The Challenges of Transcription of Music

EE221 Senior Design Project

Ayan Shafqat

To my beloved fiancée, Farhana Ferdous:
Thanks for believing in me, and thanks for being there for me.

Abstract

This paper will discuss the different signal processing challenges that is faced when we are trying to transcribe a music from an audio source. There have been many algorithms that have been proposed to solve this problem, but this paper discusses and implements one of them. This algorithm is known as YIN, which estimates fundamental frequency in an audio signal. From estimating the fundamental Frequency, we can assume the signals pitch in a given time frame. Also, from estimating the instantaneous power of that signal, we can estimate dynamics in that given time-frame.

Acknowledgement:

Thanks to Professor Matthew Campisi to help me with implementing and guiding me through the algorithm. This paper wouldn't be possible without his guidance.

Thanks to Udit Mital (GA) to help me verify the equations and expressions in this paper.

Table of Contents

Chapter 1: Characterization of the Problem

Transcribing music is an elaborative process of converting a particular piece of music, either from an audio source or a live instrument, into symbolic notations that contains all possible musical information. *Automatic Transcription of music* is doing this exact task in terms of algorithms and signal processing, which has been somewhat of a challenge for decades. There have been a lot of publications in this topic, which comes from several disciplines of science, including *Musicology, Psychoacoustics, Computer Science, Mathematics, Digital Signal Processing, Computational Auditory Scene Analysis*, and many others. This research topic has also inspired new disciplines such as, *Computer Music, Music Content Analysis*, etc. Before we can understand how to transcribe music from an audio source, we must characterize the problem in general, and understand what we information we need to extract from an audio source. In order for us to program a computer to solve this problem, we must look at how an experienced musician transcribes music.

As an experienced musician, I myself have transcribed music in various genres. But, it is still one of the most challenging tasks that I had to deal with and took a lot of ear training. In order to transcribe a piece of music, I found myself listening to the same song over many times and then break it down into parts, such as the verse, chorus, bridge, coda, etc. Then, I would have to separate different instrument's parts by listening to each instrument parts closely, and tune the rest of the music out of my head. After listening to a particular instrument, I would compare the tone with my own instrument to see if I have the right notes, which may involve my guitar or just by humming, and play along with the original track. Finally, I would figure out what the tempo of that song is with a metronome and write down the notes accordingly on a staff. Although, through experience, I have learned how to separate an instrument in my head, I have a very hard time explaining how I do so.

In any piece of music, there are several key features that experienced musicians look for. First and one of the most important features of any piece of music is its *pitch*, which contains the spectral information of a piece. Finding pitch can be simplified to a musician, if he or she figures out the key of the piece, which narrows the search from 12 notes down to 7 notes. Another piece of information that a musician looks for is its *dynamics, or perceived loudness.* This feature defines how loud or soft the piece is, and musicians address them as *piano* (soft), *mezzo-piano* (semi-soft), *mezzo-forte* (semi-loud), and *forte* (loud) (Porter & Mason, 1832). In addition, a piece of music is composed of a tonic feature, called *timbre* (pronounced: tam-bər), which determines which pitch belonging to which instrument. Last but the most important feature of any piece of music is said by Ms. Ferdous, who is an experienced singer. She said, "Music can be related to a piece of painting, where each colors can relate to different notes, different pressures and brush strokes can relate to the loudness and tonality. But, just as there cannot be a painting without a canvass, there cannot be music without *time*." In terms of signal processing, we call this feature *musical event information.* In the next sub-sections, we shall look at each of these features in detail.

Chapter 2: Background

Section 2.1: Pitch

According to Klapuri, "*Pitch* is a perceptual attribute which allows the ordering of sounds on a frequency-related scale extending from low to high. More exactly, *pitch* is defined as the frequency of a sine wave that is matched to the target sound by human listeners." In other words, pitch can be arbitrary, which does not have to exist physically in the frequency domain, rather it's what we hear in our mind. Questions may arise as, "Is perception of pitch something we are born with?" A Professor of music theory in Polytechnic Institute of NYU, Alph Edwards said, "Human perceptions of pitch are innate... like every child has their favorite songs and can sing them recognizably." There are cases observed, where a person may be able to hear sound, but does not perceive pitch, which is known as *tone deafness*. These patients do not have the ability to process music, since they cannot tell the difference between two tones with two different frequencies. But, music played a big role in our history, and has several ways of addressing them varying from culture to culture. This paper will only consider western music model of pitch.

In western music, pitch is associated with the first seven letters of the alphabet, starting from A to G. These seven letters played in a sequence is known as a scale. Western music are composed of these seven note scales which are called diatonic scales, in which each scales have two modes, major and minor. Pythagoras on Ancient Greece had discovered this phenomenon, where he laid the foundation of music intervals by dividing an octave into near integer intervals. Ancient Greeks were the first to realize the concept of an octave, by dividing a string in half and keeping the same tension would result into twice the fundamental harmonic. Then, dividing that half sized string would result into 4th harmonic and so on. It was this nature of subdividing the string into two later on resulted into the twelve note scale that is in use in modern music, which is geometric in nature (Note that A is the only note with rational frequency, the rest are geometrically spaced from A_N to A_{N+1}). This is known as the Twelve-tone Equal Temperament tuning system.

In general, pitch can be considered as a function of fundamental frequency, although it may not be applicable all the time (For more info, refer to Moore's book). In the previous paragraph, it is indicated that the early civilizations denoted musical intervals in terms of diving a string into halves, and so forth. In terms of frequency, this can be expressed as a geometric sequence ($f(n) = \alpha 2^n$), which can be described mathematically. The frequency of the next semitone or a note can be expressed as a function of the frequency corresponding to the current note, and the number of semitones above or below. Suppose we let f_0 be the current frequency corresponding to a certain note on the piano. The next semitone would be one key above (to the right), of the current note, which would have a frequency of $f_0 \sqrt[12]{2}$. Given a fundamental frequency (f_0) of a certain note, we can compute the frequency of n^{th} semitone as $f_0\left(\sqrt[12]{2}\right)^n$, where the number of semitones above will consist of a positive value for n and negative for semitones below. In standard twelve-tone tuning, we consider A_4 (A above the middle C) to be the reference note, which has a frequency of 440 Hz. This is also the 69th note in MIDI code, which is the most popular protocol for interfacing computers with digital instruments. We can create a general formula for pitch in terms of MIDI code, which is shown below:

$$f_0 = 440\left(2^{\frac{M-69}{12}}\right)\text{ Hz} \quad \text{or} \quad M = 69 + 12\log_2\left(\frac{f_0}{440\text{Hz}}\right)$$

Where f_0 is the frequency corresponding to the MIDI code, M is the MIDI code corresponding to the fundamental frequency f_0.

One important feature to note about MIDI protocol is the fact that M has a value ranging from 0 to 127. This corresponds to C_{-1}, which has a frequency of 8.1758 Hz (below hearing range, thus having an octave number of -1), to G9, which has a frequency of 12.5439 KHz. So a good rule of extracting the note's name from a MIDI code, M, is shown below:

Let N be a set of chromatic scale, where: $N = \{\text{C}, \text{C\#}, \text{D}, \text{D\#}, \text{E}, \text{F}, \text{F\#}, \text{G}, \text{G\#}, \text{A}, \text{A\#}, \text{B}\}$

$$P = \left(N\{\langle M\rangle_{12}\}\right)_{\left\lfloor\frac{M}{12}\right\rfloor-1}$$

Where $\langle A\rangle_B$ is a modulus operator (Remainder of A, as A is divided by B)

Section 2.2: Perceived Loudness

Besides hearing pitch, we also can distinguish loud parts of a musical piece form a soft part. In music theory, this is known as *dynamics*. In music, *dynamics* can refer to the amplitude of a certain note, or it can refer to the loudness of an entire section of a piece. Musicians have given them four discrete values, *piano* (soft), *mezzo-piano* (semi-soft), *mezzo-forte* (semi-loud), and *forte* (loud) (Porter & Mason, 1832). But, these are not really quantifiable values, since they are relative and not absolute. In terms of engineering, loudness can refer to the amplitude of a signal. But, in general loudness can correspond to the overall average power of a signal in a particular time window.

Basically sound in nature is a pressure wave that occurs when there is a disturbance among air molecules. Pressure, in theory, is the amount of force that is being applied per unit of area, which has a unit of N/m^2 or Pascal. Therefore, our ears can be thought of as a pressure sensor with a very high dynamic range. The unit can be so great, that we often use a logarithmic unit for sound, known as decibels (dB). It is a ratio between the minimum force it takes to vibrate air molecules and the pressure given. The formulation for this unit is shown below:

$$S_{dB} = 20\log_{10}\left(\frac{P}{P_0}\right) = 10\log_{10}\left(\frac{I}{I_0}\right)$$

S_{dB} is the sound pressure level, which depends on pressure (P) or intensity (I). P_0, or the *absolute threshold of pressure* that causes air molecules to vibrate is equivalent to 20µPa, and similarly I_0, or the absolute threshold intensity that is equivalent to 10^{-12} W/m^2 (Moore, 1997).

In principle, a transducer or a microphone can be thought of as a pressure sensor, which measures the disturbances of air molecules. The amount of pressure can be directly proportional to the size of the diaphragm, the sensitivity of the material, etc. Similarly, our ear measures sound pressure as well, with the upper limit of 120 dB above the faintest sound we can hear before damaging our eardrums. This corresponds to a dynamic range of 120 dB, or in scalar units 1 Trillion to 1 (Moore, 1997).

There have been many ongoing researches regarding how our ears react to subjected loudness in different frequencies, due to various practical uses. One of the most important applications that derived from the study of the perception of loudness in different frequencies is data compression for audio signals (Jayant, Jhonston, & Safranek, 1993). During the late 1980's and early 90's Moving Picture Expert Group (MPEG) have done a lot of tests with human subjects, and created an algorithm for a lossy compression that takes advantage of a perceptual limitation of human hearing, a study known as *auditory masking.* The oldest study of auditory masking was conducted by Alfred Marshall Mayer, who stated in a report in 1894 that a tone could be rendered inaudible by another tone of lower frequency (Moore, 1997). Research during the 1950's, conducted by Richard H. Ehmer, indicated the frequency curves of our ear by mapping out *Minimum Audible Field* (MAF) of human ears, which shows the minimum signal intensity required in certain frequency band that would make us register that particular sound (Moore, 1997). Also, one of the aspect of our hearing that led to MP3 compression is known as *Equal Loudness Contour*, which shows our ear's response subjected to equal loudness at different frequency tones (Jayant, Jhonston, & Safranek, 1993).

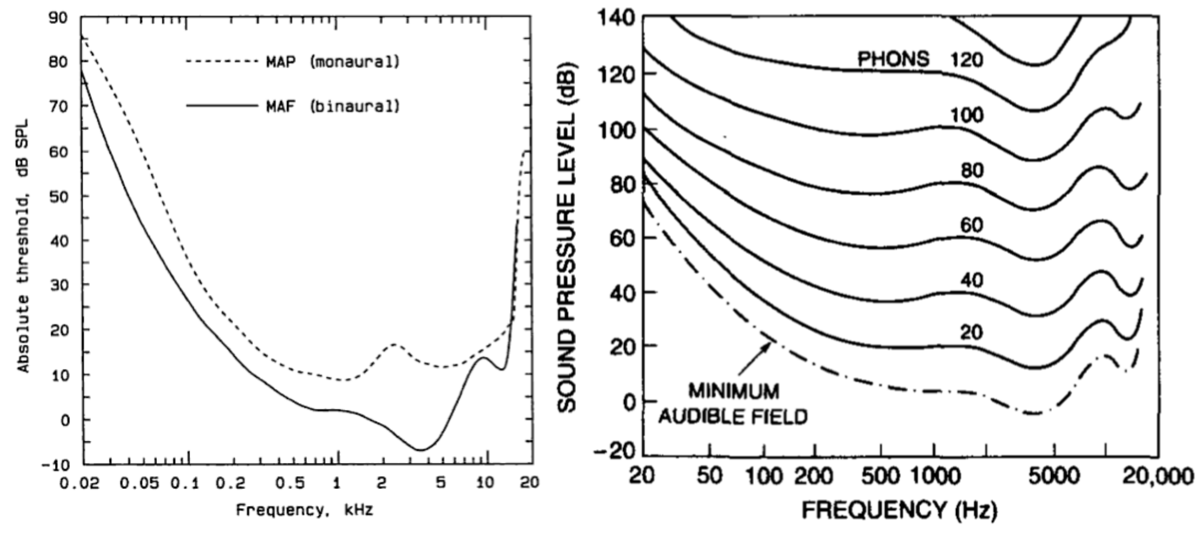


Figure 2.2a: The graph on the left shows absolute threshold of hearing (Moore, 1997), and the figure on the right shows our ear's response to different pressure levels and their dynamic ranges, which led to the MP3 compression (Jayant, Jhonston, & Safranek, 1993).

These graphs show that we hear signals with the same intensity with different magnitude, in different frequency bands. This can be thought of as the frequency response of our ear in different loudness, just as a microphone has a frequency response in different loudness levels. There are many other attributes that are associated with our perception of signal intensity, but this is beyond the scope of this paper (For more info, refer to Moore).

When digital media storage is concerned, sound intensity is normalized into the range of -1 to +1, which can be represented as 8, 16, or 24 bits of precision. Therefore, the maximum normalized power

($|x(t)|^2$) a signal can achieve is 0 dB ($10\log_{10}(|x(t)|^2)$ *or* $20\log_{10}|x(t)|$), and can go down to as much as the precision allows. For example, a 16-bit or CD quality audio would allow a minimum of $1/2^{16}$ precision, which would correspond to 96 dB of dynamic range, $20\log_{10}(2^{16})$. Therefore, anything below -96 dB is basically irrelevant. Also, if we note MAF graph above, we can infer that, for low frequencies (20Hz), anything below -16 dB (80dB – 96dB) would be irrelevant, since a human ear would not be able to hear them.

Section 2.3: Timbre

Any musical instrument is periodic in nature, which means we can expand the Fourier series coefficients from that signal. So, we can somewhat say *timbre* relates to the frequency spectrum of a particular instrument's signal, which is only the half of what a timbre actually is. According Klapuri, timbre is "sometimes referred to as sound 'colour' and is closely related to the recognition of sound sources... The concept is not explained by any simple acoustic property but depends mainly on the coarse spectral energy distribution of a sound, and the time evolution of this [signal]." A piano and a guitar may have the same Fourier series expansion, but we can clearly distinguish the sounds from these instruments. Therefore, in order to analyze timbre, we not only need to consider the spectrum of an instrument's signal, but also in time. A Short Time Fourier Transform (STFT) can be take of a signal and graphed over time to visualize timbre. The graph below shows the timbre of a Fender Stratocaster electric guitar and a Steinway Grand Piano, both playing A$_5$ (f_0 = 880 Hz). Observe both time and frequency domain envelope of the instrument.

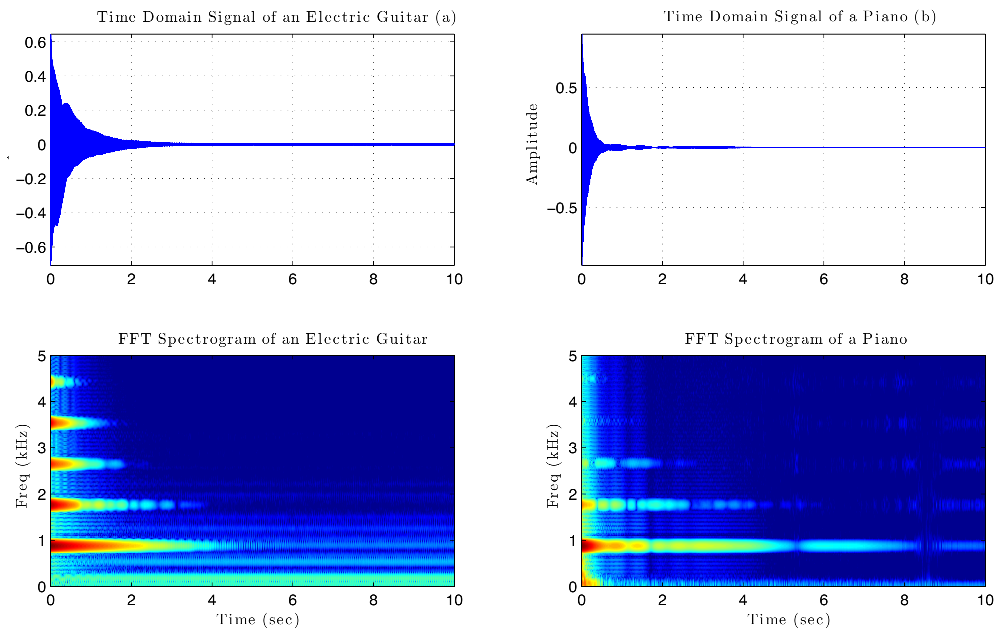


Figure 2.3a: The graph on the left (a) shows the time and frequency-time domain plots of a Fender Stratocaster electric guitar (Self Recorded), playing the note A$_5$ (880Hz). **Figure 2.3b:** The figure on the right (b) shows the time and spectrogram of a Steinway Grand Piano (Apple Logic Sample), playing the same note (880Hz).

From the two graphs in the previous page (*Figure 2.3a*), we can see that the time domain responses are very similar in both instruments. Both of them show an exponential decay as the strings dampens over time. Also, notice the frequency characteristics are different in both instruments in time, for which the piano's harmonic content damps out faster than electric guitar. Therefore, it can be inferred that both time and frequency domain characteristics contain timbre information.

Section 2.4: Music Events

In order to understand the importance of music events, we must look at an analogy that we can relate to in a daily basis, such as spoken and written language. Spoken language is composed of infinite set of syllables, which can be broken down into two or three alphabets. In each syllable, there is at least one vowel present, and two or more syllables can construct a word. Alphabets and syllables can correspond to different pitch of music, since they are both sets to describe a language. Then a few of these words can be created into a sentence, which expresses a thought. Note, in a sentence, the orders of words are relevant. For example, "How are you?" vs. "You are how?" would mean two different things (or does not express any meaning at all). This clearly shows the importance of time in spoken language, which is known as an *event*. We can go down even further, where each syllable can be thought of as an event, or each letter. Although, we may have used the same letter multiples of times in a sentence, the orders of these letters are relevant.

Similarly in music, it composes of pitch, timbre, and dynamics, but they come in a joint-time frequency model, which we will call *musical events*. According to Klapuri, "Musical information [event] is generally encoded into the *relationships* between individual sound events and between larger entities composed of these. Pitch relationships are utilized to make up melodies and chords. Timbre and loud- ness relationships are used to create musical form especially in percussive music, where pitched musical instruments are not necessarily employed at all (Klapuri & Davy, 2006)." Just like spoken languages, music consists of infinite sets, which can be quantized into pitch. The types of instruments that are played in a particular time would have certain timbre, but there is more information that can exist in an event. For example, the transition of note to another note in a guitar would create a "scratch" sound, or a singer breathing in before he or she starts singing, etc. The amount of information that can be extracted from an audio source can be overwhelming. Thus, we have some models that somewhat quantize into simplified information, which we will discuss in the next section.

It is very important for us to understand how would we want to visualize music after we transcribe. Without some sort of visualization or some means of communication protocol, transcribing would not be possible and not to say, impractical. Since the earliest method of transcribing music was by humans, we have developed various ways of writing them throughout various cultures. Among handful methods of written music, one of them stands out the most, which is known as staff-notation.

Section 2.4.1: Staff Notation

In western music, it is common to see a staff notation, which is composed of five lines. Each of these five lines corresponds to a certain pitch of a key. The way an event is written in this form is as follows: Every music is given a time signature, which defines the frame of time required for one quarter note. This is known as *tempo* of a piece. Typically, a piece of popular music has a time signature of 4/4 ("four by four"), which corresponds to four-quarter notes in a bar. Just as there are four quarters in a dollar. The equivalent time of two quarter notes is known as a half note, and half of a quarter note is known as eighth and half of that would be the sixteenth note. An example of a staff notation is shown below from Chopin's Op. 10 No. 12.

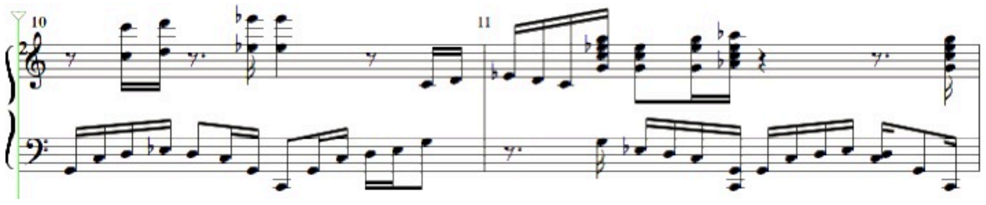


Figure 2.4.1a: Staff notation of Chopin's Op. 8 No. 12. (Midiworld, 1995)

One major disadvantage of this notation is the fact that it is "human-oriented," especially due to the fact that there is no presence of an absolute-time. In order for computers to process music, it is much easier to time in terms of a standard unit of time, such as seconds. Moreover, another key information is missing from this notation, which is the lack of dynamics of each note. It is mainly up to an instrument player to decide the loudness of a note, without any quantifiable value. Therefore, there is a need for a protocol which takes account for more details about a piece of music, and much easier for a computer to understand. Fortunately in the 1980's, engineers have created a protocol known as MIDI.

Section 2.4.2: MIDI

MIDI stands for Musical Instrument Digital Interface, which has been a standard protocol for keyboards, electronic drum sets, DJ lighting, along with many other equipment in the audio industry. What makes MIDI different from a staff notation is the fact that there is more event information recorded in a MIDI file. One of the key information that is recorded is the loudness of a note, which is known as *velocity.* The figure below shows the same piece that was shown in previous page in a MIDI piano roll view in Logic Studio. Notice the color difference in the notes (softer notes are softer in color and louder notes are warmer in color) and the reference to the absolute time.

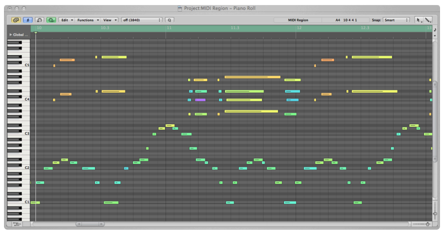


Figure 2.4.2a: Apple's Logic Studio render of Chopin's Op. 8 No. 12. (Midiworld, 1995)

Section 2.5: Simplifications

Through research, it has been found that transcribing a polyphonic music is nearly impossible with the given duration to complete the project (3 months). Therefore, this paper will focus on how to extract pitch from a monophonic signal, such as the sound of a vocal melody, or any wind instrument. Then, the problem becomes much simpler, in which we can work on a two dimensions instead of three. Given the fact that the signal is monophonic, we can assume that there will be one frequency in one particular time, and the dynamics of that note would be the power of the overall signal. If there is no note sung in a given time, we can denote it as an error code for "not found." So, we can simplify our transcriptions down to two functions.

$$f_{est}(t) \equiv \text{The pitch of a signal in a given time, } t$$

$$p(t) \equiv \text{Average power of the signal in a given time, } t$$

After finding these two functions over time, fundamental frequency can be converted into MIDI notation (as shown in Section 2.1). It can be inferred that the event of music is in time, which is in seconds, and the overall dynamics is in decibels (as shown in Section 2.2). Therefore, the overall music piece can be written as a sum of Dirac Delta Functions, such that:

$$M_\mu = \sum_n p_{dB}(t_n)\delta(t - t_n)\delta(M - M_n)$$

Where M_μ is the entire music composition, t is the time axis, and M is the MIDI axis.

This gives us a good visual aid to a monophonic music piece, which is shown in figure below (*Figure 2.5a*). As shown below, the louder notes are represented with a brighter color and duller notes are represented with darker color. This paper will discuss in details how to get from a monophonic audio source to a notation that can be visualized.

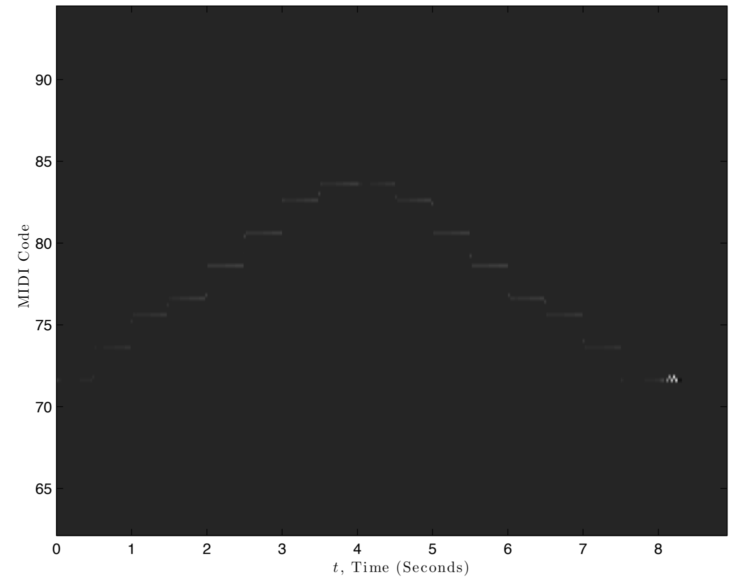


Figure 2.5a: Shows the output of the implemented algorithm showing a C Major Scale in the 5th octave (MIDI note of 72 to 84).

Chapter 3: Time Domain Methods of PDA

Section 3.1: Zero Crossing Method

Zero crossing method is a very simple statistical method that can be used for detecting fundamental frequency of a signal. It is the method that is used to detect the frequency of an outlet voltage, since the signal is stationary. It is very fast, since it only requires half a cycle of information to determine fundamental frequency. But, using this method is very impractical for detecting pitch in a music signal, since music signal is non-stationary. Moreover, a music signal consists of more than one frequency components which may cross zero axis more often than its fundamental frequency, which will cause instability in the frequency reading. Zero-crossing test is denoted as the formula shown below, and the figure below also shows with just a little noise, this method can be very unreliable (Figure 3.1a).

$$x[n]\, x[n+1] \leq 0$$

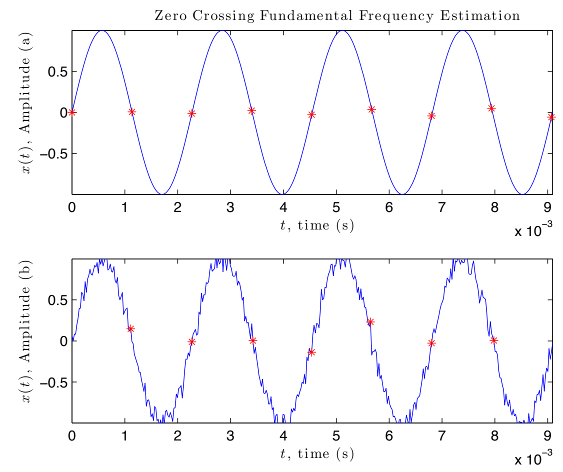


Figure 3.1a: Showing zero crossing method of estimating fundamental frequency in a clean sine tone (a). This fails in (b) which shows a similar signal with 20dB of SNR.

As shown above, zero crossing method is extremely fast in terms of detecting the fundamental frequency of a stationary signal. In practical applications, zero crossing method has been in use for digital communication over analog lines, such as modems. It is also observed that zero-crossing method is used in image processing for application in edge detection, where the product change of two consecutive pixels are less than zero.

This method can be used for pitch detection as well, if sufficient preprocessing involved. It has been observed that by using a comb filter, and then later on clipping the signal will result into the fundamental period, which can then be used to drive a tachometer. Since the output voltage of a tachometer will be a linear function of frequency, the fundamental frequency can be extracted without a system possessing memory. But, that also raises the complexity of the system, while still remaining inaccurate. Therefore, a more robust method will be used to determine the fundamental frequency.

Section 3.2: Autocorrelation Function (ACF)

Autocorrelation is a type of *cross-correlation* function that finds similarities between signals with different time lags. This is also known as the *sliding dot-product*, or *linear inner-product.* Cross correlation is used for various applications including, pattern-recognition, single particle analysis, and many others. In our context, we will use autocorrelation to find the fundamental frequency of a signal. Before we do that, we will look at what cross-correlation is defined as (1).

$$(x \star y)(t) \equiv \int_{-\infty}^{\infty} x^*(\tau)\, y(t+\tau)d\tau \tag{1}$$

The cross-correlation function can be then converted into autocorrelation by substitution. Since we will be using real signals only, we do not have to worry about the conjugate operator. Thus the equation looks as (2).

$$r'(t) = \int_{-\infty}^{\infty} x(\tau)\, x(t+\tau)d\tau \tag{2}$$

We know from the fact that, a given periodic function with a certain period τ_0 will consist of the property of $x(t) = x(t+\tau_0)$. Therefore, we can assume that integration of multiplication of a function with itself will possess maximum value when delay is equivalent to its fundamental period. Since equation (2) can cause unstable results, we can stabilize them by multiplying the window length's inverse shown in equation (3).

$$r(\tau) = \lim_{T_0\to\infty} \frac{1}{2T_0} \int_{-T_0}^{T_0} x(t)x(t+\tau)d\tau \tag{3}$$

In a practical, stable system, we can constrain the limits of integration and since we will be looking at causal systems only, we can look at a frame of time ($0 \le t \le T_0$) that simplifies our expression to (4).

$$r(\tau) = \frac{1}{T_0}\int_{0}^{T_0} x(t)x(t+\tau)d\tau \tag{4}$$

Since we will be processing discrete signals, we can just take N number of samples and sum their product in a small time frame. Throughout the rest of the paper, we will only consider discrete functions only. For discrete, causal, and practical systems, autocorrelation function will look like (5).

$$r[m] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]x[n+m] \tag{5}$$

In the expression above, m is the discrete delay of the function $x[n]$. Therefore, the maximum value of $r[m]$ will correspond to the amount of sample it takes to repeat a cycle. Therefore, we can conclude that the distance between two peaks of an autocorrelation function will correspond to its fundamental period in samples. Assuming the signal has a sample time T_0, we can easily calculate the fundamental frequency of the signal ($f_0 = (T_0 m_0)^{-1} = f_s/m_0$), which is also the pitch of the signal. The figure below shows the fundamental frequency estimation of a sinusoid and a practical signal, both 440Hz, by finding the distance between two peaks of an autocorrelation function.

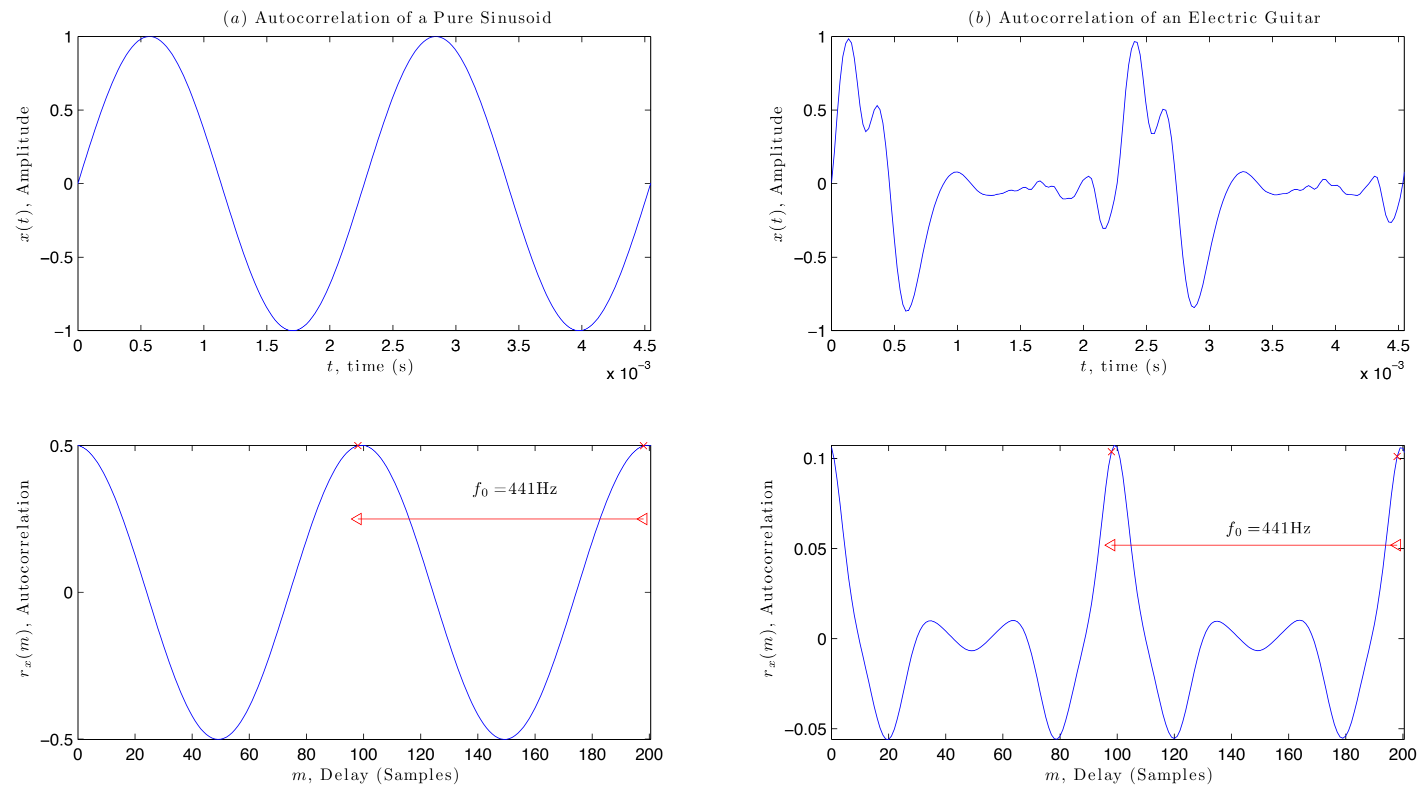


Figure 3.2a: Showing two cycles of a 440Hz sine tone ((a) top). Autocorrelation functions of the 440Hz sine tone ((a) bottom), the peaks are marked with '×' and the estimated fundamental frequency is shown in that graph as well (441Hz). (b) Showing 440 Hz electric guitar tone and its corresponding autocorrelation function below ((b) bottom).

As shown (*Figure 3.2a*), autocorrelation function can be used to determine the fundamental frequency regardless of the signal's harmonic contents. As long as the function is periodic, autocorrelation can be used to extract the fundamental period. Notice, autocorrelation function discussed here is for discrete time band limited signals, which also outputs periodicity in numbers of samples. Therefore, m is denoted as the discrete period domain, instead of τ, which in this paper denotes continuous period. Also, note the fact that the estimated fundamental frequency that will be detected from this method will be an integer multiple of the sampling rate (Note the figure above estimates a fundamental frequency of 441Hz, which is an integer multiple of the sample rate of 44100Hz by a factor of 0.01). As shown later on this paper, the error rate will increase as the function is subjected to higher frequency signals.

Section 3.3: Average Difference Magnitude Function (AMDF)

In the previous section, it has been discussed how to measure the fundamental frequency of a signal using autocorrelation function. But, one of the major drawbacks of this method is the amount is the range, and maxima detection, which may take more time and processing cost to determine pitch. Similar to an Autocorrelation function, *Average Magnitude Difference Function* (AMDF) is based on the same principle, in which we can assume the same values for periodic signals after one period of delay. Since $x(t) = x(t + \tau_0)$ for all *t*, we can make the assumption $x(t) - x(t + \tau_0) = 0$, for all t, if τ_0 is the fundamental period. Therefore, $\int_0^{\tau_0} |x(t) - x(t+\tau)|^2 d\tau = 0$, if the same condition applies. We formally write AMDF as follows (Cheveigné & Kawahara, 2002).

$$d[m] = \frac{1}{N} \sum_{n=0}^{N-1} |x[n] - x[n+m]|$$

(**Note:** This paper removed the squared operation from AMDF discussed in Cheveigné & Kawahara's paper, since the function looks similar without the square operation and in order to save processing cost.)

Testing a sinusoid at 440Hz, the function's output is shown in the graph below (*Figure 3.3a:Left*). One of the advantages that AMDF has over autocorrelation is the fact that the numbers are never negative, which can be an aid to simplifying algorithm. But, both functions are noise prone, which is shown in Figure 3.3a:Right, tested with a 440Hz square wave with signal to noise ratio of 5dB.

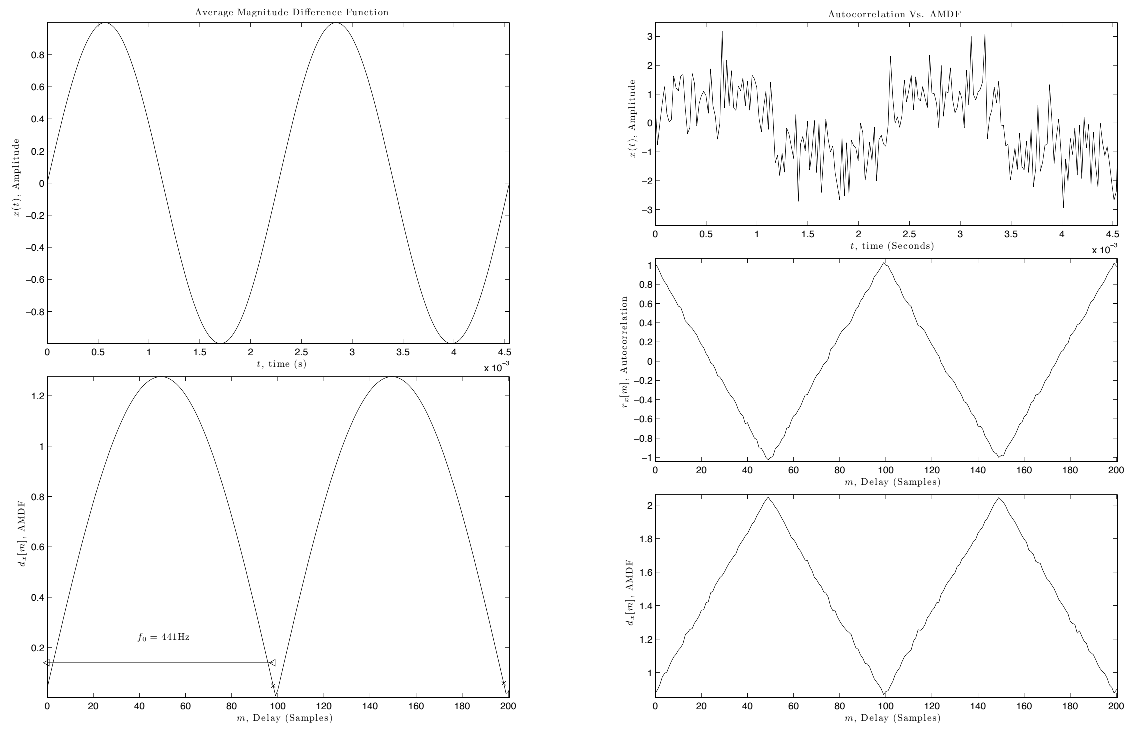


Figure 3.3a: Left figure shows *Average Magnitude Difference Function* (AMDF) of a pure sine tone of 440Hz. The right figure shows the similarities of both functions in a noisy signal of a square wave of 440Hz with a SNR of 5dB.

From the analysis, we can see that both functions can be useful in order to determine the fundamental frequency of the signal. But, these functions alone may not be enough for a computer to calculate the fundamental frequency of a signal, and it may not be so accurate. As shown for both functions, the estimated frequency detected was 441Hz, which has an error of 0.23%. The error also increases for both functions as we raise fundamental frequency, which is shown in the graph below (*Figure 3.3b*). Although autocorrelation does show less error than AMDF, it is harder to implement a fundamental frequency estimation using it. Furthermore, it is computationally cheaper to calculate AMDF than autocorrelation, since an addition is much cheaper than a multiplication in terms of computing efficiency.

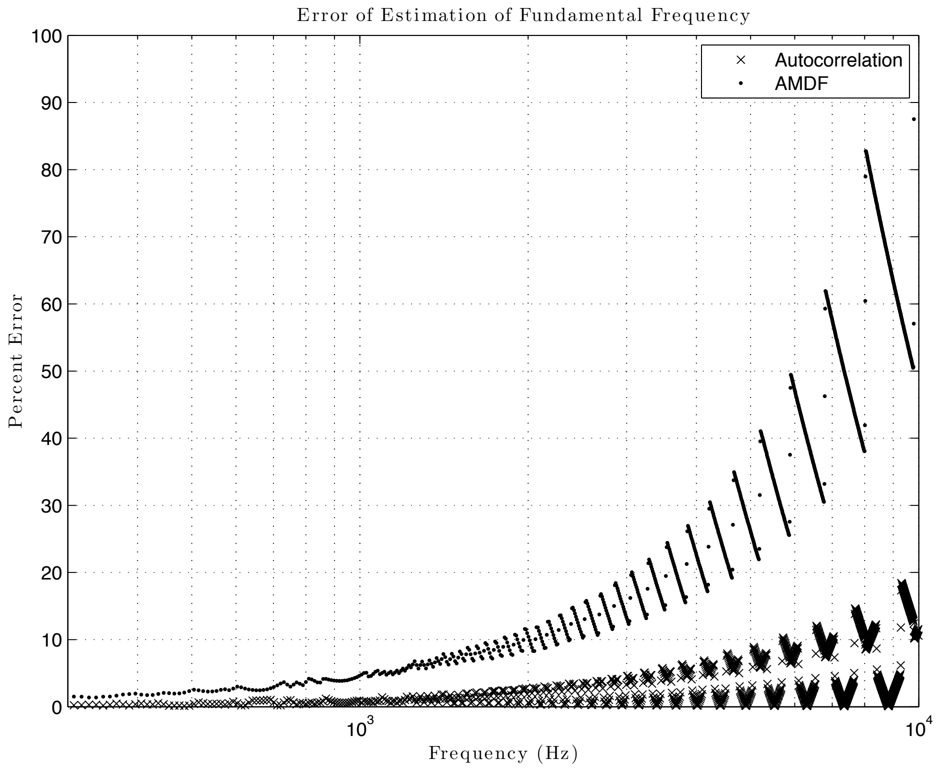


Figure 3.3b: The figure above shows the percent error of fundamental frequency estimation using Autocorrelation (ACF) and Average Magnitude Difference Function (AMDF) over the frequency ranges from 100Hz to 10KHz. This is without doing any interpolation or up sampling original data.

As shown on the figure above, the error of estimated fundamental frequency increases as the fundamental frequency increases. This is due to the sampling period and derivative calculations, and zero crossing detection of derivatives. In order to increase accuracy, we can either up-sample the raw data or interpolate autocorrelation functions. Interpolation of ACF and AMDF is computationally cheaper than up-sampling original data. Before we can get there, we must also make the algorithm more robust and faster. Moreover, if the algorithm is not set up for a certain search range for finding the fundamental period, the algorithm may choose the zero-lag period (Cheveigné & Kawahara, 2002). Therefore, we will discuss a function that removes the initial dip, which makes it more robust, called the *Cumulative Mean Normalized Difference Function* (Cheveigné & Kawahara, 2002).

Section 3.4: Cumulative Mean Normalized Difference Function (CMNDF)

Due to imperfect periodicity caused by several features of a signal, AMDF is never quiet zero at zero lag time, which may cause a failure of the algorithm (Cheveigné & Kawahara, 2002). Also, another feature of a practical signal has resonance, which may produce a much lower dip in the 2nd cycle instead of the fundamental period, which can cause an octave error. In order to prevent this, there is a function known as *Cumulative Mean Normalized Difference Function* (CMNDF), which normalizes AMDF to some absolute range that can be analyzed by the algorithm. CMNDF is defined as:

$$c[m] = \begin{cases} 1, & m \leq 0 \\ \dfrac{m\, d[m]}{\sum_{n=1}^{m} d[n]}, & m > 0 \end{cases}$$

Notice the fact that, $c[m]$ starts at 1 instead of zero. This can be useful while detecting the fundamental period, such that the algorithm just needs to look for the first dip and can make a decision regarding fundamental period (f_s/m_0), whereas for ACF or AMDF, it must measure the time between two extremes. Another advantage of CMNDF can be observed when it is subjected to a noisy signal, such as a decaying sine tone (shown below).

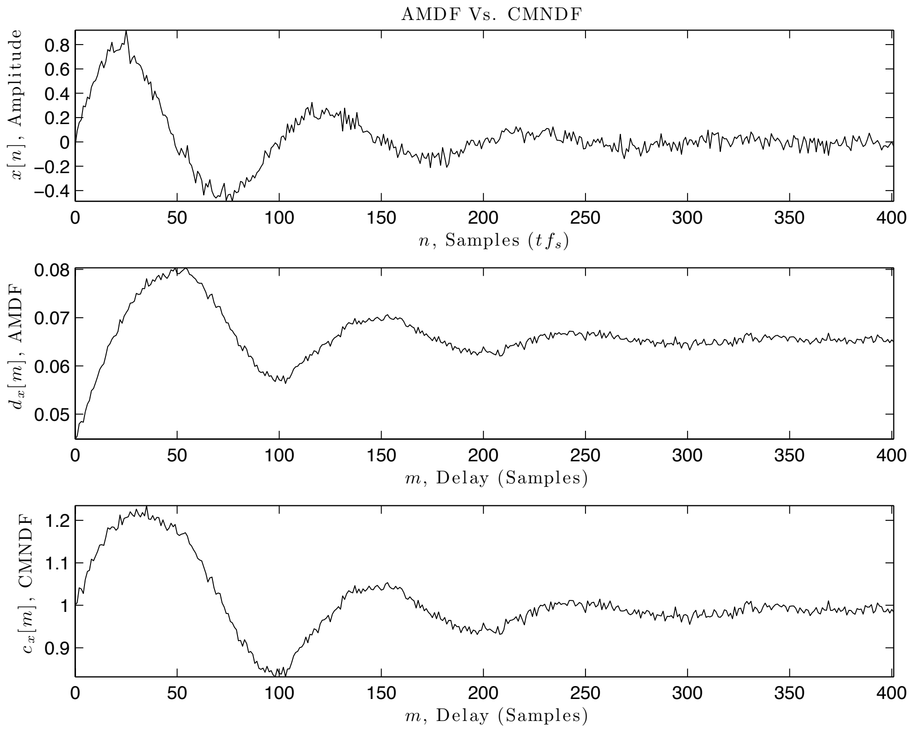


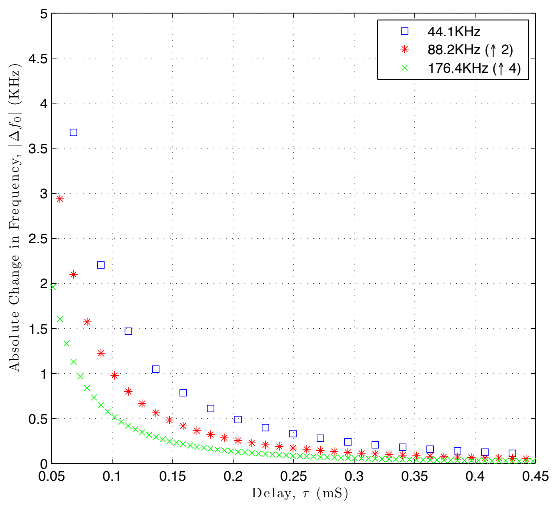
Figure 3.4a: Showing the difference between AMDF and CMNDF of a decaying sine wave with SNR or 5dB.

In the figure above, it is shown that CMNDF has a sharper dip at the fundamental period when it is subjected to a decaying sinusoidal signal, which is ideal for any string instrument. Also, this makes the fundamental period more defined and easier to calculate. Moreover, through observation it has been concluded that CMNDF can still function down to 5dB of signal-to-noise ratio.

Section 3.5: Limitations of AMDF, ACF, and CMNDF

Considering a band limited signal with a certain sampling rate, it can be assumed that the highest fundamental frequency extracted from time domain methods is a half of the sampling rate. At Nyquist frequency, periodicity of the correlation functions will be two samples apart. But, as discussed earlier in the paper, a periodic signal subjected to this algorithm will have correlation factor in the integer multiple of the sampling rate. This poses a problem in the higher frequencies, since the changes in frequency is greater as correlation factor goes towards zero. If fundamental frequency is considered as a function of the correlation factor (m), it the function is hyperbolic ($f_0(m) = f_s/m$). Thus, the absolute change in fundamental frequency as a function of correlation factor is also hyperbolic ($|\Delta f_0(m)| = f_s/m^2$). Therefore, more error will arise as the algorithm is subjected to higher frequencies.

In general, fundamental frequency of a speech signals are usually below 1KHz, which does not indicate much error when subjected to the algorithm. But, if the whole range of MIDI is considered, the highest frequency is around 12KHz, which will not be accurately detected. One solution to this problem is oversampling of the audio source, which provides more resolution at higher frequencies. If a signal is sampled twice the CD quality (88.2KHz), the absolute change in frequency goes down by a factor of four in the same range of frequencies (shown in *Figure 3.5a*).



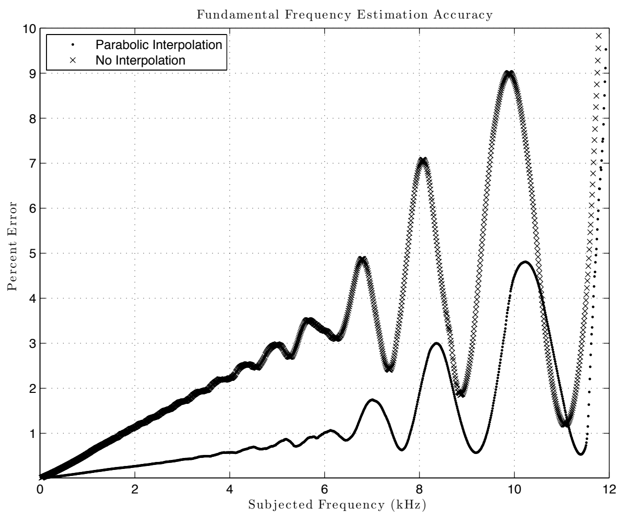
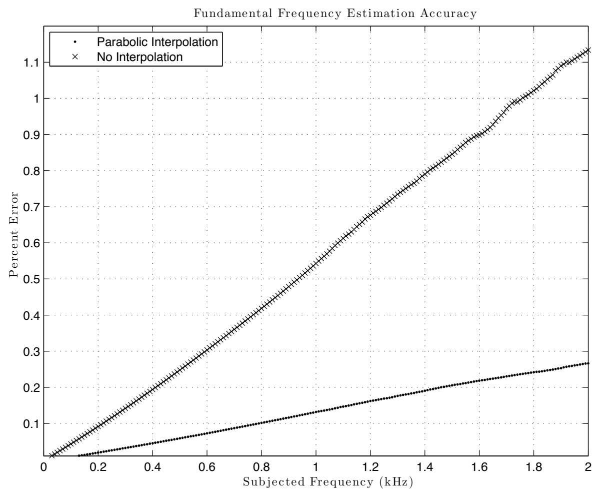


Figure 3.5a (left): Showing absolute change in frequency over a range of correlation factor. The least correlation factor shown corresponds to 20KHz, which is the limit of our hearing range. Notice the decrease of absolute change in frequency over different sampling rate. **Figure 3.5b (right):** Showing percent error of the algorithm using interpolation and without interpolation.

Although increasing the sample rate can lead to a better estimate of the fundamental frequency, it will come at a price in terms of processing and memory cost. Having higher sampling rate requires faster processor to keep up with the overwhelming information, also requires more memory to store them. In order to compromise processing cost and memory storage, we can make the following assumptions. First of all, we can think of ACF as the Fourier transform of the power spectrum.

Therefore, if the input signal is band-limited, we can infer ACF as band-limited as well (Cheveigné & Kawahara, 2002). In addition, ACF is a sum of cosines, which can be approximated as Taylor series, especially near zero with a lower degree polynomial (Cheveigné & Kawahara, 2002). This paper implements a parabolic interpolation, which requires no extra memory, since it only needs to find the minima near the initial dip (shown in Appendix, Section 1). In addition, this method needs very little processing power to calculate, which results into decrease in error by on average 64% below 12kHz (Shown in Figure 2.5b), and 75% below 2kHz (Shown in Figure 2.5c).



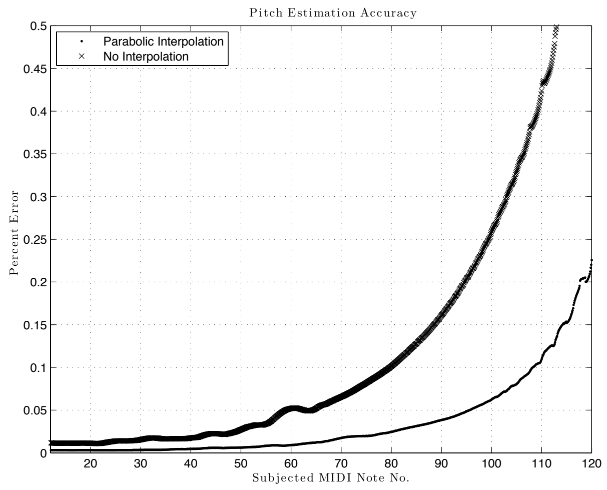


Figure 3.5c (left): Percent error of estimated fundamental frequency shown below 2KHz range. **Figure 3.5d (right):** Showing percent error of pitch, which can also be interpreted as cents

Recall from *Section 2.1*, pitch is logarithmic of base two in terms of frequency. Therefore, the fundamental frequency corresponding to a certain note is geometric, and the change of fundamental frequency over a note is also geometric. Therefore, there is more room for error when trying to detect a musical note. Note from the figure above (*Figure 2.5c and Figure 2.5d*), percent error of detecting fundamental frequency is on average 6% (from 20Hz to 12KHz), while the percent error for detecting a MIDI note is around 2.57% (From 20Hz to 12KHz, and 2.81% non interpolated). In other words, it can be concluded that the average error for estimating pitch is 2.57 *cents*[1]. The tests above were conducted with pure sine tones (using the C Math library) of those given frequencies, using IEEE754 double precision floating point arithmetic.

[1] *A cent is a logarithmic unit for measuring music interval where there are 100 cents between two semitones* (Porter & Mason, 1832).

Section 3.6: Implementing a Monophonic Music Transcriber

Before the algorithm was fully implemented, a great deal of research was done on what parameters should the user have control over. The main goal of this project was to implement an algorithm that can be used for generic purpose of pitch detection, however there were too many parameters involved. Through research, relationships between parameters were derived to simplify for user as much as possible to a user. It was expected that both engineers and musicians could use this product easily, mainly audio engineers. Therefore, the algorithm was given three parameters, which both engineers and musicians can relate to. These parameters are: Minimum Frequency (f_{Min}, measured in Hz), Maximum Frequency (f_{Max}, measured in Hz), and Threshold (T_{dB}, measured in Decibels).

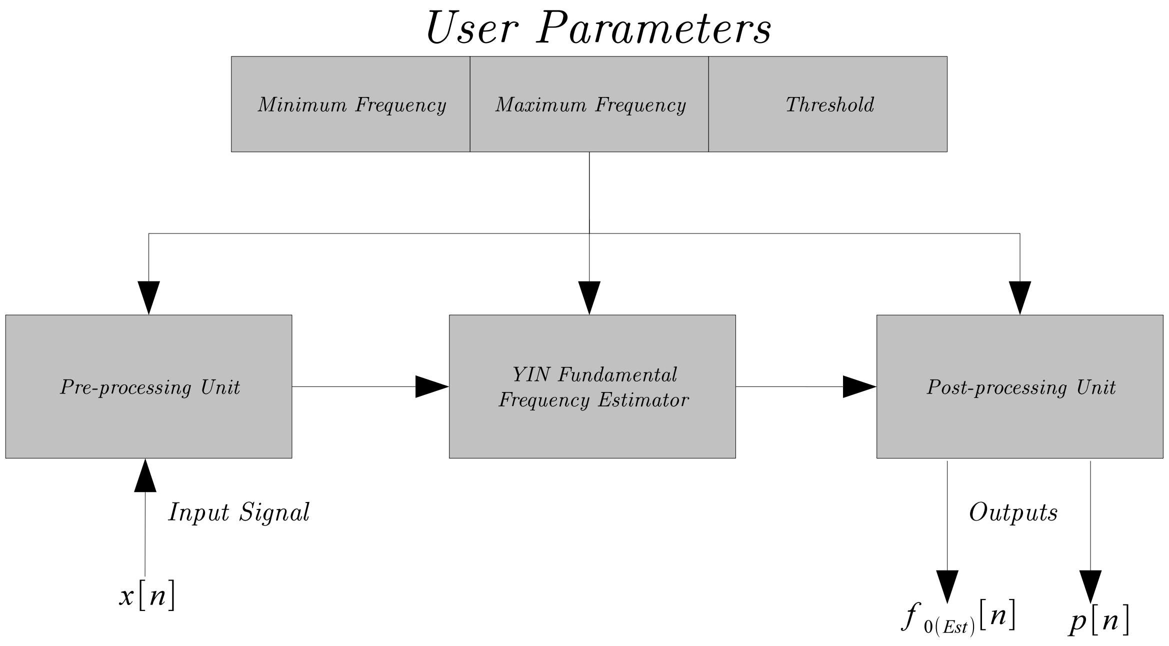


Figure 3.6a: Overall algorithm's high level diagram.

Before implementing the algorithm, it is easier to look at the input-output relationship (shown in *Figure 3.6a*), and draw a high level diagram to simplify the problem. It can be inferred that the algorithm may not function if the signal has low signal to noise ratio, so a preprocessing block is needed before the signal is fed into the algorithm. Also, it can be inferred that the algorithm may not work 100% of the time, which can cause error in the output. Therefore, a post-processing is needed to remove any errors that may have caused by the algorithm. Furthermore, notice from the figure above, in which each block is related to the user parameters to function. In the sections below, each block will be discussed in detail.

Section 3.6.1: Preprocessing

In order to guarantee a successful result, the signals were to pass through several filters to ensure the algorithm a clean signal. Through research, weaknesses of the algorithm were found (described in Section 3.5), and the job of pre-processing block is to minimize limitations as much as possible. Ranges of frequencies were taken from user to ensure the algorithm works in those given range of frequencies. A high–level block diagram is shown in the figure below, which is broken down into fewer blocks discussed in detail.

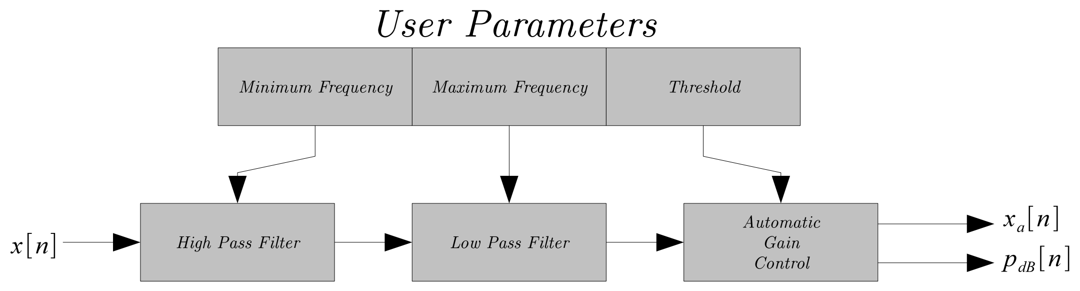


Figure 3.6.1a: Showing the high level diagram of preprocessing block.

(1) Low Pass Filter: A signal may be band-limited at 20KHz, but it was indicated in this paper the error also increases at high frequencies (Section 3.5). Also, note the fact the possibility of a fundamental frequency existing at those range of frequencies are very unlikely. Therefore, the user is given a parameter to assume the highest frequency in that signal. The low pass filter used in this implementation was a 4th order Butterworth, implemented with two cascading 2nd order filters.

(2) High Pass Filter: In order to attain a certain maximum period, a window size is chosen with the user parameter of f_{min} by the algorithm. Since this window size is limited to a certain minimum frequency (maximum period), any frequency component below that can alter the reading. Therefore, a high pass filter is proposed in this paper to remove any frequency component below user requirement. The filter type is the same as the low pass filter, which is a 4th order Butterworth.

(3) Automatic Gain Control: Any musical instrument has two features (as discussed in Section 2.3), of which are the periodical and non-periodical entities. Having non-periodical entities present given to the algorithm can causes errors to estimate (as shown in Section 3.4). Although CMNDF can be very immune to non-periodical entities, such as exponential decay, it is not always perfect. Therefore, this paper proposes an Automatic Gain Control (AGC) filter to be implemented before the signal is passed into the algorithm. The main objective of AGC filter is to keep the signal stationary, thus the gain factor is inversely proportional to its moving average power. The filter that was considered in this project can be expressed as a simple expression shown below:

$$y(t) = A_v x(t) \quad \text{Where, } A_v = \begin{cases} \frac{\alpha}{P_{rms}(t)+\epsilon} & , \bar{P}(t) \le T_{dB} \\ \alpha & , \bar{P}(t) > T_{dB} \end{cases}$$

The filter obtains the RMS power by passing the input signal to a rectifier (absolute value function), then smoothened using a second order Butterworth filter. Also, the gain is smoothened after the comparison of the threshold (from user parameters), since a sudden jump to unity gain can create a jitter in a signal. There are many advantages of this AGC filter than a regular one, since this will not amplify the unwanted noise that is present from the decay of the signal. Just by doing a little threshold (T_{dB}) and gain (α) controlling, the output result is a very clean steady signal that is shown in the figure below (*Figure 3.6.1b*).

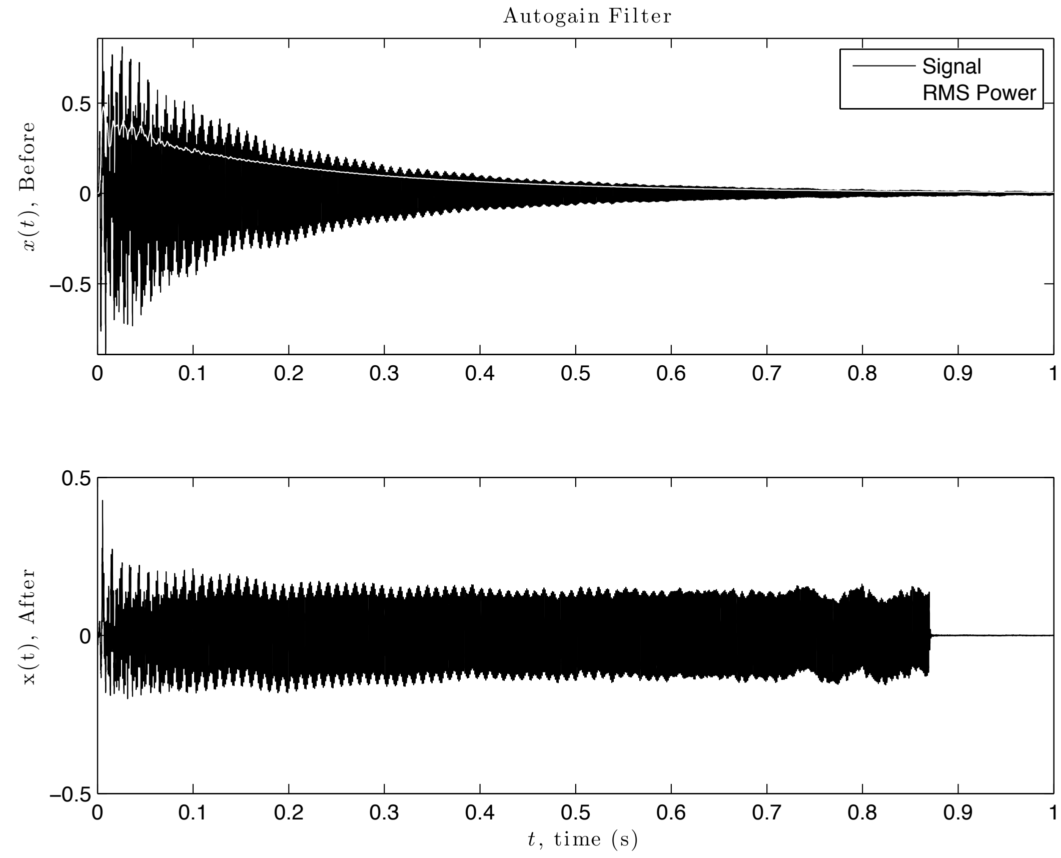


Figure 3.6.1b: Showing output of the auto-gain filter subjected to a decaying A_4 Guitar tone (440Hz). Notice the amplification of noise eliminated with threshold controlling that makes the signal very stationary at given window.

Section 3.6.2: YIN F_0 Estimator (Cheveigné & Kawahara, 2002)

Given a clean stationary signal to the algorithm, this block has to set up a few parameters before it can execute its objectives. At first, this block has to know about the window size to determine the fundamental frequency. Having a small window size will result into the algorithm to completely miss the initial dip, and having too large of a window can accumulate more processing cost and delay. Therefore, the window size is determined by the minimum frequency parameter that is provided by the user. Through experiments and trials, it was found that having a window size of 120% of the number of samples needed to store one cycle of the minimum fundamental frequency was optimal (see Appendix Section 5).

Aside from minimum frequency, maximum frequency provided by the user plays a big role in determining the fundamental frequency, since it can save time for initial dip search (from section 3.4). Given the maximum frequency, there is no need to search the first initial samples for dip. In addition, any harmonic content above the maximum frequencies are assumed filtered out from the preprocessing block (see Appendix Section 4).

The algorithm that has been implemented is slightly modified form the one proposed in Cheveigné & Kawahara's paper. First of all, in order to save processing time, the square function is removed from AMDF calculation. This saves time by removing a multiplication operation, which can accumulate time. In addition, it has been observed from tests that both functions look very similar after calculating CMNDF. Also, this algorithm differs from Cheveigné & Kawahara by having a slightly different search algorithm. They have proposed a search range that user has no control over, which is great for speech processing applications. Since the algorithm proposed in this paper was meant for more general use, more control was given to the user.

The search algorithm may be subjected to failure when searching for CMNDF minima, since the initial dip may not cross under the threshold limit (0.3 as proposed by Cheveigné & Kawahara). One way to increase the algorithm's rate of success is to increase this threshold, which may also lead to error due to jitter preset in CMNDF. Therefore, one assumption that can be made is the fact that: pitch of a signal cannot suddenly change in a length of window period. Therefore, if the algorithm fails to detect the fundamental frequency at certain window period, given the fact that the signal power is above certain threshold (from user parameters), a previous successful fundamental frequency is chosen for that window length. That way, the frequency vector is less jittery if this step was not considered (see Appendix Section 6 for the implementation of this algorithm).

Section 3.6.3: Post Processing

In order to remove any jitter the algorithm may have estimated, this block is used to stabilize the fundamental frequency contour. As discussed before, the fundamental frequency is a discontinuous function that may exist at certain time frames, and on other times there may be no signal present. This is implemented by making the algorithm return an error code (IEEE754's NaN in C++ implementation and -1 in MATLAB's implementation) when there are no signals present. Also, this can create offset values when an LTI[2] system is used to smooth out the frequency vector. Therefore, a different approach is used to remove the jitter from the output signal.

[2] LTI: Linear and Time Invariant

(1) Smoothing Frequency Contour: To prevent jumps and **jitter** in the frequency contour, which is a discontinuous function, a non-linear filter is used to remove it. This project proposed the use of a simple moving average filter, which resets every time the signal goes below threshold (from user parameters), followed by a median filter. The moving average filter used was a very low order filter due to its side effects, which can be shown in the figure below (*Figure 3.6.3a*).

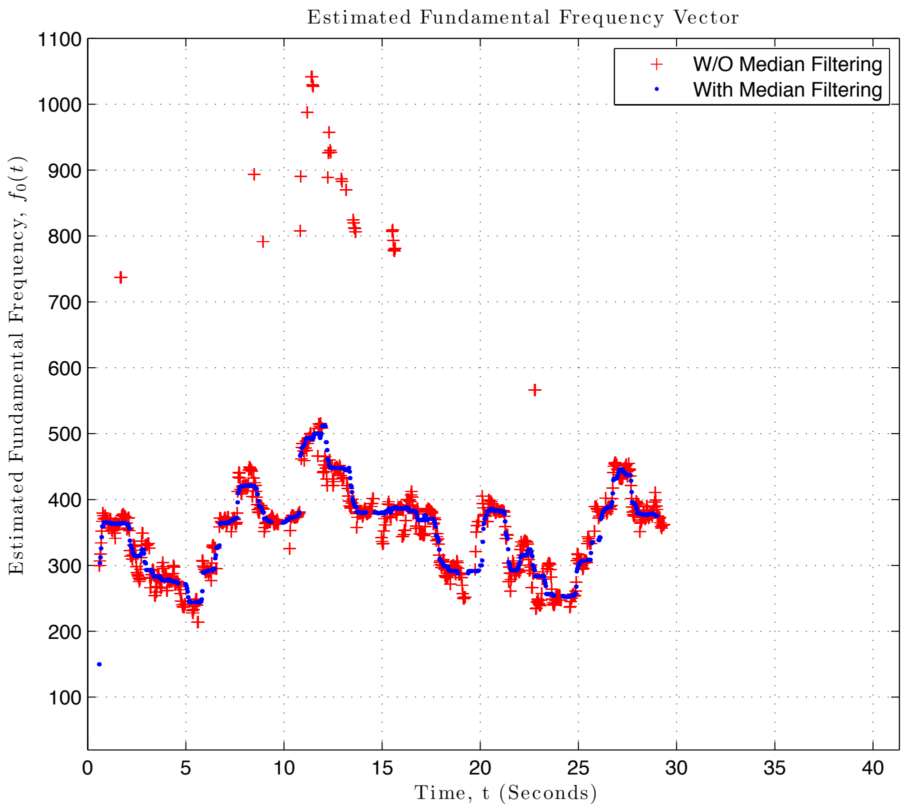


Figure 3.6.3a: The figure shows two frequency contours, the ones denoted by red points are not filtered and the ones denoted by blue have a high order moving average filter (10th order) and high order median filter (17th order). The result of these parameters offsets the frequency contour and removes crucial pitch information.

From this figure, we can see that the data is stabilized, but having a high order moving average filter and can remove crucial pitch information, such as vibrato. Median filter does not remove these effects, but it does contribute to it. However, having no filtering can produce very jittery data that is shown in the figure above. This paper proposed the use of a second order moving average filter followed by a 9th order median filter. This removes any jitter present in the frequency contour, and keeps crucial pitch information.

(2) Visualize Data: Recall from Section 2.5, it was said that transcribing music consisted of two pieces of information. One of this information is pitch contributed by YIN algorithm (Cheveigné & Kawahara, 2002), and the other one is dynamics. As noted before from the AGC filter, the overall average power at specific time can be interpreted as the dynamics information of a piece. It can be represented as colors as MIDI was described in section 2.4. Therefore, this paper implements a simple monophonic music visualizer that takes in the information provided by the algorithm and plots it in time. Consider the figure above (Figure 3.6.3.a), which contains only the pitch information, the dynamics information included in the figure below (Figure 3.6.3b).

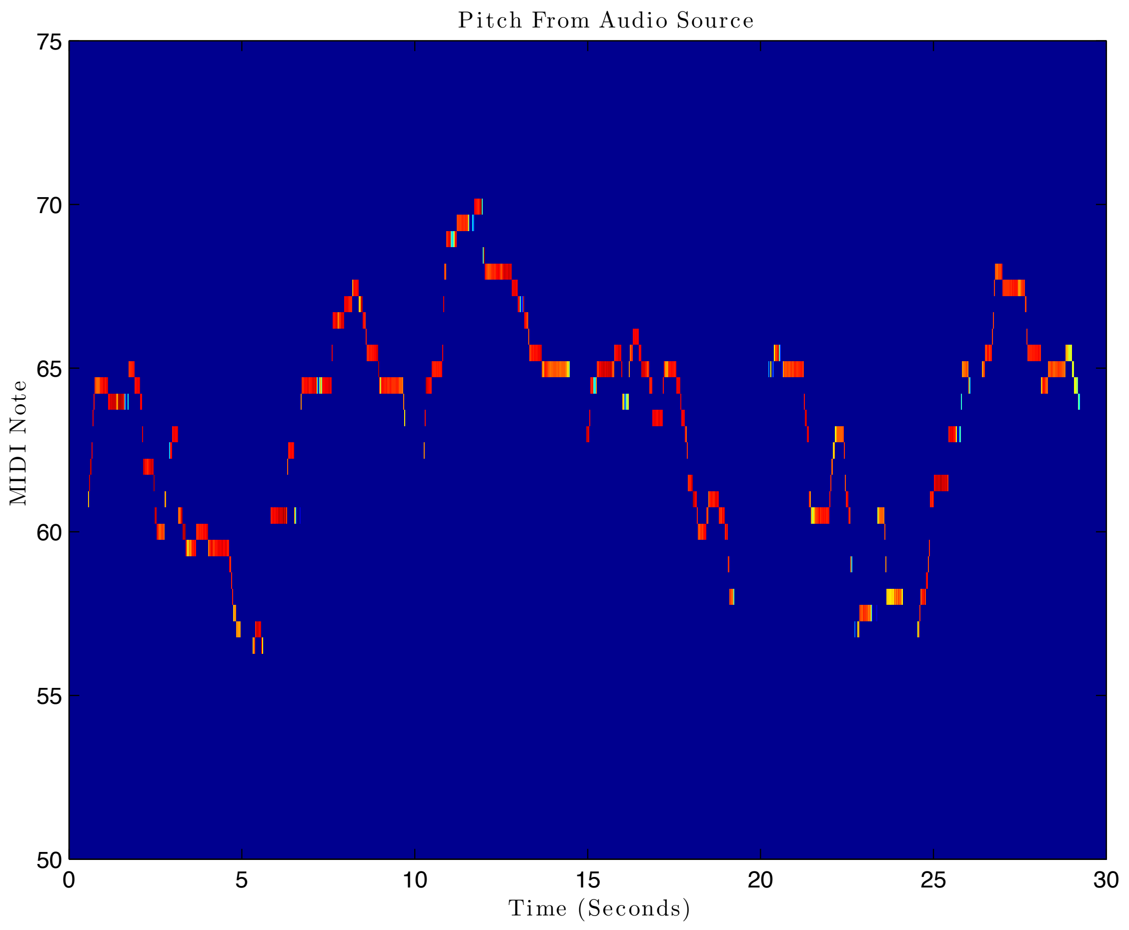


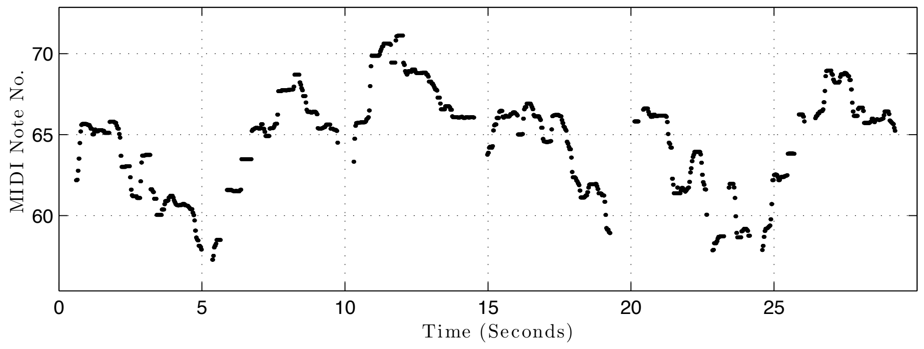
Figure 3.6.3b: Showing the output of the algorithm subjected to a singing voice.

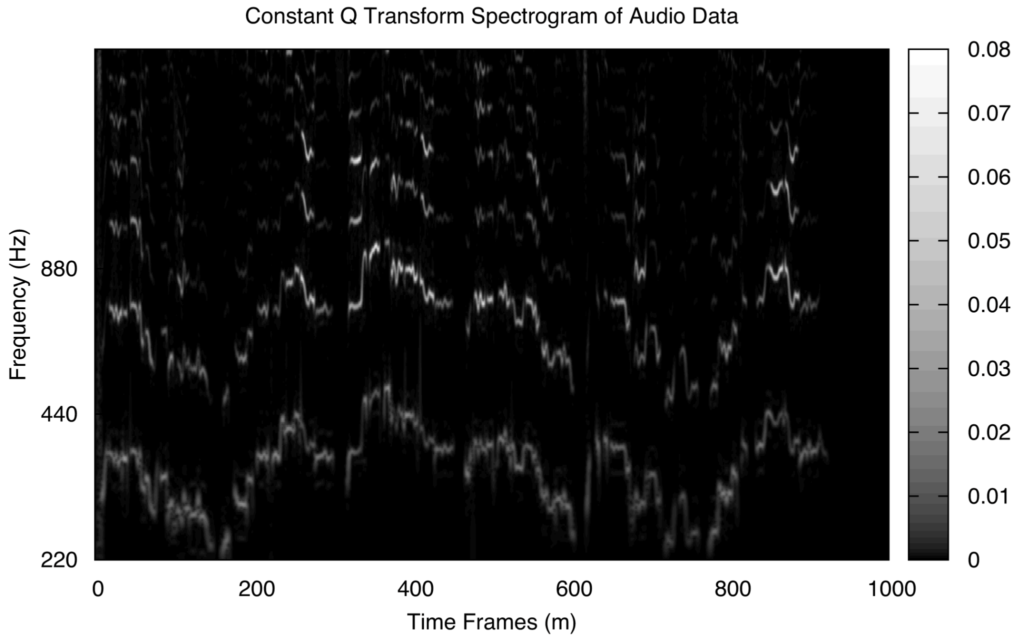
As observed above, the contour looks very similar aside for the fact there are several rounding errors. But, this figure holds more information about the melody than the figure above, since it also includes the loudness of the signal at that given time (louder portions are represented with warmer colors, such as red, yellow, etc. And softer portion represented with cooler color, such as blue, green, etc.). By visualizing the data accordingly, users can have a better idea on how to sing a certain melody. Also, this information can be driven into a music synthesizer, which can be used to add textures to a voice. However, as said before in this paper, over filtering can remove crucial pitch information when synthesizing the voice, but it may also make visualized data look very jittery. Therefore, the visual information was presented to the user with very high order median filter, while the synthesized output was filtered with a low order moving average and median filter.

Chapter 4: Results

It can be argued to test this algorithm with ready-made algorithms out there. Therefore, the tests were conducted with human subjects, where the user was given a sample of speech recording (usually a singing voice of different subjects), and also the synthesized signal. The user was given these signals simultaneously (different channels in stereo) to compare the results. Later on, the user was to give a rating of how accurate the algorithm is, and with 10 test subjects, this algorithm has scored 8.76 out of 10.

Another way to test the algorithm was to compare with a spectrogram, especially Constant Q Transform (CQT) spectrogram (Brown, 1991). This project has chosen that transform over FFT spectrogram since it targets discrete musical frequencies that are presented in section 2.1. By comparing the MIDI notation contour with a CQT spectrogram (implemented with C), it is expected to look very similar at the fundamental frequency, which is shown in the figures below.





Figures 4.1a (top): Showing the frequency contour detected by the YIN Algorithm.
Figure 4.1b (bottom): Showing the Constant Q Transform spectrogram of the same signal.

A second test was conducted with an applet that was written in C++, where the application was subjected to different musical instruments. Assumed these instruments were tuned beforehand, the reading of the output was compared to the note currently played. The users were satisfied with the result and some of them even considered to use this as their primary tuner. The applications user interface is shown in the figure below (Figure 4.1c).

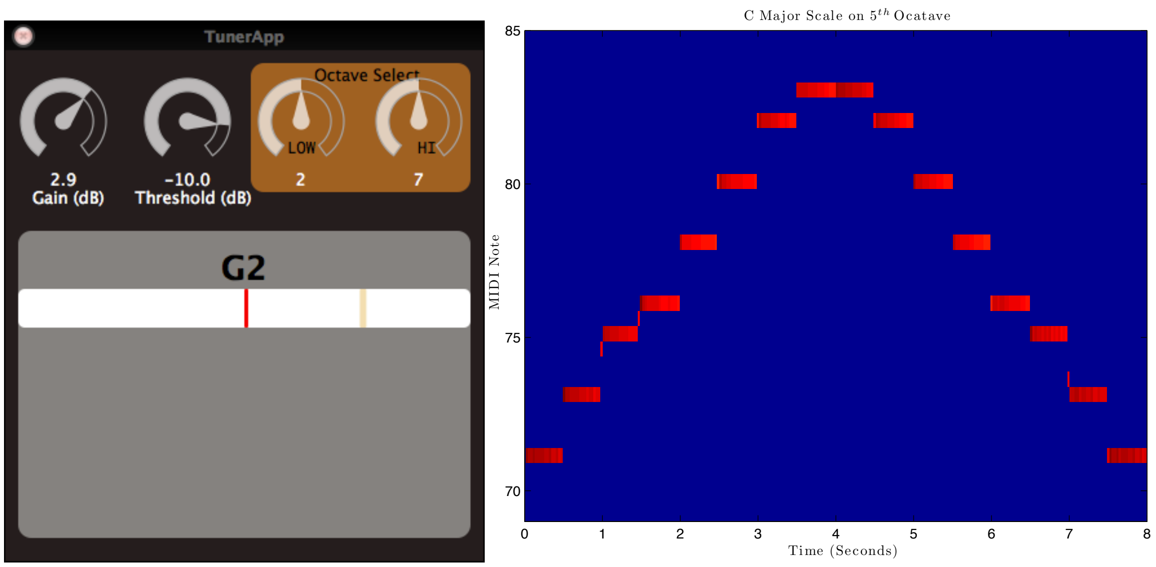


Figure 4.1c: (Left) Real time application of the YIN algorithm using C++. (Right) Output of YIN subjected to a piano playing a C Major scale.

Chapter 5: Conclusion and Future Research

It was possible to implement this algorithm in real time considering the fact that our eyes take about 33.3mS to refresh. This gives us enough time to analyze down to 30Hz signal, which is more than enough for a user to perceive it as real time. According to benchmark testing, the C++ implementation takes around 11±4mS to process through the data, which lets the user to test anything above 55Hz in near real time.

The original goal for this project was to create a polyphonic pitch detection algorithm that was to be used to drive synthesizers in real time. After researching and implementing this algorithm, it was concluded it is not possible with autocorrelation or any FFT analysis. With our ears, a latency of 20mS or more is noticeable, and the desired latency for our auditory senses is around 3 to 5mS, which is the amount of time it takes for the sound of our voice travel to our ears. So, future research will be conducted to resolve *polyphony* and *latency* issues.

Bibliography

Brown, J. C. (1991, January). Calculation of a constant Q spectral transform. *Journal of Acoustical Society of America* , 425-434.

Cheveigné, A., & Kawahara, H. (2002). YIN, a fundamental frequency estimator for speech and music. *Acoustical Society of America , 111* (4), 1917-1930.

Jayant, N., Jhonston, J., & Safranek, R. (1993). Signal Compression Based on Models of Human Perception. *Proceeding of The IEEE , 81* (10), 1385-1422.

Klapuri, A., & Davy, M. (2006). *Signal Processing Methods for Music Transcription.* New York, New York, United States: Springer Science+Business Media LLC.

Lapp, D. *The Physics of Music and Musical Instruments.* Tufts University, Wright Center for Innovative Science Eudcation, Medford, MA.

Midiworld. (1995). *Chopin MIDI Files - Download for free*. Retrieved August 30, 2011, from Midiworld: http://midiworld.com/chopin.htm

Moore, B. C. (1997). *An Introduction to the Psychology of Hearing.* London, United Kingdom: Academic Press.

Porter, W. S., & Mason, L. (1832). *The musical cyclopedia: or, The principles of music considered as a science.* Boston, Massachusetts, United States of America: James Loring.

Tan, L., & Karnjanadecha, M. *Pitch Detection Algorithm: Autocorrelation Method and AMDF.* Prince of Songkhla University, Faculty of Engineering, Hat Yai, Songkhla, Thailand, 90112.

Appendix

Section 1: Parabolic Interpolation for CMNDF

Given a set of three points: $P = \{(x_{-1}, y_{-1}), (x_0, y_0), (x_{+1}, y_{+1})\}$, of which assume (x_0, y_0) to be the minima of that vector. In order to find the period of the signal, a parabolic interpolation must be used. In order to fit three points above the parabolic curve, these were the steps that was taken. The generic equation for parabolic fitting is:

$$y = ax^2 + bx + c$$

With the three given points, we can create a system of equations.

$$\begin{bmatrix} x_{-1}^2 & x_{-1} & 1 \\ x_0^2 & x_0 & 1 \\ x_{+1}^2 & x_{+1} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_{-1} \\ y_0 \\ y_{+1} \end{bmatrix}$$

We can get the solution for a, b, and c if we invert the matrix and multiply with the Y vector, such that:

$$\therefore \begin{bmatrix} x_{-1}^2 & x_{-1} & 1 \\ x_0^2 & x_0 & 1 \\ x_{+1}^2 & x_{+1} & 1 \end{bmatrix}^{-1} \begin{bmatrix} y_{-1} \\ y_0 \\ y_{+1} \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

From this, our solution is:

$$\therefore a = \frac{x_{+1}\,(y_0 - y_{-1}) + x_0\,(y_{-1} - y_{+1}) + x_{-1}(y_{+1} - y_0)}{(x_{-1} - x_0)(x_{-1} - x_{+1})(x_0 - x_{+1})}$$

$$b = \frac{x_{+1}^2(y_0 - y_{-1}) + x_0^2(y_{-1} - y_{+1}) + x_{-1}^2(y_{+1} - y_0)}{(x_{-1} - x_0)(x_{-1} - x_{+1})(x_0 - x_{+1})}$$

Since we know that x_0 will be an integer, and the previus and the next values will be $(x - 1)$ and $(x + 1)$, we can simplify those expressions above, such that:

$$a = \left(-\frac{1}{2}\right)[(x_0 + 1)(y_0 - y_{-1}) + x_0\,(y_{-1} - y_{+1}) + (x_0 - 1)(y_{+1} - y_0)\,]$$

$$b = \left(-\frac{1}{2}\right)[(x + 1)^2(y_0 - y_{-1}) + x_0^2(y_{-1} - y_{+1}) + (x - 1)^2(y_{+1} - y_0)]$$

Note: We do not need to calculate the value for c, since we are only interested in finding the minima, which is denoted as $\tau_0 = -\frac{b}{2af_s}$.

Section 2: AMDF Implementation

C++ Code for AMDF

```
#define DspDataType float
int AMDF(DspDataType* output, DspDataType* input, size_t length) {
        if(length & 0x1) return -1; // If length is odd, no execution
        size_t n, m, k;
        m = k = (length >> 1) - 1;  // m goes from 0 to (N/2 - 1)
        DspDataType tempSum;        // Temoporary register value for storing sum
        while(m + 1) {
                n = k;
                tempSum = 0.0;
                while(n + 1) {
                        tempSum += fabs(input[n] - input[n + m]); --n;
                }
                output[m] = tempSum; --m;
        }
        return 0; // Success!
}
```

Section 3: CMNDF Implementation

C++ Code for CMNDF

```
int CMNDF(DspDataType* output, DspDataType* input, size_t length) {
        if(length & 0x1) return -1;  // If length is odd, no execution
        AMDF(output, input, length); // Calculating AMDF of the vector
        DspDataType sum = 0.0;
        output[0] = 1;
        size_t m = 1, n = (length >> 1);
        while(n) {
                sum      += output[m];
                output[m] = (output[m] * static_cast<DspDataType> (m)) / (sum);
                m++; n--;
        }
        return 0; // Success!
}
```

Section 4: CMNDF Minima Search

C++ Code:

```
DspDataType findMinima(DspDataType* d, DspDataType threshold,
                       DspDataType fs, DspDataType fmax, size_t length) {
       // We do not need to search from zero if fmax is given
        size_t m = floor(fs/fmax) - 1;
        DspDataType tau0 = 1.0f, x, y1, y2, y3, A, B;
        while( !((d[m] <= threshold) && (d[m + 1] > d[m])) ) {
                if(m > length - 1) {
                        tau0 = -1.0f; break;
                }
                m++;
        }
        if(tau0 != (-1.0f)){
           // Step 1: Initialize
           x  = static_cast<DspDataType>(m);
           y1 = d[m - 1];
           y2 = d[m    ];
           y3 = d[m + 1];
           // Step 2: Finding the true minima
           A = -0.5f * ((x + 1) * (y2 - y1) + x * (y1 - y3) + (x - 1) * (y3 - y2));
           B = -0.5f * ((x*x + 2*x + 1)* (y1 - y2) + x*x*(y3 - y1)
                                                   + (x*x - 2*x + 1)*(y2 - y3));
           tau0 = ((-1.0)*B)/(2.0*A*fs);
        }
        return tau0;  // f0 = 1/tau0
}
```

Section 5: Window Length Determination According to f_{min}

Through experiment, it has been observed that minimum window length should be 220% of a cycle. Therefore:

$$W = \frac{1.2}{f_{min}} f_s$$

C++ Code for Window Length Determination

```
......
windowSize = static_cast<size_t> ((1.2f/minFrequency)*samplerate);
......
```

Section 6: MATLAB Implementation

```
function [F0vec power M W P] = PDA_YINv3(input, fs, thresholdDB, fmin, fmax)
% Pitch Detection Algorithm (PDA) using YIN
% Input parameters are:
%   input     - Input signal (must be a single channel or mono audio,
%               audio also must be monophonic)
%   fs        - Sample rate (Hz)
%   threshold - Trigger threshold in dB
%   fmin      - Assumed minimum fundamental frequency of given signal
%   fmax      - Assumed maximum fundamental frequency of given signal
% Function outputs:
%   Fundamental frequency estimated vector
%   M = F0vector in W terms
% ------------------------------------------------------------------
% Usage example:
% >> [x fs] = wavread('file.wav');
% >> fmin = 80; fmax = 1200; threshold = -10;
% >> [f0est power] = PDA_YIN(x, fs, threshold, fmin, fmax);
% returns a vector of length(x) with fundamental frequency
% >> xSynth = YIN_Synthesizer(f0est, power, fmin, fmax, fs);
% Compare..
% >> sound(x, fs); sound(xSynth,fs);
% -----------------------------------------------------------------

% Converting threshold (dB) to scaler unit
  threshold = 10^(thresholdDB/20);
% Calculating window size required from assumed minimum frequency
  W = ceil(1.2*fs*(1/fmin));
  if(isEven(W) == 1), W = W - 1; end;
% Note: According to observations, 120% of minimum frequency window is
% required in order to execute successfully.
% ----------------------------------------------------------------------
% Preprocessing Signal
% ----------------------------------------------------------------------
% Step 1: Filter input using a low-pass-filter. The cut off frequency is
% the maximum assumed frequency passes on by the user.
  [b , a ] = butter(3, fmax/fs, 'low'); % Butterworth gave the best results
  [b2, a2] = butter(3, fmin/fs, 'high');
  input = filter(b ,a , input);      % Applying LPF
  input = filter(b2,a2, input);      % Applying HPF
  input = input./max(input);         % Normalizing output
% Step 2: Use a variable gain filter to saturate the signal
  disp('Preprocessing signal');
  [inputS power] = AutomaticGainControl(input, thresholdDB);
% Finding trigger to activate YIN algorithm
  signalPower = power;
  trigger = power > threshold;
% ----------------------------------------------------------------------
% Pitch detection algorithm
% ----------------------------------------------------------------------
disp('Estimating fundamental frequency using YIN F0 Algorithm');
L = length(input);       % Length of the input signal
n = 0;                   % Discrete time variable
x = inputS;              % x is the conditioned input signal
F0vec = zeros(floor(L/W), 1) - 1; % Estimated frequency table
```

```
F0prv = -1;              % Previous value for F0
for k = 1: ((floor(L/W) - 1)),
    if trigger(n + 1) == 1,
        xTemp = x(n + (1:(2*W)));
        F0vec(k) = YINEstimateF0(xTemp, fmin, fmax, fs, 0.3);
        if F0vec(k) < 0,
           F0vec(k) = F0prv;
        end
        F0prv = F0vec(k);
    else
       F0prv = -1;
       F0vec(k) = -1;
    end
    n = n + W;
end
power = signalPower;
% -------------------------------------------------------------------------
% Post-processing Signal
% -------------------------------------------------------------------------
disp('Post processing F0 vector');
F0vec  = averageFilter(F0vec, 2);
F0vec2 = medianFilter(F0vec, 9);
M = F0vec2;
P = power(1:W:L);
F0vec  = zeros(L, 1);
for n = 1:( floor(L/W)-1),
   F0vec(n*W + (1:W)) = F0vec2(n);
end
% -------------------------------------------------------------------------
% Plotting Data
% -------------------------------------------------------------------------
t = (0:(length(input)-1))./fs;
subplot(4,1,1); plot(t, input, 'b-', t, power, 'r-');
title('Input Signal and Power');
subplot(4,1,2); plot(t, inputS, 'b-', t, trigger.*max(inputS), 'r-');
title('Input Signal after Applying Autogain Filter');
subplot(4,1, 3:4); plot(t, F0vec, 'b.');
axis([0 t(length(t)) fmin max(F0vec)+max(F0vec)/10]); grid on;
title('Estimated Fundamental Frequency f_0 Using YIN');
xlabel('Time (Seconds)'); ylabel('Fundamental Frequency f_0(t), Hz');
end
function [y ag] = AutomaticGainControl(x, thDB)
[b a] = butter(2, 0.02, 'low'); % Creating a smoothing IIR Filter
power = sqrt(abs(filter(b,a,abs(x).^2))); % RMS Power
ag    = power;
th    = 10^(thDB/20);
power(power < th) = 1;
power = filter(b,a, power);
% Changing gain coefficients to 1 if above if below threshold.
offset = 1e-3; % A small value to prevent clipping of signal
gain   =  ((0.5) ./ (power + offset));
gain(gain <= 1) = 1;
gain   = filter(b, a, gain);
y      = tanh((x.*gain)); % Clipping the excessive signal
end
```

```
function f0est = YINEstimateF0(x, fmin, fmax, fs, threshold)
% This function estimates the fundamental frequency of a signal using YIN
% method of fundamental frequency estimation. This function assumes the
% signal give to it is pre-processed and outputs the fundamental frequency
% of one state
% Input Variables:
%       x - Input signal, fs - Sample frequency, W - Window Size,
%       Threshold - for minima search
% Output: fundamental frequency (f0)

if nargin ~= 5,
    error('Inputs are x, fmin, fmax, fs, threshold');
end
% Find the fundamental period using YIN
c = ee221CMNDF(x); % Average Magnitude Difference Function
% Interpolating to give better results
% start on the index corresponding to max frequency
m = floor(fs/fmax);
mMax = ceil(fs/fmin);
f0est = 0;
```

```
while(~( (c(m) < c(m + 1)) && (c(m) < threshold) )),
    m = m + 1;
    if(m > mMax),
        f0est = -1; break;
    end;
end
if(f0est == 0),
    y1 = c(m - 1);y2 = c(m); y3 = c(m + 1);
    A = -0.5 * ((m + 1) * (y2 - y1) + m * (y1 - y3) + (m - 1) * (y3 - y2));
    B = -0.5 * ((m^2 + 2*m + 1)* (y1 - y2) + m^2*(y3 - y1) + (m^2 - 2*m + 1)*(y2 - y3));
    f0est = (2.0*A*fs)/((-1.0)*B);
end
% Return to main function
end
```

```
function d = ee221CMNDF(x)
xd = AMDF(x); d = [1 zeros(1, length(xd) - 1)]; sum = xd(1);
for k = 2:length(xd),
    sum = sum + xd(k); d(k) = xd(k)/(sum/k);
end
```

```
function xdf = AMDF(x)
    if (isEven(length(x)) == 0),
        error('length of the vector has to be an even number');
    end
    N    = length(x)/2; xdf  = zeros(1,N);
    for k = 1:N,
        xdf(k) = (1/N)*sum(abs(x(1:N) - x(k+(1:N))));
    end
end
```

The End