$$y(n) = \sum_{u=-\infty}^{\infty} w(n - t_s(u))y_u(n - t_s(u)) \tag{35}$$

where w is a synthesis window.

This is how the standard phase vocoder works. The phase-propagation formula ensures *horizontal phase coherence*, which is coherence in each frequency channel between frames. But the vocoder doesn't ensure phase coherence between frequency channels within a frame; *vertical phase coherence*. The improved phase vocoder as described in [8] offers a fix for this problem. As described in the following sections, some issues arise when using the phase vocoder in a real-time environment and we have therefore not implemented the improved phase vocoder.

4.1.5 Pitch shifting

The steps described above leads to a time-shifted signal with unaltered frequency contents. That is, the signal has become longer or shorter but the spectral qualities (the pitch) hasn't changed. Let $x(n)$ be the input signal to the phase vocoder. If we want to pitch shift this signal by a factor of $\alpha = a/b$, where a and b are positive integers, then we have to set $R_s = \alpha R_a$ in order to scale it correctly. In order to pitch shift this new signal we have to resample it. This is done according to the principles given in [10]: The signal is first upsampled (interpolated) by a factor of a. Then the signal is convolved with a lowpass filter to filter out noise generated by the interpolation and to remove frequencies that become out of range when the signal is downsampled (decimated) by a factor of b.

The implementation uses MATLAB's built-in function `resample`, which is a polyphase implementation of the resampling method described above.

4.1.6 Phase vocoder implementation

Implementing the phase vocoder in a real-time system necessitates, that we work on small frames instead of the full length data. Assuming that the signal is stationary[4] for a frame size of 4096 samples, and we will tolerate a latency of 23 ms (frame size of 1024), we split the data as depicted in figure 5 a). We use the usual phase vocoder techniques on these 4096 samples, which implies time-shifting, phase calculation and resampling. Frame (3) is then returned, and we switch the 4096 sample vector as depicted on figure 5 b). Here, new frame (4) is of course the last frame from previous iteration a). The reason for using a frame size of 4096 samples instead of 1024 samples[5] is only for trying to achieve a more smooth transition for subsequent frames after the resampling. Using a window of size 512, all, but the last 512 samples are time-stretched exactly as in the non-real-time phase vocoder, and therefore it will not be computed again. Instead, we reuse the original data, except for the first 1024 samples, and start moving the window frame from the last index$-512+1$ sample. We also need to reuse the time-streched signal, and compute which samples to cut and at which index to the start the next iteration. The process of time-streching a signal is depicted in figure 6 using a window frame of 512 samples, 3/4 overlap resulting in a hop factor R_a, of 128, and a scaling factor 13/12, resulting in a hop factor R_s, of 138.

As it can be seen in fig. (6), with these parameters, the loop should stop exactly when frame 28 is reached, and the reconstructed vector should have length 4376. It is important to compute these values exactly, otherwise errors will be made at the beginning and the end of every 1024-frame, given to the phase vocoder. It can also be seen, that we have to cut 1094 samples of the time-stretched signal in the next iteration, which corresponds to 1024 samples in the original signal. These parameters are computed as follows:

$$R_a \quad = \quad \text{floor}(N \cdot (1 - \text{overlap})) \tag{36}$$

$$R_s \quad = \quad \text{floor}(\alpha \cdot R_a) \tag{37}$$

where N is the window frame size, overlap is the amount we want the windows to overlap and α is the scaling factor we want to pitch shift. On the basis of R_a, R_s and the length of the vector (4096 samples) to be processed, we can compute the number of times we can move the window frame inside the vector (nrframes) without exceeding the border. This can be computed as

$$\text{nrframes} \cdot R_a + 1 \quad \leq \quad (\text{len}_{orig} - N + 1) \tag{38}$$

$$\text{nrframes} \quad \leq \quad \frac{(\text{len}_{orig} - N)}{R_a} \tag{39}$$

[4] Remember that we will have to pitch shift the frames different

[5] Actually, the smallest frame size is 1024 + 512 samples, where the 512 samples is the window size for the vocoder. This will be explained in the following

a)	prev frame (1) [1024]	prev frame (2) [1024]	prev frame (3) [1024]	new frame (4) [1024]

b)	prev frame (2) [1024]	prev frame (3) [1024]	new frame (4) [1024]	new frame (5) [1024]

Figure 5: a) A given iteration, where a new 1024 frame (4) is entering the phase vocoder, and the old 1024 frame (3) is leaving. b) The next iteration, where a new 1024 frame (5) is entering the phase vocoder, and the old 1024 frame (4) is leaving.

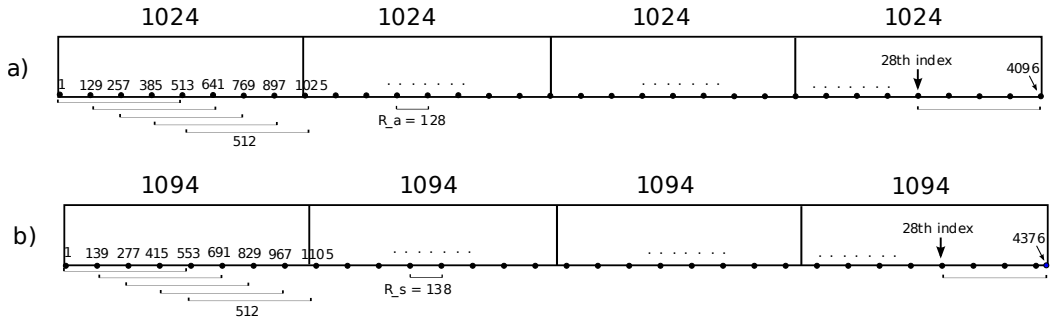

Figure 6: a) Shows how the loop process the original 4096 samples in the phase vocoder, by moving a window R_a samples each iteration. b) Shows how the original signal is time-shifted. The time-stretched signal is made by reconstructing the signal by the new hop factor R_s.

where len_{orig} is the length of the original signal (4096) and R_a is the hop factor computed on the basis of the overlap factor. If the fraction is a natural number, we can use the equality, otherwise we will have to floor the fraction. In the implementation, we will for simplicity always choose a overlap, so that the fraction will be a natural number.

The length of the reconstructed signal is computed as

$$\text{len}_{rec} = N \cdot (\text{nrframes} + 1) - (\text{nrframes}) \cdot (N - R_s) \quad (40)$$

This formula can be interpreted as computing $N \cdot (\text{nrframes} + 1)$ which is the length of the recontructed data, without overlap, and then subtracting the overlap $(N - R_s)$ multiplied with the number of overlap occurrences. As explained above, in the next 4096-iteration we will need to throw 1024 samples away from the original data, since we are getting 1024 new samples in. For the reconstructed signal, we compute the samples to throw away by

$$\text{cut}_{\text{time-streched}} = \frac{\text{size}_{frame}}{R_a} \cdot R_s + 1; \quad (41)$$

where size_{frame} is the size of the new incoming frame. Using this formula with the parameters used until now, we get the result depicted on figure (6), namely 1095 as the start sample. We want to start were the last iteration turn loose, therefore we subtract the window length (512) from the length of original signal (4096) and the reconstructed signal (4376), respectively. Futhermore, the phase and Fourier transform from the last iteration[6] are kept, ensuring that the phase transition will be computed correctly as in the original phase vocoder. When we have the time-stretched signal, we want to resample it, so that the length is the same size as the original size, here 4096, which is done by a a fractional resampling

$$\text{resample factor} = \frac{\text{length of the original signal}}{\text{length of the time-stretched signal}}$$

The time-stretched signal before the resampling is exactly the same as in the original phase vocoder, the difference from this real-time phase vocoder and the original phase vocoder is, that we resample the individual frames. This step introduce some serious artifacts, which is discussed in the next section.

4.1.7 Real-time versus original phase vocoder

Since the AudioUnit template requires that we give it one frame and let it send out the processed frame before receiving another, the idea is to take a frame, analyze it, extract pitch information, calculate new target pitch, shift the signal corresponding to the target pitch and send the frame out again. Since the phase vocoder splits into frames, the plan was to exploit this and integrate the pitch detector into the pitch shifter algorithm. But it turned out that the phase vocoder, as described in the litterature only shifts the full signal by a constant factor, and not each frame with different factors as we would like. So what we do here is to run the full phase vocoder algorithm on each frame by dividing each frame into further subframes.

Because we need to resample each individual frame, the borders between succeeding frames are destroyed. We have tried to smoothen the resampling by taking 1024 samples of the next frame, resample all 4096 (including these 1024 samples), and only returning 1024 samples from index 2049 to index 3073. But as can be seen in figure 7, and indeed be heard, this does not avoid the phase to be destroyed. The result being heard, is a clipping sound between each frame.

[6] Xu_current and PhiY_current in the Matlab implementation

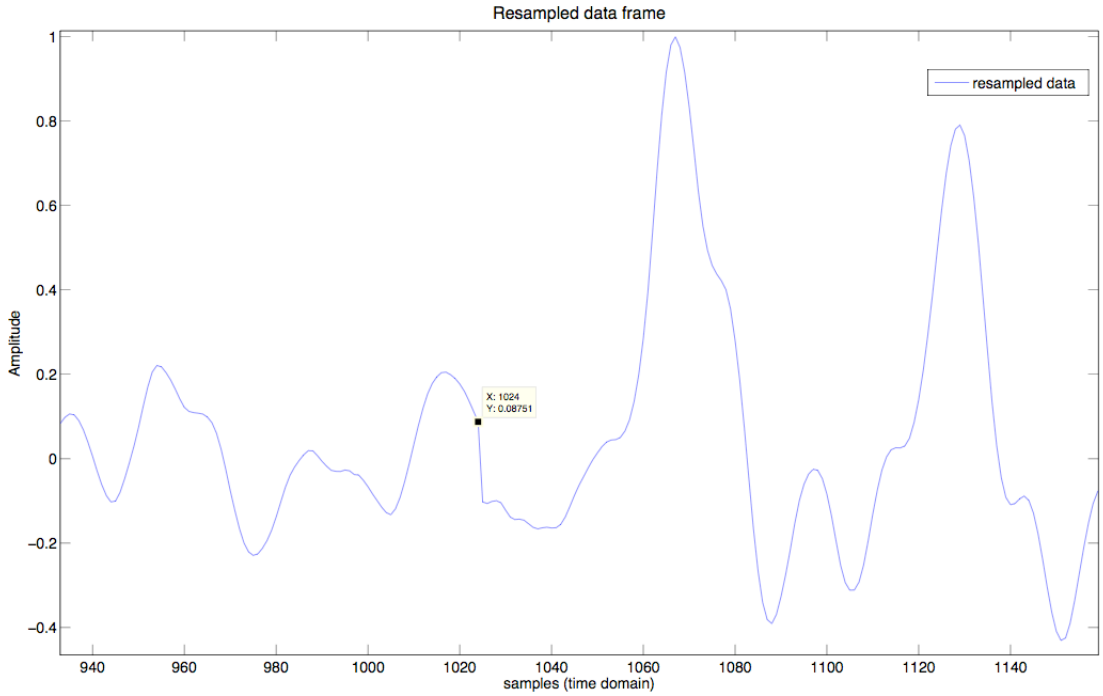

Figure 7: It can be seen, that the difference in amplitude between sample 1024 and sample 1025 has a much greater difference, than the others, causing a clicking sound.

4.2 Bernsee's pitch shifter

The phase vocoder doesn't deliver the good quality of speech when used in this real-time setting as it does when the whole signal is shifted at the same time. We therefore present and document another pitch shifting algorithm derived by Stephan Bernsee [2]. This algorithm avoids resampling and delivers a much higher quality level in real time than the phase vocoder.

The algorithm has many similarities with the phase vocoder as it still uses magnitude and phase information in an analysis/synthesis system. Bernsee's C program can be found at his website[2] where he also documents the method. He describes his method very well, but avoids using too much mathematics. We therefore supply a more formal description of his method in this report along with some changes to the code to better fit our AudioUnit code.

In the rest of this section we will refer to Bernsee's algorithm as *smbPitchShift*.

4.2.1 Analysis

In the analysis stage, the stFT of the signal is computed for each frame u, and for each frequency bin k in the frame, the magnitude $A_u(k)$ is found using the formula

$$A_u^a(k) \quad = \quad \sqrt{\Re(x_u(k))^2 + \Im(x_u(k))^2} \tag{42}$$

and the phase $\theta_u^a(k)$ is computed as in (28). Then we compute the true frequency of each bin equivalently to how it was done in the phase vocoder:

$$\Delta\Phi_u^a(k) \quad = \quad \theta_u^a(k) - \theta_{u-1}^a(k)(k) - H\Omega_k \tag{43}$$

$$\Delta_p\Phi_u^a(k) \quad = \quad (\Delta\Phi_u^a(k) + \pi \mod 2\pi) - \pi \tag{44}$$

$$\hat{\omega}_u^a(k) \quad = \quad f_{\text{bin}} \cdot (k + \frac{l \cdot \Delta_p\Phi_u^a(k)}{2\pi}) \tag{45}$$

where $\Omega_k = \frac{2\pi k}{N}$ is the center frequency of the k'th bin and where N is the DFT size and where H is the hop factor (equivalent to R_a in phase vocoder). Formula (43) and (44) are equivalent to formulas (29) and (30) in the phase vocoder.

With an overlap factor l and $f_{\text{bin}} = \frac{\text{sample rate}}{N}$ being the distance between the bins, we can calculate the true frequencies in the frequency domain as in formula (45). This formula can be interpreted as $f_{\text{bin}} \cdot k$ being the frequency detected directly from the bins in the frequency domain, added together with the part $f_{\text{bin}} \cdot \frac{\Delta_p\Phi_u(k)}{2\pi}$ compensating for the smearing introduced by the resolution of the DFT. This is equivalent (but not identical) to formula (31) of the phase vocoder.

4.2.2 Processing

Now that we have found the true frequency on the basis of the phase difference, and we have increased the resolution by using overlapping frames, we can do the actual pitch shifting. Let p be the pitch shifting factor, where $p < 1$ corresponds to a downward pitch shift (lower pitch) and $p > 1$ corresponds to an upward pitch shift (higher pitch). Let $\perp_p^i = \{k \mid \lfloor kp \rfloor = i\}$ be a set of all k for which, given p and an integer i, $\lfloor kp \rfloor = i$. Then the processing part of

smbPitchShift consists of these two formulas

$$A_u^s(i) = \begin{cases} \sum_{k \in \perp_p^i} A_u^a(k) & \text{if } \perp_p^i \neq \emptyset \\ 0 & \text{if } \perp_p^i = \emptyset \end{cases} \quad (46)$$

$$F_u^s(i) = \begin{cases} p\hat{\omega}_u^a(k) & \text{if } i = \lfloor kp \rfloor \\ 0 & \text{else} \end{cases} \quad (47)$$

which determines the synthesis amplitude and the synthesis frequency of the new pitch shifted signal. Formula (47) sums all analysis amplitude values in the synthesis amplitude for all k for which $\lfloor kp \rfloor$ hits the same index, whereas $p\hat{\omega}_u(k)$ simply overwrites the current value at the index.

4.2.3 Synthesis

In the synthesis part, we have to compute the new phase of the shifted signal, and this is done by doing all the steps above in the reverse order. Hence

$$\Delta\Phi_u^s(k) = \frac{F_u^s(k) - f_{\text{bin}} \cdot k}{f_{\text{bin}} \cdot l} \cdot 2\pi \quad (48)$$

$$= \theta_u^s(k) - \theta_{u-1}^s(k) - H\Omega_k \quad (49)$$

Adding $\theta_{u-1}^s(k)$ and $H\Omega_k$ to (49), we get the new phase. So, the formula for getting the new phase is

$$\Phi_u^s(k) = \frac{F_s(k) - f_{\text{bin}} \cdot k}{f_{\text{bin}} \cdot l} \cdot 2\pi \quad (50)$$

$$+ \theta_{u-1}^s(k) + H\Omega_k \quad (51)$$

We now extract the real and imaginary parts by Euler

$$\Re(y_u(k)) = A_u^s(k) \cdot \cos(\theta_u^s(k)) \quad (52)$$

$$\Im(y_u(k)) = j \cdot A_u^s(k) \cdot \sin(\theta_u^s(k)) \quad (53)$$

Taking the IDFT of $\Re(y_u(k)) + \Im(y_u(k))$ yields the signal pitch-shifted by p in the time-domain.

4.2.4 Implementation

Like the phase vocoder, smbPitchShift uses the stDFT to split the incoming signal into frames of relatively small lengths. Because of this we have incorporated the core of the algorithm into our own shell and use our way of splitting the input signal into frames. We also use FFTW[6] instead of Bernsee's own FFT implementation.

5 Tests

In this section we will describe the results we have achieved with the methods just described. We will again split up into pitch detection and pitch shifting and furthermore describe the tests of the individual methods.

5.1 Pitch detection

In this section, we test the precision of three different algorithms, namely the Cepstrum, HPS and the CBHPS. The tests are conducted using the three algorithms on both a female- and a male voice. Both voices are singing a chromatic scale on the vowels 'a', 'eh', 'i' and 'ooh', where the female starts singing a F$^\sharp$ (34) at 184.997 Hz and ends up singing a D$^\sharp$ (55) at 622.254 Hz, and the male starts singing a G (23) at 97.999 Hz and ends up singing a D$^\sharp$ (55) at 622.254 Hz. We only focus on testing common voices, therefore the range from 97.999 to 622.254 Hz. We are testing the precision with respect to octave error and other grave errors[7], and since all errors made by the three algorithms are such errors, the tests gives a trustworthy result. Though, errors within a half tone from the correct pitch is not detected, since this would require some synthesized sounds, which we did not have access to.

5.1.1 HPS

A main flaw of the HPS method is its vulnerabillity to octave errors. In fact almost all of the errors listed for the HPS in Table 2 can be classified solely into this category. However, in most cases these errors are manifested in the pitch being detected one octave above the actual fundamental frequency. This fault can be corrected by using various heuristics; e.g. the following heuristic can be applied with good results [3]:

"IF the second peak amplitude in the HPS emphbelow the determined fundamental frequency is approximately one half of the determined pitch AND the ratio of amplitudes is above a threshold (e.g., 1/5 for 5 harmonics), THEN select the lower peak as the fundamental frequency."

5.1.2 Cepstrum

The errors presented for the Cepstrum in table 2 is not representative for the method, since we could have improved the method dramatically by using some heuristics presented in [9]. Among other things a linear multiplicative weighting can be applied over the cepstral peaks, which aim is to magnify the higher-quefrency components, since these will decay as explained in Section 3.3.1. This could probably increase the precision in the CBHPS, but was not implemented due to lack of time. Another problem is pitch doubling, which can be resolved by a heuristic comparing the preceeding and the following frame[9]. We have no reasons to believe that

[7] errors worse that one octave from the correct fundamental frequency

	HPS	Cepstrum	CBHPS
Woman I	11.1%	21.7%	2.67%
Woman A	0%	12.5%	0%
Woman Ooh	9.76%	20.3%	4.2%
Woman Eh	1.7%	9.5%	1.86%
Man I	15.0%	23.0%	7.7%
Man A	2.5%	4.33%	2.5%
Man Ooh	5.6%	32.3%	9.69%
Man Eh	3.2%	9.4%	5.8%

Table 2: Percentage of wrong detected frames for the HPS, Cepstrum and CBHPS pitch-detection algorithms applied on male and female voices.

these improvements would improve the CBHPS significant, since these errors are cancelled out to a big extent when combining the HPS and the Cepstrum.

5.1.3 CBHPS

When combining the HPS with the Cepstrum, the result achieved is better, even without any post-processing, than the result of using the methods on an individual basis. This is because their respective error-patterns in many cases are neutralized when used together. Yet still, in some cases, the error-pattern of one of them is significant enough to cause the combined method to fail.[8] In these cases, heuristics could be used to improve the accuracy. For example heuristics could be used, as the one mentioned for the HPS, which consider only the particular frame being estimated. But also heuristics which consider preceding frames could be incorporated. If, for example, the detected frequency suddenly drops one octave from one frame to the next, this would probably indicate that the algorithm has failed.

5.1.4 Resolution errors

The resolution error of maximum 12.5 percent is in the limit of what we can tolerate if it should be used for pitch correction, but we have not tested if the error is audible, since it is not trivial to do such perceptual test.

5.2 Pitch correction

We have conducted empirical tests of the perceptable quality of both the pitch shifters.

5.2.1 Phase vocoder

The phase vocoder implementation in the real-time environment suffers from severe artifacts when shifting frames individually. These errors occur as described above because of phase mismatch after resampling. We have tried to improve on the phase vocoder but it turned out to be futile.

5.2.2 Bernsee's method

The Bernsee method was implemented in the Audio Unit and tested empirically. Shifting the pitch works correctly but some reverberation effect is introduced. This is because the method does not ensure vertical phase coherence as described in Section 4.1.4.

6 Implementation

We have implemented both a MATLAB prototype of the pitch corrector described in this paper, but also we have developed an Audio Unit [1]. As MATLAB contains all the necessary tools for signal analysis and graph-plotting this was the obvious choice for prototyping. Hence, each of the described algorithms has first been implemented and tested in MATLAB, before being ported to ANSI C and tried in a real-time context.

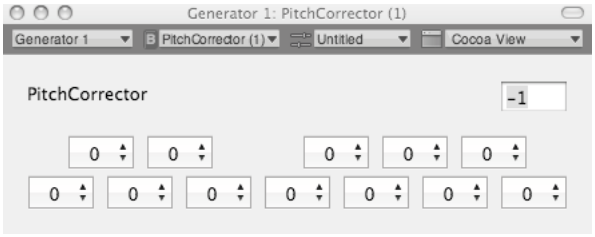

Figure 8: A screen-shot of the Audio Unit

For developing the platform dependent framework of the Audio Unit we have used C++, while all low-level platform independent DSP-code is written in ANSI C. The Audio Unit can be used in a musical host application e.g. as Logic, Cubase or Live. However, by having implemented the pitch corrector as an Audio Unit, we have shown that the descibed pitch corrector can be used in a real-time setting on a normal nowaday laptop in a musical application. In figure 8 a screen-shot of the Audio Unit is depicted. Here, we see the simple interface, where the user can pitch shift different tones in the range of one octave by use of a virtual piano.

6.1 Optimizations

We exploit the fact that the Cepstrum pitch detection algorithm uses the *log power spectrum*; hence the equation for evaluating the LHPS in Section 3.2.1 deliberately is rewritten, so that it takes the *log power spectrum* as its innermost input in the summation sequence. Now, as both the Cepstrum and the LHPS uses the *log power spectrum*, we need to evaluate each of the following three steps only once for each frame

1. apply a Hanning window on the frame
2. calculate the stFT of the windowed frame
3. take the logarithm of the absolute value pointwise of the complex elements in the transformed frame

Futhermore, we have used the free library FFTW3 (The Fastest Fourier Transform in the West) [6] for calculating the stDFT, as this library is available for all major platforms and is indeed fast[9].

7 Conclusion

We have documented and implemented three different methods for pitch detecting, where the CBHPS yields a correct[10] pitch estimation more than 90 percent of the time, which is a fairly good result. The resolution error of maximum 12.5 percent is in the limit of what we can tolerate if it should be used for pitch correction, but we have not tested if the

[8] Surprisingly, in rare cases, when both the HPS and the Cepstrum fails, the combined algorithm succeeds.

[9] For benchmarks see *www.fftw.org/speed*

[10] "correct" in the meaning of no octave error and other grave errors

error is audible, since it is not trivial to do such perceptual test. Neither has the precision of the pitch detection been tested, since this would have required some synthesized sounds, which we did not have access to.

For the pitch shifting, we have implemented two different methods, namely the phase vocoder and Stephan Bernsee's method. We have found out, that the phase vocoder was not applicable in a real-time environment because of the resampling. In the original phase vocoder, the resampling is done for the whole signal in a post-processing step, and trying to do the resampling on the individual frames destroy the phase, and therefore introduces critical errors. We solved the problem by using Bernsee's method, which is essentially built on the same analysis/synthesis approach as the phase vocoder, but is suitable in a real-time implementation because of the lack of resampling. With this method, we achieve a result beyond compare with the real-time phase vocoder, though a little reverberation is still introduced.

With these methods it should be possible to reach our main goal, namely the pitch correction to the nearest half tone. Though, there are some aspects about how fast the pitch should be corrected, since a singer would typically use glissando more or less and the precision of the pitch detection is of course crucial. Therefore, instead of making a pitch correction to the nearest half tone, we have made a real-time correction where a user specifies one or more tones to be shifted to the max of one octave. This implies that the singer should pitch relative precise, and when this constraint is obeyed, the method is working very well with very few errors. With this method working, there is not a long way to go for implementing the pitch correction to the nearest half tone, but we did not have the time to do so. Real-time pitch correction of a voice with respect to an incoming MIDI signal was not meet either, but with a pitch correction to the nearest half tone implemented, this could also be done without great effort.

References

[1] Apple. *Audio Unit Programming Guide*, 2006.

[2] Stephan Bernsee. Pitch shifting using the fourier transform. http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/.

[3] Patricio de la Cuedra, Aaron Master, and Craig Sapp. Efficient pitch detection techniques for interactive music. http://ccrma.stanford.edu/ pdelac/research/MyPublishedPapers/icmc_2001-pitch_best.pdf.

[4] John R. Deller, Jr, John H.L. Hansen, and John G. Proakis. *Discrete-Time Processing of Speech Signals*. IEEE, 2000.

[5] J. L. Flanagan and R. M. Golden. Phase vocoder. *The Bell System Technical Journal*, pages 1493–1509, November 1966.

[6] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[7] Nicolas Juillerat, Simon Schubiger-Banz, and Stefan Mller Arisona. Low latency audio pitch shifting in the time domain. 2008.

[8] Jean Laroche and Mark Dolson. Improved phase vocoder time-scale modifications of audio. *IEEE Transactions on Speech and Audio Processing*, 7(3):323–332, May 1999.

[9] A. Michael Noll. Cepstrum pitch determination. *Journal of the Acoustical Society of America*, 41:293–309, Feb 1967.

[10] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Prentice Hall, 4th edition, 2007.

[11] Lawrence R. Rabiner and Ronald W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, 1978.