REAL-TIME TIME DOMAIN PITCH TRACKING USING WAVELETS

by

Eric Larson

SUPERVISORS:

Dr. Steve Errede

Physics Department

University of Illinois at Urbana-Champaign

Dr. Jan Tobochnik

Physics Department

Kalamazoo College

A paper submitted in partial fulfillment

Of the requirements for the degree of

Bachelor of Arts at

Kalamazoo College

Fall Quarter, 2005

Contents

Acknowledgments

Thanks to R. Maddox for designing and implementing the pitch tracker with me; Profs. J. Tobochnik and S. Errede for advising this project; M. Winkler for helping in the lab with various portions of this project; J. Beauchamp and M. Bay for advice on pitch tracking; University of Illinois at Urbana-Champaign and the National Science Foundation for supporting this project via grant PHY-0243675.

Preface

From June 1st to August 5th, 2005, I participated in the NSF-funded Research Experiences for Undergraduates (REU) program at the University of Illinois at Urbana-Champaign (UIUC). Under the supervision of Dr. Steve Errede and his graduate student Matt Winkler, I worked with a friend and colleague, Ross Maddox, on several projects related to the physics of music. Dr. Errede afforded us a great deal of latitude in choosing a project to work on, so Ross and I decided to start a new project: development of a real-time pitch tracking scheme for use on vocal samples.

We began by researching and familiarizing ourselves with both well-established methods and lesser-known, newer methods of pitch tracking. After coming across Ergun Ercelebi's paper [1], Ross and I decided to build on his work and develop an improved method of pitch tracking based on the fast lifting wavelet transform. Initially, developing a pitch tracker based on the wavelet transform was meant to be a type of personal etude—we were motivated primarily by an urge to familiarize ourselves with the wavelet transform, which neither of us had encountered in our undergraduate educations. Shortly thereafter, we saw that the versatility of the wavelet transform made it particularly useful for our project. The fast lifting wavelet transform developed by Debauchies and Sweldens [2] is well suited to simplify and smooth waveforms, allowing periodicity to be seen more easily (especially in noisy signals). Ross and I saw smoothing/de-noising as a process essential to human recognition of periodicity; this in turn gave us the idea to build our algorithm around techniques qualitatively similar to those used by humans to determine waveform periodicity (e.g. smoothing, peak detection, zero-crossing detection).

We read about the various wavelet transforms (continuous, discrete, fast lifting) and sub-

sequently began building the algorithm. Weeks of testing different methods of constraining period determination—implementing different peak-finding algorithms, frequency limiters based on zero crossings, statistical elimination methods, and so on—yielded a fairly simple yet accurate and reliable method of pitch tracking vocal samples. After we finalized the algorithm, we tested and documented its performance on a set of test signals we designed to test the most important aspects of pitch trackers. We finished our work at UIUC by writing up a report on the project [3] and presenting the results to the REU group.

By the end of our time at UIUC, Ross and I had completed a self-contained project in real-time pitch tracking. We started with an idea, worked through research and development, and completed a functional product. The remainder of this paper documents the more important aspects of this process. Section I details the initial stages of development, including the motivation and relevant background material. Section II explains the inner workings of the algorithm we developed. The results of algorithm testing are given Section III, followed by the Discussion in Section IV and a brief conclusion in Section V.

Abstract

A pitch tracker based on the fast lifting wavelet transform (FLWT) is developed in MATLAB and C++. The algorithm is designed to function on vocal samples with a range of 90–1500 Hz. With under 25 ms latency, RMS error of under 2 cents (i.e. 1/50 of a semitone) on sinusoidal signals through four octaves is achieved. The algorithm is shown to be robust to noise (up to a SNR of 20–25 dB) and missing/weak fundamentals. Analysis is performed in the time domain using the FLWT. The Haar wavelet is used as the basis for the FLWT, which is shown to be mathematically equivalent to splitting a signal into low-pass and high-pass components and down-sampling (generating *approximations* and *details*, respectively). Peak detection on successive wavelet approximations is used determine the pitch of vocals.

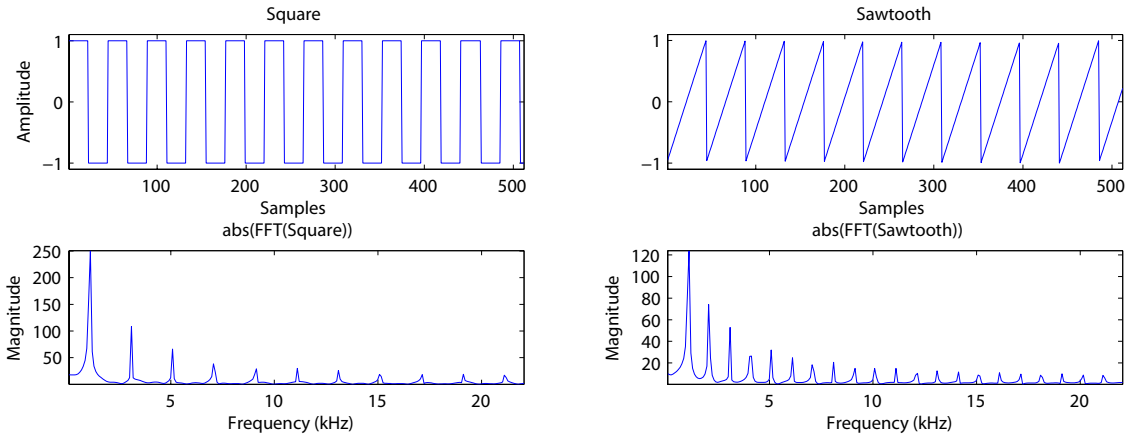

FIG. 1: A square wave and a sawtooth wave, above their respective magnitude spectra derived by the Fourier transform. Although they differ in upper harmonic content, both signals have a fundamental frequency of 1000 Hz, and thus are said to have the same pitch.

I. INTRODUCTION

Sound signals form complex waveforms. Every sound is a combination of sinusoids that vary in frequency, time, and amplitude, and phase. Periodic sounds predominantly consist of a superposition of a fundamental frequency (the root frequency of a periodic waveform) and a set of harmonics/overtones (frequency components that are integer multiples of the fundamental). For the purposes of this paper, the fundamental frequency is taken to be equivalent to the pitch of a periodic signal; when two signals have the same fundamental frequency, they are said to have the same pitch.

The intensity of harmonics generated relative to the fundamental varies across instruments. Depending on several factors—such as size and shape of an instrument, or source of vibration (reeds, vocal cords, strings, etc.)—varied levels of harmonic content are generated. See Figure 1 for an illustration of two audio signals sharing the same fundamental frequency but different harmonic content. Timbre—which is the tone quality (or tone color) that allows us to differentiate between two instruments playing the same note—is determined by this varied intensity of overtones relative to the fundamental. Despite differences in waveform shape and harmonic content, people can easily recognize when different instruments are playing the same note due to differences in timbre. In Figure 1, the square wave has the same pitch as the sawtooth wave, but a distinctly different timbre.

Since the 1950's, the fast Fourier transform (FFT) has been used to study the underlying frequency structure of discrete signals (see Figure 1 for an example of two such FFTs). The FFT is based on the discrete Fourier transform (DFT), which is used to transform discrete signals from the time domain into the frequency domain. The FFT reduces computation time required for signal analysis (compared to the DFT) by reducing the number of trigonometric operations required to transform a signal from the time domain to the frequency domain. The FFT has proved useful in many areas of signal analysis, from determining speech formants to quantitative descriptions of timbre. Although useful for studying harmonic content of signals, the FFT does not determine the fundamental frequency of a signal.

To determine the pitch of a signal as the pitch varies over time, computational schemes called pitch trackers are used. Although most people can readily detect the fundamental of a given signal, it proves difficult to achieve computationally. Monophonic pitch trackers are designed to take a musical signal with one note playing at any given time (monophonic) and transform the signal from amplitude vs. time to fundamental frequency vs. time (pitch tracking). This information, in turn, is useful in music and speech analysis; see the Section II B for details.

Pitch trackers use windowing to determine pitches within discrete signals. For example, given a CD-quality source (44100 Hz, 16 bit), a second-long segment of audio will be 44100 samples. To determine the pitch during this time frame, analyses are done on consecutive windows of the signal; e.g. the pitch of the first 1024 samples is calculated, then that of the next 1024 samples, and so on until the end of the signal is reached. This analysis technique assumes the pitch is constant within a frequency window, but the use of a sufficiently short analysis window can enable intelligent tracking of glissandi (sliding vocal pitches), pitch bends, or other rapidly changing pitches.

In this paper, a new algorithm is developed for use primarily on vocal samples. Focus is placed on 1) low latency (defined as the time between signal onset and fundamental determination), 2) pitch detection accuracy, 3) correct discrimination of voiced and unvoiced sections, and 4) noise robustness. The fast lifting wavelet transform (FLWT) is used to simplify signal analysis. In the following sections, a general background of pitch tracking is provided, and then a detailed account of the original work—developing a new pitch tracking method based on the FLWT—is provided.

A. Background

There are two basic categories of pitch tracking methods: frequency domain and time domain. Frequency domain analysis utilizes Fourier analysis to transform a window of a signal from amplitude vs. time to amplitude vs. frequency and compute a frequency using the Fourier components. Time domain analysis is performed on the window of the signal without transforming it to the frequency domain, performing calculations on the original signal to determine the pitch.

One of the best established methods of pitch tracking is autocorrelation, which uses the autocorrelation function (ACF) to determine the pitch of a signal. As stated by de la Cuadra [4], given a discrete audio signal $\mathbf{x}(n)$ and window length N, the ACF is given by

$$\phi(\tau) = \frac{1}{N} \sum_{n=0}^{N-1} \mathbf{x}(n)\mathbf{x}(n+\tau). \tag{1}$$

Pitch tracking via the ACF works by 1) windowing a signal; 2) taking a smaller portion of the windowed signal; 3) multiplying this smaller portion by the signal values at offset distances τ from it; 4) summing these products and dividing by the length N of the portion; and 5) using the offsets at which the peak(s) of this function occur to determine the period of the signal. The locations of the peaks of the ACF indicate what the period of the signal is, yielding the frequency.

A similar pitch tracking method uses the average magnitude difference function (AMDF). This operates in the same manner as the ACF, but by using average magnitude differences, reduces computation time. The valleys (minima) of the function are sought instead of the peaks, as the minima tend to occur at the fundamental period of the signal. The equation for the AMDF is given by

$$\psi(\tau) = \frac{1}{N} \sum_{n=0}^{N-1} \left|\mathbf{x}(n) - \mathbf{x}(n+\tau)\right|. \tag{2}$$

These two functions can also be combined to form a more robust method of pitch tracking, which tends to balance out frequency doubling and frequency halving characteristics of the two methods (de la Cuadra [4] provides more information on this).

Probably the most popular method of pitch tracking is the cepstrum, or cepstral method. A rearrangement of the word "spectrum," cepstrum is based on the FFT. Cepstrum is performed by 1) calculating the FFT of a window of a signal; 2) taking the logarithm of

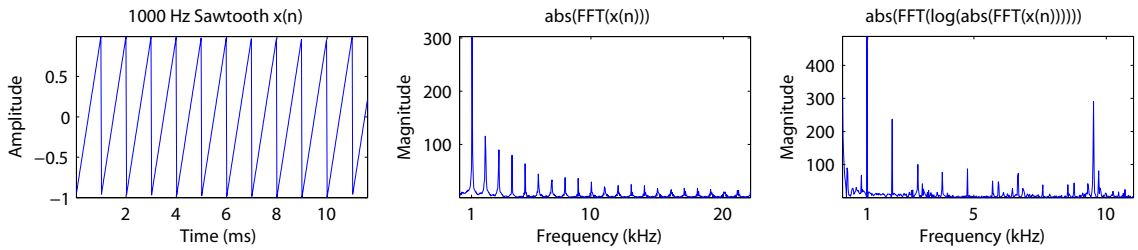

FIG. 2: The cepstral method takes a signal (the first plot of a 1000 Hz sawtooth wave), computes the magnitude spectrum (the second plot, the FFT of the signal), and then analyzes the magnitude of the FFT of the logarithm of the magnitude of the FFT of the original signal (the third plot), seeking the first dominant peak. From the third plot, the frequency of the original signal is seen to be 1000 Hz, with the remaining peaks intelligently disregarded (e.g. the peak around 10khz is an aliasing artifact).

the resulting magnitude spectrum; 3) taking an FFT of the result; and 4) analyzing the peaks of this FFT-FFT magnitude spectrum to extract the underlying periodicity of the signal. See Figure 2 for a depiction of this process. This method relies upon the periodicity of harmonics to generate periodicity in the first FFT, which then leads to pronounced peaks in the second FFT.

While cepstrum has become an established method of pitch tracking and generally produces accurate results, it is limited in several ways due to its implementation two Fourier transforms. First, computing two FFTs is computationally expensive. Second, the Fourier technique has poor low-frequency resolution, and therefore is limited in its ability to determine low frequencies accurately. Finally, the FFT-FFT technique requires that the signal contains significant upper harmonic content, so the technique fails on sinusoidal signals (the magnitude spectrum of the first FFT will be a single peak, which becomes meaningless under an additional FFT). Despite these drawbacks, the cepstrum performs well on natural signals, and is a popular choice for use in signal analysis.

II. METHOD

The algorithm developed here is designed to provide a faster, more accurate method of pitch tracking. It is based on the idea that, when visually examining a window of a periodic

musical signal, people can readily determine the period of the signal a large majority of the time. Our algorithm is designed to do the same; by employing the fast lifting wavelet transform (FLWT) to simplify waveforms and then applying an intelligent peak-finding method, the period (and hence the frequency) should be accurately determined by the computer. By keeping the mathematics behind the computations simple, we reduce latency and improve response time.

To keep computation time short but still manipulate signals in a useful way, this pitch tracker uses the wavelet transform. The wavelet transform is known for allowing a compromise between accuracy in the time domain and accuracy in the frequency domain. In this setting, this is useful because it enables the algorithm parameters to be tweaked to suit our needs. With the development of the fast lifting wavelet transform, computation time of the transform is significantly decreased by decreasing the number of multiplication and addition operations required from the computer. This allows us to use the wavelet transform to simplify signals with only a small latency increase.

A. Wavelets

The wavelet transform is similar to the Fourier transform in that it breaks a signal down into components parts. Instead of using infinite-length sinusoids and assuming constant periodicity, the wavelet transform breaks the signal down into a set of scaled and shifted mother wavelets that vary in time. Each wavelet transform is based on a given mother wavelet of finite length. See Figure 3 for an image of the Haar wavelet, which serves as the mother wavelet in the wavelet transform used in this algorithm.

The continuous wavelet transform (CWT) is based on the wavelet function, which is given by [5]

$$\psi_{s,\tau}(t) = \frac{1}{\sqrt{|s|}}\psi\left(\frac{t-\tau}{s}\right). \tag{3}$$

In this equation, ψ is the equation of the mother wavelet, which serves as the basis of the wavelet transform as discussed above. From this wavelet equation, for a given signal $\mathbf{x}(t)$

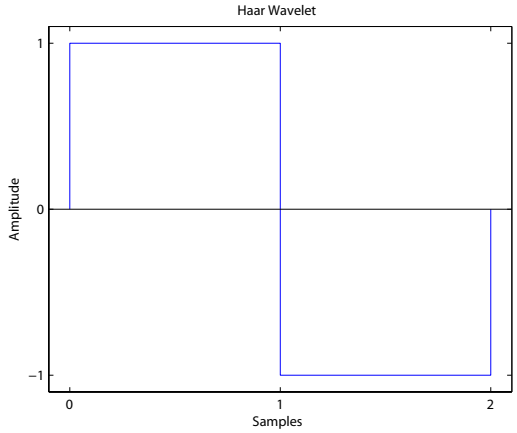

FIG. 3: An Amplitude vs. Samples image of the Haar wavelet. The fast lifting wavelet transform used in this algorithm is based on the Haar wavelet, which is shown to be mathematically equivalent to applying a low-pass filter and down-sampling to generate an approximation signal, and applying a high-pass filter and down-sampling to generate a detail signal.

we determine a set of wavelet coefficients $\mathbf{c}_{\mathbf{s},\tau}$ as

$$\begin{aligned} \mathbf{c}_{\mathbf{s},\tau} &= < \psi_{\mathbf{s},\tau}, \mathbf{x} > \\ &= \sum_t \psi_{\mathbf{s},\tau}(t)\mathbf{x}(t) \\ &= \frac{1}{\sqrt{|s|}} \sum_t \psi \left(\frac{t-\tau}{s} \right) \mathbf{x}(t). \end{aligned} \tag{4}$$

This set of coefficients provides the forward CWT. To get the original signal back, we combine Eqn. 3 and Eqn. 4 to derive the inverse CWT, which is given by

$$\mathbf{x}(t) = \sum_s \sum_\tau \mathbf{c}_{\mathbf{s},\tau}\psi(t). \tag{5}$$

Thus we arrive at the CWT, which decomposes a signal into a set of wavelet coefficients based on translations and dilations of the original mother wavelet.

In discrete time domains, this process can be simplified to create the discrete wavelet transform (DWT). The DWT reduces computation time by reducing the number of scales considered. With the DWT, a discrete signal $\mathbf{x}(\mathbf{n})$ of length N is split up into a set of approximations and details. To do this, first $\mathbf{x}(\mathbf{n})$ is low-pass filtered and down-sampled to produce the first approximation $\mathbf{a_1}(\mathbf{n})$, and $\mathbf{x}(\mathbf{n})$ is high-pass filtered and down-sampled to produce the first detail $\mathbf{d_1}(\mathbf{n})$. $\mathbf{a_1}(\mathbf{n})$ and $\mathbf{d_1}(\mathbf{n})$ both have $N/2$ points. Wavelet coefficients

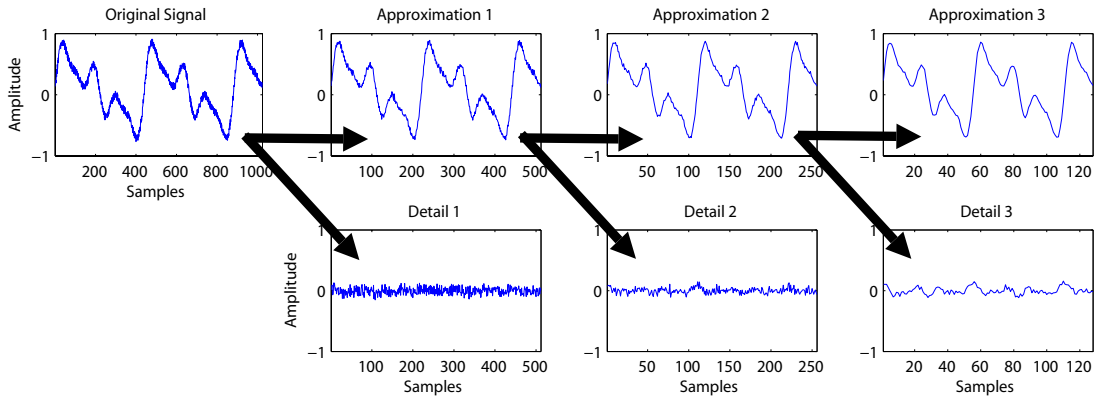

FIG. 4: A visual representation of the dyadic tree generated by the wavelet transform. An original signal is split into an approximation and a detail, and successive wavelet transforms can be applied on the approximations to generate additional levels of the transform.

are extracted from $\mathbf{d_1(n)}$. The process repeats (indefinitely) by taking $\mathbf{a_1(n)}$ and splitting it up into $\mathbf{a_2(n)}$ and $\mathbf{d_2(n)}$ by low-pass and high-pass filtering and down-sampling, followed by extracting wavelet coefficients from $\mathbf{d_2(n)}$. This process can repeat indefinitely (limited by N). See Figure 4 for a visualization of this process as it occurs with the Haar wavelet.

After the CWT became established, work began on the second generation of wavelet transforms, also known as fast lifting wavelet transforms. For an account of the derivation of the FLWT, we refer you to the self-contained, extensive, and complete work by Daubechies and Sweldens [2]. From their work, we arrive at the equations of the FLWT transform based on the Haar wavelet. Given an original signal $\mathbf{x}(n)$, an approximation signal $\mathbf{a_1}(n)$ and a detail signal $\mathbf{d_1}(n)$ are given by the Haar Wavelet FLWT as

$$\begin{aligned} \mathbf{d_0}(n) &= \mathbf{x}(2n+1) \\ \mathbf{a_0}(n) &= \mathbf{x}(2n) \\ \mathbf{d_1}(n) &= \mathbf{d_0}(n) - \mathbf{a_0}(n) \\ \mathbf{a_1}(n) &= \mathbf{a_0}(n) + \frac{\mathbf{d_1}(n)}{2}. \end{aligned} \tag{6}$$

Note that with each FLWT, the number of samples in the approximation signal, as well as in the detail signal, is cut in half. In this way, a signal is split up into a representation (approximation and detail) with length equivalent to that of the original signal. Repeated applications of the FLWT can then be performed by taking the approximation $\mathbf{a_1}(n)$ as the original signal, and generating a second approximation $\mathbf{a_2}(n)$ and detail $\mathbf{d_2}(n)$ from it,

leaving the first detail $\mathbf{d_1}(n)$ half the length of the original signal $\mathbf{x}(n)$, and the second approximation $\mathbf{a_2}(n)$ and second detail $\mathbf{d_2}(n)$ each a quarter of the length of the original signal $\mathbf{x}(n)$. In theory, this process can be repeated indefinitely, to the point at which the final approximation and detail signals have length one; in practice, only a few levels of wavelet transform get used. See Figure 4 for a visual representation of this process. Additionally, the computations required to perform the FLWT using the Haar wavelet are simple, which means that computation time required for analysis will be low.

It is interesting to note that performing the FLWT with the Haar wavelet on a signal is mathematically equivalent to performing a low-pass operation followed by down-sampling on the signal to produce the approximation signal, and performing a high-pass operation followed by down-sampling on the signal to produce the detail signal. See Appendix B for details. For the purposes of the pitch tracker, this will prove useful because repeated low-pass operations will help to reduce noise levels in the signal being analyzed, thus helping determine the pitch.

The algorithm developed in this paper is similar to that developed by Ercelebi [1]. However, there are several important differences between the FLWT-based algorithm of Ercelebi and the one presented here. Distinct improvements made by this algorithm are 1) a more robust peak-detection algorithm (accompanied by a more explicit description of our method); 2) a different wavelet level decision-making scheme; 3) a transient detector that marks the windowed signal as unpitched if the root mean square (RMS) amplitude of the first third is more than four times that of the last third (or vice versa); as well as 4) a mode-averaging method that increases frequency resolution, especially in the high frequencies.

B. Motivation

Pitch tracking has many applications in audio-related research. Today, there are projects underway to enable singing-based retrieval of songs from within music libraries [6, 7], sometimes referred to as "query by humming." To locate a song within a music library, a melody is sung into a microphone, and monophonic pitch tracking is performed to extract frequency versus time information (see Figure 5). This is used to compare against the known pitch contours of library songs to determine a match. Monophonic pitch tracking is also used for automatic transcription of certain musical performances [8]. For vocalists, pitch tracking

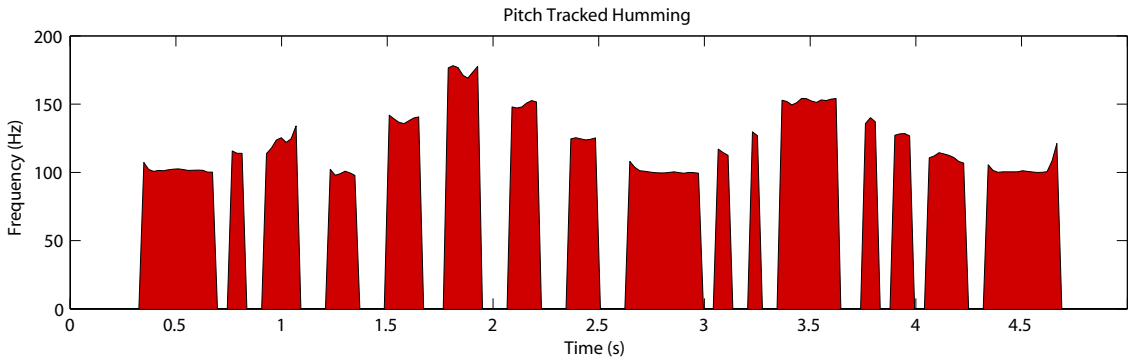

FIG. 5: Pitch tracking takes an amplitude vs. time signal and turns it into fundamental frequency vs. time (pitch tracking here done by our algorithm). The vocalist is humming the beginning melody of a Bob Marley tune, which could, for example, then be used (in [6, 7]) to find the original tune, "Redemption Song," within a music library.

can provide useful feedback for intonation and singing techniques, enabling detailed analysis and documentation of pitch characteristics, such as vibrato and bends.

Developing a robust, accurate method of real-time pitch tracking can aid in the training of singers. Providing feedback in real-time allows performers to see how accurately they are singing, helping them to make appropriate adjustments on the fly, overcome bad habits, and develop improved technique by seeing exactly what they are doing while they are doing it. For these purposes, having a real-time pitch tracker with low latency is useful—the shorter the latency, the easier it is to adjust singing appropriately.

Speech analysis researchers utilize pitch tracking to better understand how pitches are used in communication. For example, in the English language, speakers use tonal inflections to cue listeners into the meanings of their utterances (e.g. to indicate that the expression is interrogative rather than assertive). Having a clear, reliable method of extracting these tonal patterns helps us understand how tonality is used in speech. While simple spectral analysis is useful for understanding certain aspects of speech (such as determining speech formants), pitch tracking provides a clearer, more detailed picture of how pitch changes within speech.

C. Algorithm

Speech analysis occurs on consecutive 1024-sample segments of an input sampled at 44100 Hz. The algorithm is developed in both C++ and MATLAB environments. C++ development allows the use of real-time (and low-latency) processing via the RT-Audio plug-in developed by Gary Scavone [9]. Using RTAudio, a 1024-sample window of an input is passed to the algorithm after all 1024 samples have been buffered by the sound card. These samples can then be processed in real-time. For consistency's sake, in MATLAB WAV files are read in as arrays and analyzed in consecutive 1024-sample segments. Although this does not allow for real-time processing of musical signals, it provides an easy method of visualizing (and debugging) the algorithm developed in parallel in C++. The MATLAB implementation of the algorithm is included in Appendix C for the reader's reference.

The algorithm described below is designed to work after this 1024-sample windowing has already occurred; different window sizes can be used, but 1024 samples was selected to provide low latency and good time resolution without sacrificing low frequency response. Since a minimum of two periods must fit within a window for this algorithm to analyze it correctly, 1024 samples implies a pitch detection frequency response floor of approximately 90 Hz. Increasing the window size decreases this floor, but increases latency (due to increased buffering time).

Once a signal (understood from here on as a 1024-sample windowed potion of the original signal) is passed to the pitch detector, the first voiced/unvoiced disqualification occurs. If the RMS amplitude of the first third of a signal is greater than four times (or less than one quarter of) the RMS amplitude of the last third, then the signal is assumed unvoiced and no pitch analysis takes place. (In the included MATLAB code, this step is not included because it is part of the windowing function.) This step is necessary to eliminate transient segments of signals; since the analysis window is very short, the elimination of transients in this way serves to reduce miscalculation at the onset and end of pitched segments without substantially degrading voiced/unvoiced judgment.

Once this windowing and initial disqualification occurs, the algorithm proceeds according to Table I. The following subsections detail the function of the more important parts of the algorithm.

TABLE I: Pitch Tracking Algorithm

Taking $i = 1$ as an initial condition,

1. Perform the FLWT on the window to generate approximation i, denoted $\mathbf{a_i}$.
2. Find the maxima (minima) of $\mathbf{a_i}$ above (below) a level-dependent threshold.
3. Find the center mode distance ($\mathbf{CMD}(i)$) between maxima (minima) samples.
4. Use averaging to improve the value of $\mathbf{CMD}(i)$.
5. If $\mathbf{CMD}(i) = \mathbf{CMD}(i-1)$, assume $\mathbf{CMD}(i-1)$ is the period of the signal.
   Otherwise, increment i. If i is below a specified level limit, return to step 1.
   Otherwise, assume the window is unpitched (no matching $\mathbf{CMD}(i)$'s found).

1. Peak Finding

The peak finding algorithm uses two ideal properties of periodic waveforms to find legitimate peaks. Following the first derivative test of introductory calculus, the first principle is that a given peak must occur at a point where the first derivative changes sign. Since we are dealing with discrete signals, equivalently, an extreme value must be a point of change of the first difference sign. Therefore, if the first difference does not change sign from positive to negative (or negative to positive), then point of change is not a local maximum (or minimum).

The second property of periodic waveforms used is zero crossings. The algorithm relies on the idea that a periodic waveform will not have more than one extreme value (maximum or minimum) between zero crossings. To compensate for a possible DC offset within a window, any time a "zero crossing" is referred to here, in the code it is actually implemented as a DC-offset crossing.

Combining these two concepts yields an intelligent peak finder. A sample is considered a local maximum (or minimum) if it is the first point to pass the first derivative test with a value above (or below) a certain threshold M of the maximum absolute value of the windowed signal. By setting the threshold sufficiently high, the pitch tracker tends to find only maximum values close to the true peaks of the periodic signal. The zero crossing detector helps to eliminate additional extraneous peaks caused by noise or upper harmonics.

2. Period Determination

Once the peaks are found, their indices are used to determine the period of the waveform. First, the distances between indices of the maxima and distances between indices of the minima are calculated. Distances are calculated between nearest neighbors, next-nearest neighbors, and so on up to a separation of N_d indices apart. For example, for $N_d = 3$, the distance between the first index and the second, the third, and the fourth indices will be recorded, as well as those between the second and third, the fourth, and the fifth, and so on.

From the set of all combined maxima and minima differences $\mathbb{D}$, the center mode distance of a transform level i (denoted $\mathbf{CMD}(i)$) is used to determine the period. The $\mathbf{CMD}(i)$ is, loosely, the distance which occurs most often. More specifically, the $\mathbf{CMD}(i)$ is the distance which has the most other distances within a small δ neighborhood of it. The small neighborhood δ a user-specified tolerance parameter related to the maximum frequency parameter F by

$$\delta = \mathbf{max}\left(\left\lfloor \frac{F_s}{2^i F} \right\rfloor, 1\right), \tag{7}$$

Where F_s is the sample rate and i is the current level of the wavelet transform. In practice, $F = 3000$ Hz and $F_s = 44100$ Hz yields $\delta = 7$ samples at the first level ($i = 1$) of the wavelet transform. This tolerance is important because the peaks will not always line up exactly, and will not always yield equal differences.

In addition to this tolerance, a special case arises when two different differences have the same (or within one) number of similar distances. In this case, the previously determined frequency is taken into account—this feedback technique is the first way of resolving discrepancies of this sort. Failing a rough equivalence between the previous analysis period and one of the two tied distances, the larger of the two modes is taken if it is twice the length of the smaller (to bias toward frequency halvings instead of doublings). This method is used because the algorithm had a slight bias toward doublings which was corrected by this measure.

Once the CMD of level i ($\mathbf{CMD}(i)$) is found, an averaging scheme is employed to improve the estimation of the period. The $\mathbf{CMD}(i)$ is taken to be the mean of the set of all distances in the δ neighborhood of $\mathbf{CMD}(i)$. If $\mathbf{CMD}(i) = \mathbf{CMD}(i-1)/2$, then $\mathbf{CMD}(i)$ is taken to

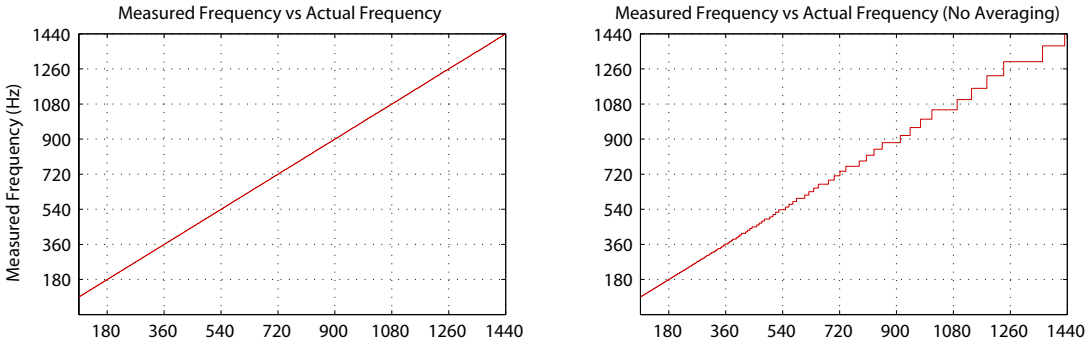

FIG. 6: A plot of the accuracy of the pitch tracker on sinusoidal signals with the mode averaging scheme employed, and without. Note that in the high frequencies averaging improves accuracy, reducing some errors by approximately 100 Hz.

be the period of the segment. Division by two is necessary here to compensate for the half down-sampling that occurs with each successive wavelet transform. Similarly, multiplying $\mathbf{CMD}(i)$ by 2^i yields the signal period and thus the frequency. In the case that $\mathbf{CMD}(i) \neq \mathbf{CMD}(i-1)/2$, the algorithm increments i and starts back with the wavelet transform and peak detection. However, the incrementation of i has an upper limit of N levels; once this is exceeded, the algorithm reports the segment as unpitched.

The averaging scheme employed to improve the estimation of $\mathbf{CMD}(i)$ helps to improve the accuracy of the algorithm in the upper frequencies where a small integer variation in distance yields a large variation in reported frequency. See Figure 6 for a visual representation of this improvement. A signal of sinusoids (of constant frequency within each window) was constructed, ranging from $90 - 1440$ Hz in 1 Hz increments. Without averaging, the pitch tracker made gross errors on the order of 100 Hz around 1400 Hz, while the averaging scheme kept the analysis correct throughout the range. A more rigorous examination of sinusoidal signal accuracy is provided in the results section.

III. RESULTS

The pitch tracker's performance was tested on several different signals. Throughout testing, the parameters of the pitch tracking algorithm were as in Table II. These parameters were determined to qualitatively yield the best performance.

TABLE II: Pitch Tracker Parameters

Maximum Frequency F	Difference Levels N_d	Maxima/Minima Threshold M	FLWT Levels L
3000 Hz	3	0.75	6

The four major components tested were 1) latency, 2) voiced/unvoiced detection, 3) accuracy, and 4) noise robustness. Latency is important for real-time applications, and is treated here geared toward that context. Voiced/unvoiced decisions are important for the correct operation of a pitch tracker. Accuracy is perhaps the most important aspect, where detail in both the frequency and time domains is desirable for optimal analysis. Finally, robustness to noise is important for real-world applications, where noise in signals is always an issue.

A. Latency

Latency is, loosely, how long it takes a pitch tracker to determine the pitch of an audio signal. More precisely, the latency is the amount of time between the onset of a pitched signal to the output of pitch by the pitch tracker. The latency of our pitch tracker is limited by two factors: the window size and computation time. Since a window size of 1024 samples of 44100 Hz signals is used in analysis and analysis does not take place until the entire window is buffered, a minimum latency of $1024/44100 = 23$ ms is created before computation occurs. The second factor contributing to latency is computation time; after the signal is buffered, computations must be performed to determine the pitch.

Maintaining low latency is important for real-time pitch tracking applications. In musical performances, latencies above 10 ms are noticeable, and latencies above 30 ms are distracting to performers. For example, if a pitch tracker were used to provide real-time accompaniment, latency under 30 ms would be essential for musical continuity.

Our algorithm is designed to limit the amount of computational overhead that contributes to latency. The FLWT keeps the number of required arithmetic operations small, thereby reducing latency. In MATLAB, tests on a computer running a 3.2 GHz processor averaged around 4 ms of computation time per analysis window. The C++ implementation of the

TABLE III: Comparative Pitch Tracker Latency Ranges

	AXON (ms)	ZETA (ms)	fiddle~ (ms)
Open E pizzicato	16.0-19.6	17.4-24.7	16.2-27.4
Open E legato	16.2-31.8	17.5-42.3	17.2-45.0
Open G pizzicato	23.9-29.8	31.9-46.4	25.9-48.1
Open G legato	25.4-55.0	38.7-67.0	26.8-96.9
Average latency	16.0-55.0	17.4-67.0	16.2-96.9

algorithm was significantly faster, and did not increase latency time significantly beyond the buffer time; this yields a total latency around 24 ms.

Table III lists latencies of three of the fastest pitch trackers available: the hardware-based AXON, hardware-based ZETA, and software-based and fiddle~ (table taken from Yoo and Fujinaga [10]). Note that the performance of the algorithm presented here—with a 24 ms latency across the frequency range—generally performs better than both the hardware and software methods of pitch tracking in the study. AXON, ZETA, and fiddle~ all have variable latencies that increase with lowering pitch, with a majority of latencies greater than the uniform 24 ms latency of our algorithm. This suggests that our algorithm can outperform some of the best software and hardware pitch trackers available.

B. Voiced/Unvoiced Decision

Figure 7 presents a plot of the pitch tracker's analysis of a female vocal sample of length 26 s. In this sample, there are three unvoiced-to-voiced errors and one voiced-to-unvoiced error in approximately 1000 analysis windows. This yields a total voiced/unvoiced error rate of 0.4%, an unvoiced-to-voiced error rate of 0.3%, and a voiced-to-unvoiced error rate of 0.1%. Although no other plots or data of voiced/unvoiced performance are included in this paper, this was representative of the general performance of the pitch tracker on all other signals of comparable quality tested against the algorithm.

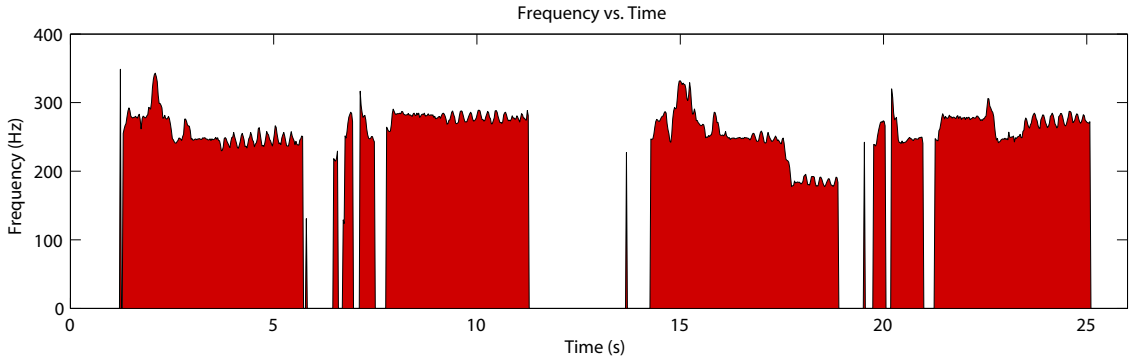

FIG. 7: Frequency vs. Time for a 26-second sample of female vocals. One voiced-to-unvoiced and three unvoiced-to-voiced errors occurred (approximate 0.4% error rate). Note that the good time and frequency resolution of the pitch tracker reveal vibrato (toward the end of sustained notes) and vocal trills.

C. Accuracy

A qualitative account of the accuracy of the pitch tracker can be made on the analysis in Figure 7. Time resolution of 23 ms can be seen in the voiced/unvoiced errors, which are one sample window (23 ms) wide. Frequency resolution limitations cannot be seen, but details about singing technique are revealed by this plot. Toward the end of notes, vibrato is used by the vocalist. In several places, brief trills are also revealed by the pitch tracker.

A quantitative analysis of the pitch tracker's accuracy is provided via a simple test on a signal of sequential sinusoids. Sinusoids ranging in frequency from 90–1440 Hz in 1 Hz increments were sequentially added to create a long signal, with constant frequency within each analysis window (1024 samples). To test the pitch tracker's accuracy, this signal was pitch tracked and the RMS error in both Hz and cents was recorded. (Note: cents are the unit in musical intervals representing 1/100 of a semitone; 1 cent $= 1200 \log_2(f_2/f_1)$.) See Figure 8 for a plot of 1) RMS error (Hz), 2) RMS error (cents), 3) maximum error (Hz), and 4) mean error (Hz) vs. octave band.

To test the pitch tracker's performance on more challenging synthesized signals, a missing fundamental test was performed. In the case of a missing fundamental (but present upper harmonics), the pitch tracker should be able to correct for the missing fundamental and accurately asses the pitch. To test this, for each frequency from 90–720 Hz (in half octave increments), a signal comprised of two adjacent harmonics was generated. This signal was

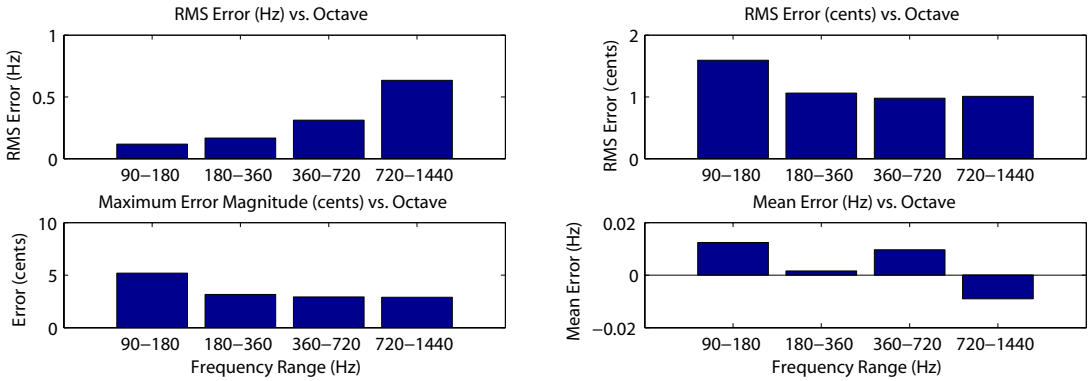

FIG. 8: Several plots detailing the RMS errors of the pitch tracker across octave bands on a synthesized sinusoidal signal. In the first plot, the RMS error (Hz) increases approximately exponentially, which leads to a approximately constant, very low RMS error (cents) across octave bands (second plot). The third plot shows that the maximum error in any octave band was approximately 5 cents, which is around the just-noticeable difference. The fourth plot shows that errors are centered around zero.

TABLE IV: Missing Fundamental First Error Frequencies

Fundamental f_1 (Hz)	90	127	180	255	360	509	720
Lower Harmonic (LH) f_n (Hz)	720	1018	1620	1527	1440	1527	1440
LH Number n	8^{th}	8^{th}	9^{th}	6^{th}	4^{th}	3^{rd}	2^{nd}

then pitch tracked to determine if the pitch tracker was capable of estimating the correct frequency despite the missing fundamental (and missing other harmonics). The initial harmonic number began at two (so the first signal analyzed was the second harmonic added to the third), and was incremented by one until the test ended at harmonic number n (of frequency f_n Hz) when the pitch tracker failed to recognize the correct fundamental pitch. See Table IV for the tracker's performance.

To provide an example of this signal analysis, we examine the 90-Hz case. First, a signal of a 180 Hz sine wave added to a 270 Hz sine wave was generated and pitch tracked. The pitch tracker reported a fundamental frequency of 90 Hz, so a signal of 270 and 360 Hz was created, and so on until the pitch tracker failed to determine the fundamental as 90 Hz—which occurred at $n = 8$ and $f_n = 720$, or on a signal of 720 Hz added to 810 Hz.

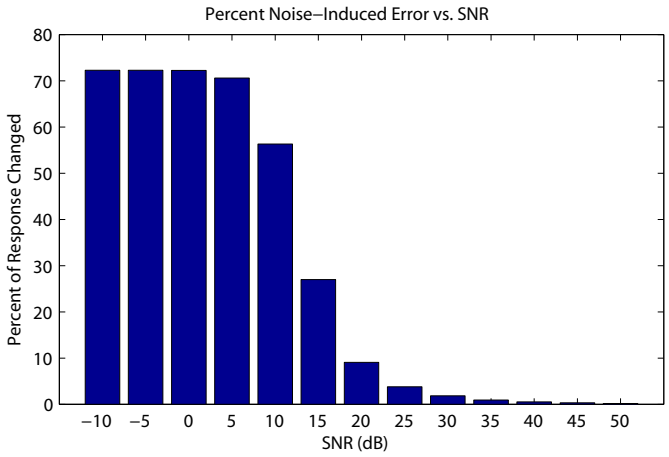

FIG. 9: Performance of the pitch tracker on a signal with additive white Gaussian noise with SNRs of −10 to 50 dB. After pitch tracking the original signal, errors were called whenever the noisy-signal analysis differed from the original-signal analysis by more than 5%. The percentage of induced errors drops off sharply around 20 dB, where the error is below 10%. The percent induced errors has an upper limit around 70% because the original signal was only 70% voiced, and higher the noise levels increase unvoiced decisions in this algorithm.

D. Noise Robustness

An important quality of a pitch tracker is its ability to deal with noise. This pitch tracker was tested against noise using a female vocal sample of 26 seconds. After running an initial analysis on the signal (which found approximately 70% of the signal to be voiced), white Gaussian noise (WGN, which is noise which follows a Gaussian distribution of equal power across the frequency spectrum) was added to the signal at signal to noise ratios (SNRs) ranging from −10 to 50 dB in 5 dB increments. For each dB SNR, 100 iterations were performed of 1) taking the original signal and adding WGN at the specified SNR, 2) pitch tracking the signal, and 3) determining the percentage of windows where the original pitch and the new pitch differed by greater than 5%. One hundred iterations were used to ensure statistical accuracy (since additive white Gaussian noise is randomized), and the average percentage change in analysis was determined.

After the required 1301 pitch tracking runs, the plot in Figure 9 was obtained. It is important to note that adding WGN had an upper threshold of error equal to the amount of sections originally determined to be voiced. This tells us that, with the addition of noise, an

increasing number windows are determined unvoiced. Since the original analysis said 70% was voiced, the upper threshold of error was 70%.

IV. DISCUSSION

A. Latency

The pitch tracker did well in terms of latency. While a shorter analysis window could be used to decrease latency (by lowering the 23 ms buffering latency floor), low-frequency extension of the pitch tracker would be sacrificed. This algorithm requires at least two periods to be present within an analysis window to work correctly; a 1024-sample window implies a lower bound on frequency of $44100/(1024/2) \approx 86$ Hz. Increasing the window size by a factor of two drops the low frequency bound to about 43 Hz, but doubles buffering latency, increases analysis time requirements, and halves the time resolution. Shortening the analysis window size, on the other hand, decreases latency but reduces low frequency extension, which would be impractical for many vocalists.

Given the general speed limitations of MATLAB as an interpreter, the performance of the pitch tracker in MATLAB was surprisingly good. 4-ms computation time per analysis window (and a total latency around 27 ms) is low. The latency of 24 ms in C++ means very good pitch tracking response in real time.

B. Voiced/Unvoiced Decision

The voiced/unvoiced decision was perhaps the pitch tracker's weakest performance category. The error rate was under 1%, which makes the performance acceptable but not exceptional. A lower voiced/unvoiced error rate would be desirable. For the purposes of real-time singing, though, the fact that three of the four errors recorded were unvoiced to voiced errors (finding a pitch when there isn't one) is positive; for voice-training purposes, for example, finding a pitch when there isn't one is less a problem than finding no pitch during a sung section.

C. Accuracy

The ability of the pitch tracker to resolve singing characteristics in Figure 7 suggests that the pitch tracker has very good accuracy in both frequency and time domains, allowing accurate visual reproduction of the vocalist's performance. This indicates that this algorithm would be good for vocal training and analysis.

From the four plots of Figure 8 (error vs. frequency band plots) a few characteristics of the pitch tracker are clear. From the first plot, it appears as though the RMS error (Hz) increases exponentially as the octave is increased. This suggests that, in terms of musical intervals, the RMS error will be approximately constant, since musical intervals increase exponentially, as well. This is in fact what is seen in the second plot of RMS error (cents) vs. octave; the error in cents is nearly constant (and under 2 cents) across all octave bands.

The third plot in Figure 8 shows the maximum error (cents) achieved at any frequency within each octave band. From this we see that the maximum error of all of the octave bands was around 5 cents. Since the just noticeable difference (JND) in terms of pitch for humans is around 5 cents [11], the pitch tracker keeps all of its pitch estimations at or below the JND across its frequency range.

In the fourth plot of Figure 8, the mean error is clearly centered around zero with very small deviations. This tells us that the pitch tracker is unbiased—that is, it has no systematic inclination toward over- or under-estimating the pitch of a signal. From the four plots in Figure 8, we see that, across the frequency range tested, pitch tracking error in terms of musical intervals is uniformly small (RMS error around 2 cents with maximum of 5 cents) and unbiased. This suggests the pitch tracker will have uniformly accurate performance across its frequency range on natural signals.

From the missing fundamental test, performance in the 90-Hz case was very good, ending only on the eighth/ninth harmonic pair. In practice, it will be extremely rare to encounter a signal with no fundamental and no harmonics up to the eighth/ninth position, so failure at this level is not unexpected and does not reflect poor performance. Overall, performance at all frequencies was exemplary, with failures occurring not prior to the eighth/ninth pair or when the second of the two harmonics exceeded the practical 1440 Hz frequency limit of the pitch tracker (e.g. the 360 Hz case failed when the lower harmonic was the fourth harmonic, 1440 Hz, implying the upper harmonic was 1800 Hz, which is too high a frequency). The

pitch tracker performed very well on the missing fundamental test across its frequency range, suggesting that it is robust to missing fundamentals in natural sounds. This means that in signals with a weak fundamental but multiple upper harmonics (at least one even and odd), the pitch tracker will detect the correct pitch.

D. Noise Robustness

It is clear that the pitch tracker performed well in noisy situations, staying under 10% induced error up to 20 dB SNR. Under 25 dB SNR, the response of the pitch tracker remained well under 5% induced error. A signal with 25 dB SNR is noticeably noisy and, in general, has more noise than signals will have in practice. This high level of performance in the case of noisy signals suggests that the pitch tracker is well-suited to deal with a large range of signals, even those of reduced quality with undesirable noise characteristics.

V. CONCLUSION

In this paper we presented a pitch tracker based on the fast lifting wavelet transform that is low-latency, accurate, and robust to noisy signals. Although performance in voiced/unvoiced decision making was limited, it was acceptable for the purposes of real-time vocal pitch tracking. Overall performance of the pitch tracker was very good, and suggests that this algorithm is well-suited for real-time vocal pitch tracking applications, especially in cases where low latency as well as good time and frequency resolution are desired.

For future work, this pitch tracker could be developed further into a very good non real-time pitch tracker. Since the focus in this paper was on real-time performance, several options available to non real-time pitch trackers were not explored. One option is to implement a global maximum-based threshold for voiced/unvoiced decisions. When this was tested in the MATLAB code, preliminary results were positive. A second possibility is the addition of a smoothing function for the frequency vs. time curve; this could help to reduce errant voiced/unvoiced decisions as well as frequency doublings/halvings. Finally, combining methods of pitch tracking (such as this and the cepstrum, or this and the AMDF) has been shown to be useful, but was not explored in this paper.

APPENDIX A: SYMBOLS

ACF	Autocorrelation Function	3
AMDF	Average Magnitude Difference Function	3
$\mathbf{a}_n$	Wavelet approximation n	6
$\mathbf{CMD}(i)$	Center Mode Distance for wavelet level i	12
δ	Small sample distance neighborhood	12
$\mathbf{d}_n$	Wavelet detail n	6
DFT	Discrete Fourier Transform	2
F	Maximum frequency parameter (Hz)	12
FFT	Fast Fourier Transform	2
FLWT	Fast lifting wavelet transform	2
f_n	Harmonic n	17
F_s	Sample rate (Hz)	12
L	Maximum number of wavelet transform levels	14
M	Maxima/Minima threshold percentage	11
N_d	Number of difference levels to calculate	12
NSF	National Science Foundation	iii
RMS	Root Mean Square	8
UIUC	University of Illinois at Urbana-Champaign	iii
WGN	White Gaussian noise	18
$\mathbf{x}(n)$	Discrete audio signal	8

APPENDIX B: HAAR WAVELET EQUATIONS

Here we look at the approximation and detail ($\mathbf{a_1}$ and $\mathbf{d_1}$) generated by a FLWT at the first level over a signal $\mathbf{x}(n)$, and show how they are equivalent to a low-pass and high-pass, followed by down-sampling.

Examining the equations derived earlier

$$\begin{aligned}
\mathbf{d_0}(n) &= \mathbf{x}(2n+1) \\
\mathbf{a_0}(n) &= \mathbf{x}(2n) \\
\mathbf{d_1}(n) &= \mathbf{d_0}(n) - \mathbf{a_0}(n) \\
\mathbf{a_1}(n) &= \mathbf{a_0}(n) + \frac{\mathbf{d_1}(n)}{2},
\end{aligned} \tag{B1}$$

note that these can be collapsed into

$$\begin{aligned}
\mathbf{d_1}(n) &= \mathbf{x}(2n+1) - \mathbf{x}(2n) \\
\mathbf{a_1}(n) &= \mathbf{x}(2n) + \frac{\mathbf{x}(2n+1) - \mathbf{x}(2n)}{2}.
\end{aligned} \tag{B2}$$

A final touch of algebra yields

$$\begin{aligned}
\mathbf{d_1}(n) &= \mathbf{x}(2n+1) - \mathbf{x}(2n) \\
\mathbf{a_1}(n) &= \frac{\mathbf{x}(2n+1) + \mathbf{x}(2n)}{2}.
\end{aligned} \tag{B3}$$

Here we see that $\mathbf{d_1}(n)$ is simply a high-pass (first difference) filter that has been down-sampled by taking every other resulting sample. Similarly, $\mathbf{a_1}(n)$ is simply a down-sampled low-pass (averaging) filter.

APPENDIX C: MATLAB CODE

```
function freq = wavePitch(data,fs,oldFreq)
%
%   WAVEPITCH Determine the pitch of a given short portion of data.
%
%   WAVEPITCH(data, fs, oldFreq) data is a [ 1 x samples ] array. Optional
%   inputs are fs (the sample rate of the signal, defaulting to 44100 if
%   unspecified), and oldFreq (the frequency from the previous window).
%
%   N.B. Data input should be at least 256 samples long. 1024 is recommended.
%        It also must be a multiple of 64.
%
%   Copyright 2005 Ross Maddox (University of Michigan)
%              and Eric Larson (Kalamazoo College)

if (nargin < 1) return; if (nargin < 2) fs = 44100; if (nargin < 3) oldFreq = 0; end

oldMode = 0;
if(oldFreq)
  oldMode = fs/oldFreq;
end

dataLen = length(data);
```

```
freq = 0;              % The freq to return
lev = 6;               % Six levels of analysis
globalMaxThresh = .75; % Thresholding of maximum values to consider
maxFreq = 3000;        % Yields minimum distance to consider valid
diffLevs = 3;          % Number of differences to go through (3 is diff @ third neighbor)

maxCount(1) = 0;
minCount(1) = 0;

a(1,:) = data;
aver = mean(a(1,:));
globalMax = max(a(1,:));
globalMin = min(a(1,:));
maxThresh = globalMaxThresh*(globalMax-aver) + aver; % Adjust for DC Offset
minThresh = globalMaxThresh*(globalMin-aver) + aver; % Adjust for DC Offset

%% Begin pitch detection %%

for (i = 2:lev)
  newWidth = dataLen/2^(i - 1);

  %% Perform the FLWT %%

  a(i,1:newWidth) = a(i-1,2*(1:newWidth)-1)/2 + a(i-1,2*(1:newWidth))/2;

  %% Find the maxes of the current approximation %%

  minDist = max(floor(fs/maxFreq/2^(i-1)),1);
  maxCount(i) = 0;
  minCount(i) = 0;

  climber = 0; % 1 if pos, -1 if neg
  if (a(i,2) - a(i,1) > 0)
    climber = 1;
  else
    climber = -1;
  end

  canExt = 1;   % Tracks whether an extreme can be found (based on zero crossings)
  tooClose = 0; % Tracks how many more samples must be moved before another extreme

  for (j = 2:newWidth-1)
    test = a(i,j) - a(i,j - 1);

    if (climber >= 0 && test < 0)
      if(a(i,j - 1) >= maxThresh && canExt && ~tooClose)
        maxCount(i) = maxCount(i) + 1;
        maxIndices(i,maxCount(i)) = j - 1;
        canExt = 0;
        tooClose = minDist;
      end
      climber = -1;

    elseif (climber <= 0 && test > 0)
      if(a(i,j - 1) <= minThresh && canExt && ~tooClose)
        minCount(i) = minCount(i) + 1;
        minIndices(i,minCount(i)) = j - 1;
        canExt = 0;
        tooClose = minDist;
      end
      climber = 1;
    end
```

```
  if (a(i,j) <= aver && a(i,j - 1) > aver) || (a(i,j) >= aver && a(i,j - 1) < aver)
    canExt = 1;
  end

  if(tooClose)
    tooClose = tooClose - 1;
  end
end

%% Calculate the mode distance between peaks at each level %%

if (maxCount(i) >= 2 && minCount(i) >=2)

  % Calculate the differences at diffLevs distances

  differs = [];
  for (j = 1:diffLevs)       % Interval of differences (neighbor, next-neighbor...)
    k = 1:maxCount(i) - j;   % Starting point of each run
    differs = [differs abs(maxIndices(i,k+j) - maxIndices(i,k))];
    k = 1:minCount(i) - j;   % Starting point of each run
    differs = [differs abs(minIndices(i,k+j) - minIndices(i,k))];
  end

  dCount = length(differs);

  % Find the center mode of the differences

  numer = 1;   % Require at least two agreeing differs to yield a mode
  mode(i) = 0; % If none is found, leave as zero

  for (j = 1:dCount)

    % Find the # of times that distance j is within minDist samples of another distance
    numerJ = length(find( abs(differs(1:dCount) - differs(j)) <= minDist));

    % If there are more, set the new standard
    if (numerJ >= numer && numerJ > floor(newWidth/differs(j))/4)
      if (numerJ == numer)
        if oldMode && abs(differs(j) - oldMode/(2^(i-1)) ) < minDist
          mode(i) = differs(j);
        elseif ~oldMode && (differs(j) > 1.95*mode(i) && differs(j) < 2.05*mode(i))
          mode(i) = differs(j);
        end
      else
        numer = numerJ;
        mode(i) = differs(j);
      end
    elseif numerJ == numer-1 && oldMode && abs(differs(j)-oldMode/(2^(i-1))) < minDist
      mode(i) = differs(j);
    end
  end

  %% Set the mode via averaging %%

  if (mode(i))
    mode(i) = mean(differs(find( abs(mode(i) - differs(1:dCount)) <= minDist) ));
  end

  %% Determine if the modes are shared %%

  if(mode(i-1) && maxCount(i - 1) >= 2 && minCount(i - 1) >= 2)

    % If the modes are within a sample of one another, return the calculated frequency
```

```
      if (abs(mode(i-1) - 2*mode(i)) <= minDist )
        freq = fs/mode(i-1)/2^(i-2);
        return;
      end
    end
  end
end
```

---

[1] E. Ercelebi, Journal of Applied Acoustics **64**, 25 (2003).

[2] I. Daubechies and W. Sweldens, Tech. Rep., Bell Laboratories, Lucent Technologies (1996).

[3] E. Larson and R. Maddox, in *Proceedings of the University of Illinois at Urbana Champaign Research Experience for Undergraduates Program* (2005).

[4] P. de la Cuadra, A. Master, and C. Sapp., in *Proceedings of the 2001 International Computer Music Conference* (2001).

[5] R. Polikar, *The wavelet tutorial*, http://users.rowan.edu/p̃olikar/WAVELETS/ (2005).

[6] M. Mellody, M. A. Bartsch, and G. H. Wakefield, Journal of Intelligent Information Systems **21(1)**, 35 (2003).

[7] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith, in *Proceedings of the third ACM international conference on Multimedia* (1995), pp. 231–236.

[8] J. P. Bello, G. Monti, and M. Sandler, in *Proceedings of the 1st Annual International Symposium on Music Information Retrieval* (2000).

[9] G. P. Scavone, *Rtaudio*, http://www-ccrma.stanford.edu/ gary/rtaudio (2002).

[10] L. Yoo and I. Fujinaga, in *Proceedings of the International Computer Music Conference* (1999).

[11] T. D. Rossing, *The Science of Sound* (Addison-Wesley, 1990), 2nd ed.