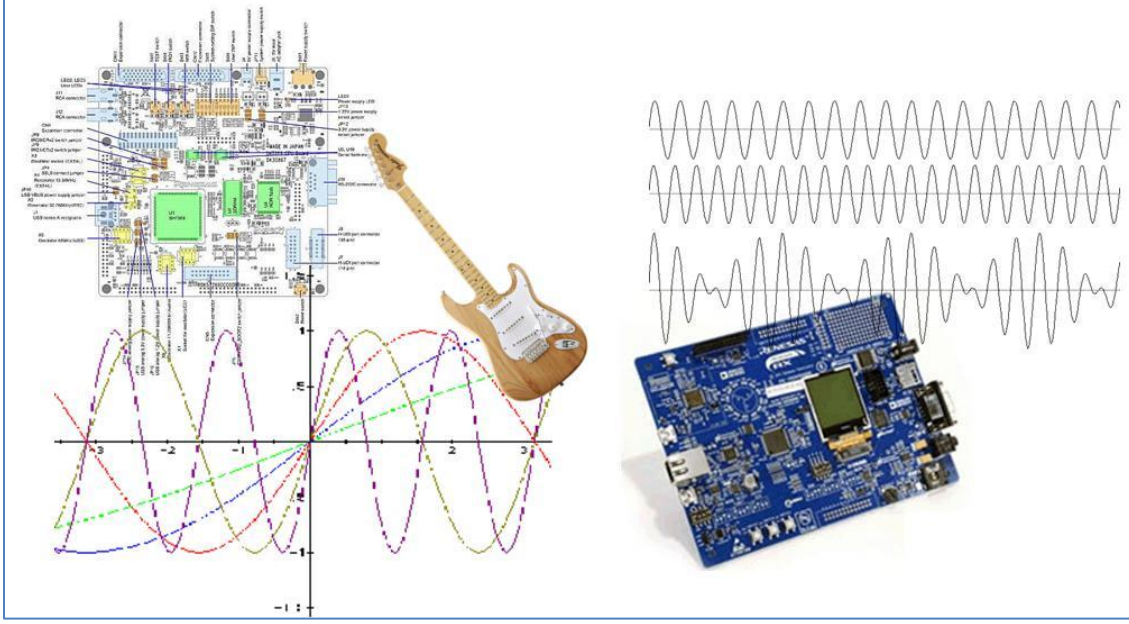
Real-Time Frequency Estimation and Tracking

aka “Let’s Build a Guitar Tuner”



ABSTRACT: Two methods of sinusoidal frequency estimation via adaptive filters are presented. Their MATLAB implementations are ported to an embedded system to provide a real-time application of identifying the fundamental frequency in audio signals. This application could be used as the basis for an instrument tuner.

Contents

1. Introduction

Identification of the frequency of a sinusoidal signal has many applications, eg communications systems, radar and sonar, biomedical data processing, mechanical resonance identification. A more pedestrian but intuitive domain of this type of system identification is commercial audio: frequency analysis is used in identifying, synthesizing, and transforming audio signals for a variety of music applications.

This project demonstrates adaptive filtering as a technique for identifying the fundamental frequency in an audio signal: MATLAB implementations and results provide an illustration of the effectiveness of adaptive filters for analyzing both pure sinusoids and (crudely) recorded instruments. The MATLAB algorithms are ported to a real-time platform to process live signals: a microcontroller evaluation board that integrates microphone, pre-amplifier, and DAC components is used to qualitatively illustrate frequency identification and tracking performance of two adaptive filter approaches to frequency identification.

1.1. Approach

Frequency identification of a signal is a well-studied topic due to its multi-disciplinary application. Indeed, one of the most challenging aspects of this project was limiting the selection of identification techniques from the extensive literature; the landscape of time-domain, frequency-domain, and joint time-frequency approaches[1] is sizable.

Additional criteria was that the algorithms could easily be implemented online, for both MATLAB prototyping of pre-recorded audio (i.e. complete time-sequences) as well as real-time implementation on hardware, for live audio demonstrations.

The final algorithms used in both the MATLAB investigative phase, and the embedded application, are

- Adaptive linear prediction: based on the linear prediction property of sinusoids
- Adaptive IIR notch filter: based on tuning a notch filter to minimize the signal

In addition: a simple zero-crossing algorithm is also examined, primarily to serve as a baseline for verifying both MATLAB audio file processing, and the real-time hardware implementation.

[1] Variations on Pisarenko's method; Fourier transform approaches; autocorrelation methods (AMDF, ASMDF); cepstral analyses; maximum likelihood approaches; as well as numerous papers on the two selected approaches.

2. MATLAB Investigations

In this section, the approach taken is to

- implement the given frequency analysis algorithm in MATLAB code
- Execute the algorithm, and plot the results, using the following audio files:
    - Five different pure sinusoid tones
    - A combination of two sinusoids added together
    - A five-tone melody, played with pure sinusoids
    - A single 440Hz tone played by a guitar
    - A single 440Hz tone played by a keyboard
    - The five-tone melody, played by a guitar
    - The five-tone melody, played by keyboard

We restrict our analysis to only perfunctory observations about the results; our interest is primarily to confirm sufficient correctness of these algorithm implementations, prior to their use in a real-time embedded environment.

2.1. Zero-Crossing

A coarse but simple approach to frequency identification is counting periodic events within the time sequence. *Zero-crossing* identifies when the signal value transitions from a positive to negative value:

- for DC-biased signals, crossing the DC-bias center line could be used
- counting negative-to-positive transitions works just as well

The illustration below is an example of the two variations described above:

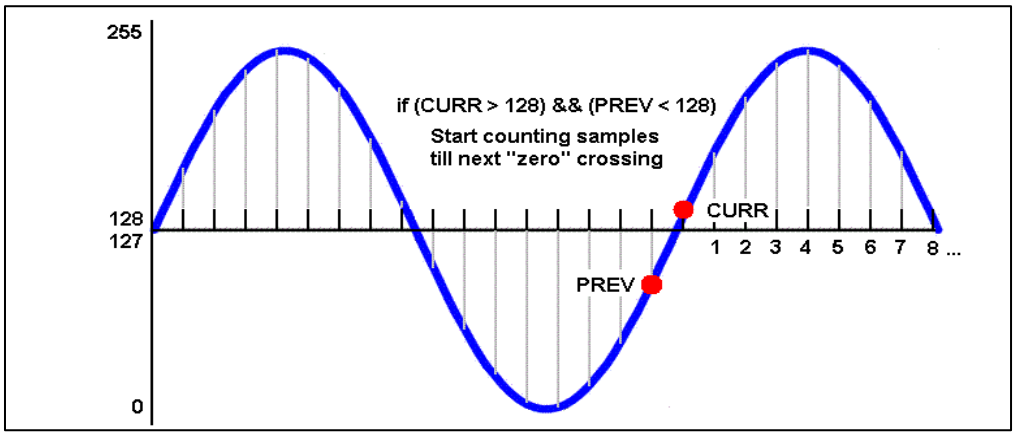


Figure 1. Zero-crossing illustration.

The MATLAB function **zcross** implements a rudimentary zero-crossing algorithm; it populates an array with the running estimate of the frequency of the input signal, and displays the last array value. The implementation is concise enough to display in its entirety, as follows:

```
function [F]=zcross(x,Fs)
%
% ZCROSS - Simple Zero Crossing Algorithm
%
% 'x' assumed to be a periodic unbiased signal
%

N=size(x);
freq=0;
count=0;

% start on 2nd iteration
for n=2:N
    count = count + 1;

    % detect zero crossing
    if ( ( x(n-1) > 0 ) &&  ( x(n) < 0 ) )
      freq = Fs / count;
      count = 0;
    end

    F(n)=freq;

end

F(n)
```

Listing 1. Zero crossing algorithm in MATLAB.

Note the assumption that the input signal, ie the time sequence **x**, is unbiased.

2.1.1. Pure Sinusoid Results

The sinusoidal input data set consists of five input files each containing five different frequencies played for five seconds, sampled at 44.1kHz. The data from each file was processed by the **zcross** function, and a plot of the per-sample frequency estimates, and the final estimate, was obtained[2].

The results of processing the stationary sinusoidal signals are shown below.

INPUT FILE	zcross FINAL OUTPUT (Hz)
100Hz_44100Hz_16bit_05sec.wav	100
250Hz_44100Hz_16bit_05sec.wav	250.5682
440Hz_44100Hz_16bit_05sec.wav	441
1kHz_44100Hz_16bit_05sec.wav	1002.3

[2] Section 6 illustrates the MATLAB commands used to read and process a data file, and plot the results.

10kHz_44100Hz_16bit_05sec.wav	11025

Table 1. Zero-crossing results.

Plots of the **zcross** output of estimated frequencies as processed by each sample input are shown below; these plots provide some insight as to the discrepancies in some of the final estimated frequencies (for brevity, only a subset of each 250k+ sample plot are displayed):

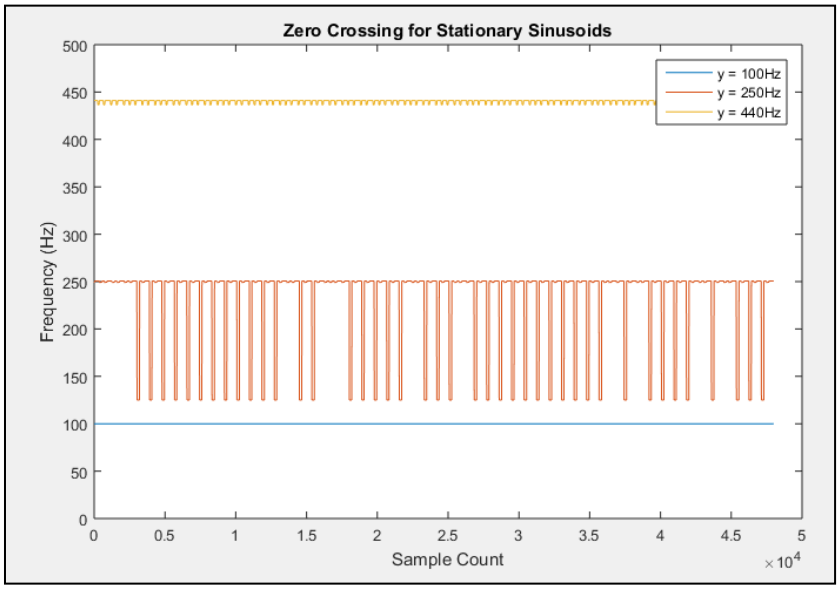


Figure 2. Zero-crossing plot for 100Hz, 250Hz, 440Hz.

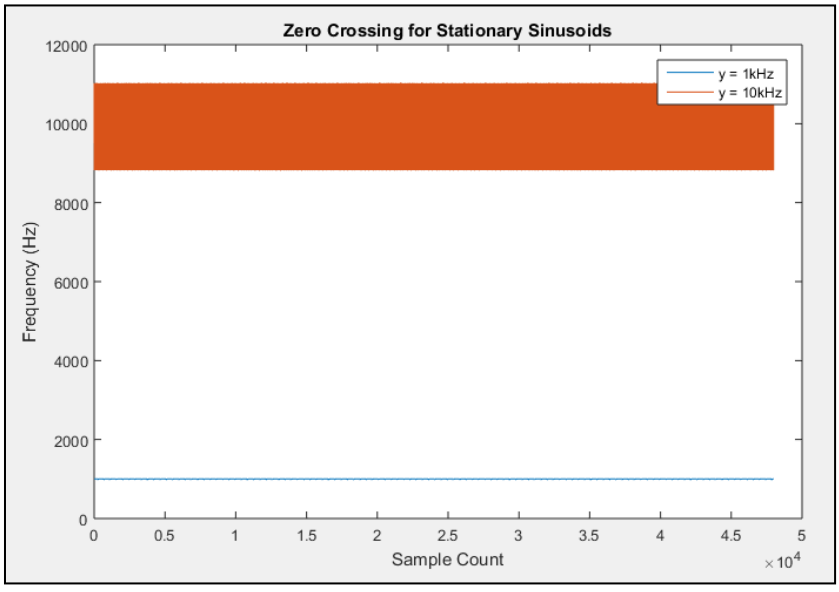


Figure 3. Zero-crossing plot for 1kHz, 10kHz.

These per-sample estimate plots illustrate deficiencies:

- the 250Hz signal shows several instances where the estimated frequency is approximately 125Hz
- There are small variances in the 440Hz signal
- The 10kHz signal shows significant deviations

While our main focus in this project is primarily adaptive filtering, it is interesting to investigate one of the anomalies listed above, for this zero-crossing approach:

The 250Hz plot shows valleys where the estimated frequency is approximately 125Hz. Note that the **zcross** implementation only checks for values above and below zero; examination of the input data shows that these anomalies are most likely due to the sample value being exactly 0.0.

```
% detect zero crossing
if ( ( x(n-1) > 0 ) &&  ( x(n) < 0 ) )
  freq = Fs / count;
  count = 0;
end
```

Listing 2. Original zero-crossing detection code.

A slight modification (given in **zcross2**) can accommodate zero value samples and multiple counting:

```
% detect zero crossing
if ( ( ( ( x(n-1) >= 0 ) && ( x(n) < 0 ) ) || ...
       ( ( x(n-1) > 0 ) && ( x(n) <= 0 ) ) ) && ...
   ( count > 3 ) )
  freq = Fs / count;
  count = 0;
end
```

Listing 3. Modified zero-crossing detection code.

The discriminator (count > 3) ensures that a single zero-crossing event, where the sample value is 0.0, is not counted twice, eg the transition from some positive value +a to 0.0, and from 0.0 to some negative value -b.

The resulting plot of the 250Hz sinusoid as processed by the modified **zcross2** implementation is as follows:

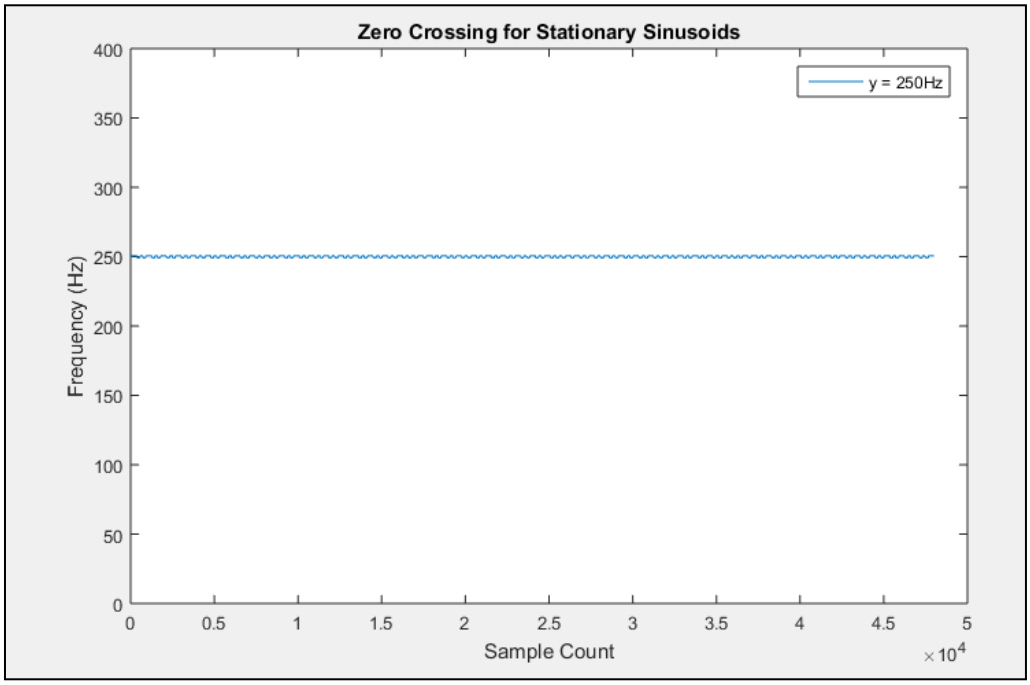


Figure 4. 250Hz sinusoid frequency estimation using zcross2 implementation

There may be further refinements to our zero-crossing algorithm that might alleviate the other anomalies; however, we choose to forego further investigation.

In summary, we observe that, for pure sinusoidal signals, zero-crossing appears to be an effective method of providing fair estimations of frequency.

2.1.2. Combined Sinusoid Results

While effective for pure sinusoidal signals, overtones and harmonics in real audio as generated by instruments augment the fundamental sinusoid; depending on the relative weight of these additional signal components, the zero-crossing approach may be ineffective.

The following diagram illustrates a 440Hz sinusoid combined with a 1kHz sinusoid, sampled at 44.1kHz. As both signals are equally weighted, the maxima and minima are (2, -2).

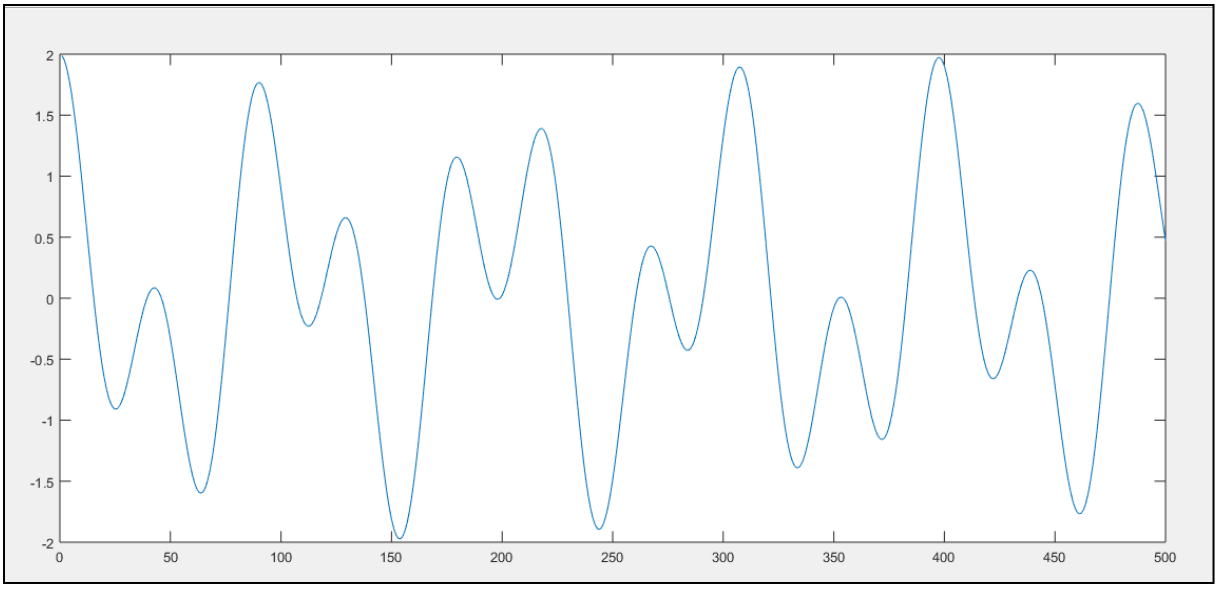


Figure 5. 440Hz + 1kHz sinusoids.

Using the **zcross2** implementation, the resulting frequency estimate is, as expected, wildly inaccurate, since the algorithm cannot differentiate zero-crossing due to the 440Hz component vs the 1kHz component:

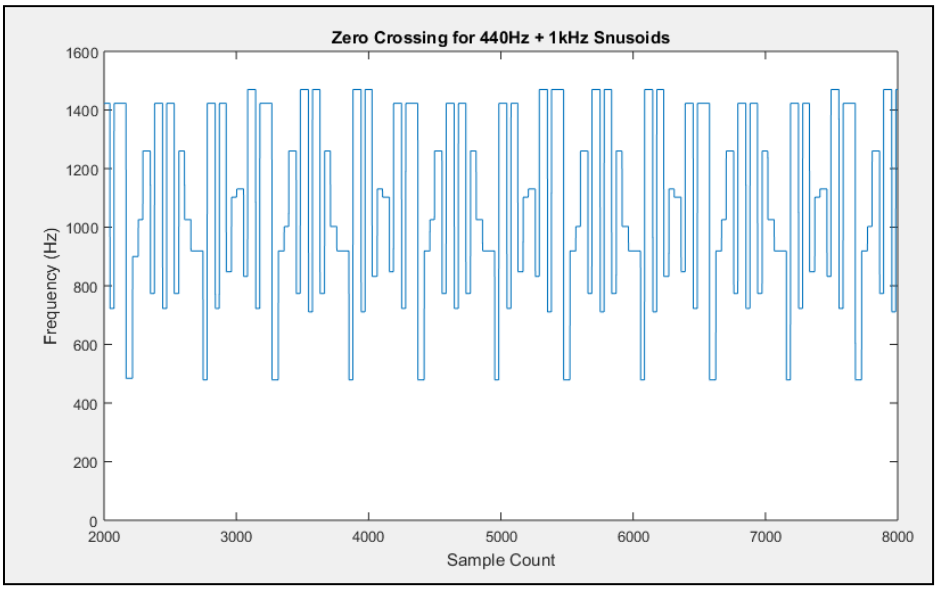


Figure 6. zcross2 estimated frequencies of combined sinusoids.

The closeup, above, does illustrate some intuitive expectations on the limits:

- the maximum detected zero-crossing count is approximately the sum of the frequencies (440Hz + 1kHz = 1440Hz)
- the minimum appears to be no less than 440Hz

2.1.3. Tracking Changes in Pitch

The following plot is the result of the five-tone melody played by pure sinusoids, 1-puretone.wav, estimated by **zcross2**.

The five-tone melody is: A, B, A, G#, A, where A=440Hz. Assuming equal temperament tuning (ie each adjacent pair of the 12 notes in the chromatic scale from root to octave is separated by the same frequency interval ratio, i.e. $2^{1/12}$), the other frequencies are: B= 493.88Hz, and G# = 415.30Hz.

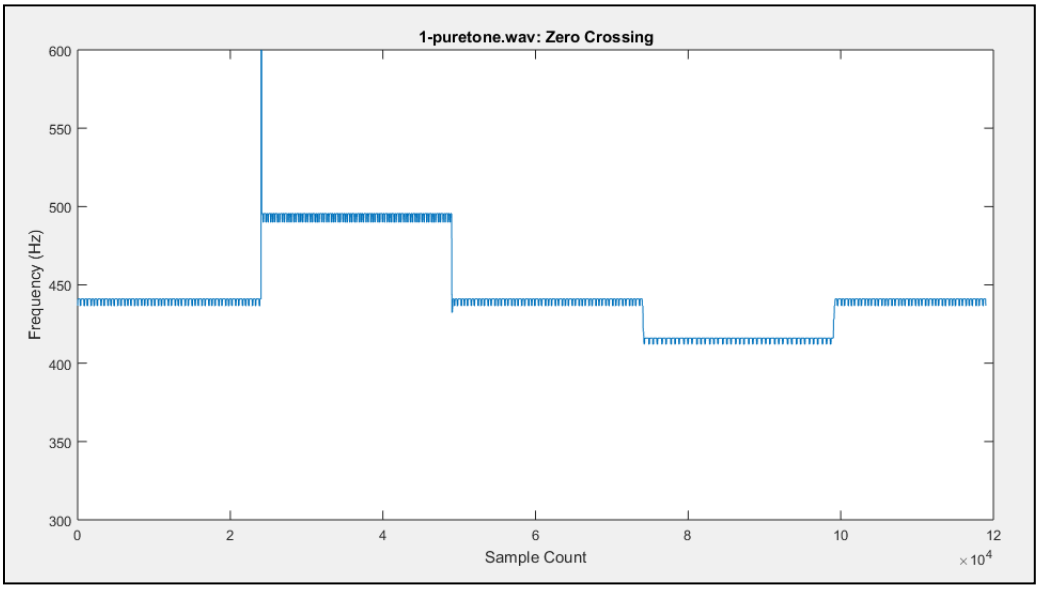


Figure 7. Zero-crossing estimate of five-tone melody.

The plot shows a significant discontinuity in the first tone transition, and variances in all five tones; but otherwise, the algorithm appears to accurately identify the five different tones.

2.1.4. Single-Note Instrument Results

2-guitar.m4a is a single A440Hz tone played on an acoustic guitar; the audio signal is plotted below. The note begins at approximately sample 75,000; recording "hiss" (background noise), represented by the samples from 0 to 75,000, is significant.

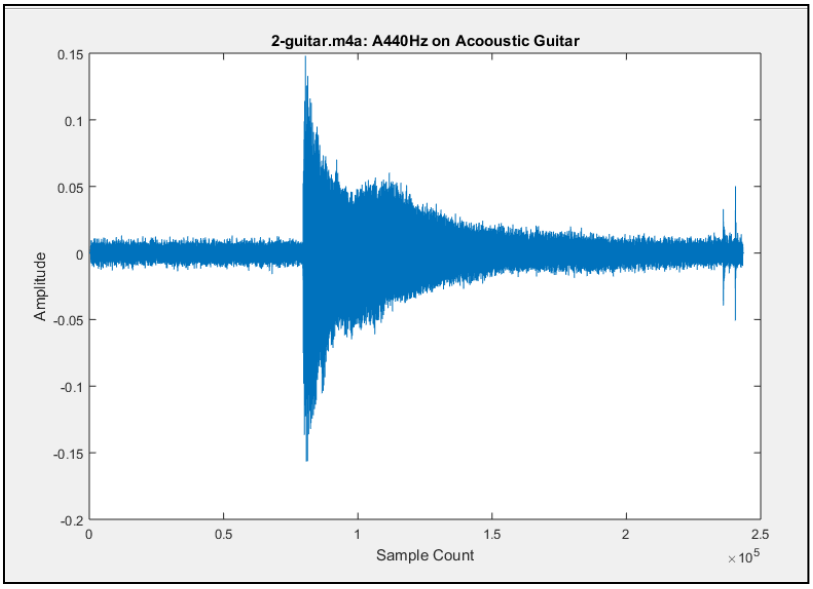


Figure 8. A440Hz played on an acoustic guitar with a low-quality microphone.

As with any instrument: the *timbre*, or tone quality, of the musical note played owes much to the rich spectrum generated by the instrument; this spectrum also renders a simple algorithm such as **zcross2** ineffective, for the same reasons as illustrated with the combined 440Hz + 1kHz signal: the algorithm cannot differentiate zero-crossing events due to the fundamental vs the overtones or harmonics.

The **zcross2** output, shown below, is truncated to only the samples between approximately 80k to 200k. As expected, we do not see a single, clean frequency line; however, there appear to be a pattern of consistent maxima at certain frequencies. We might assume these are the significant overtones generated by the instrument.

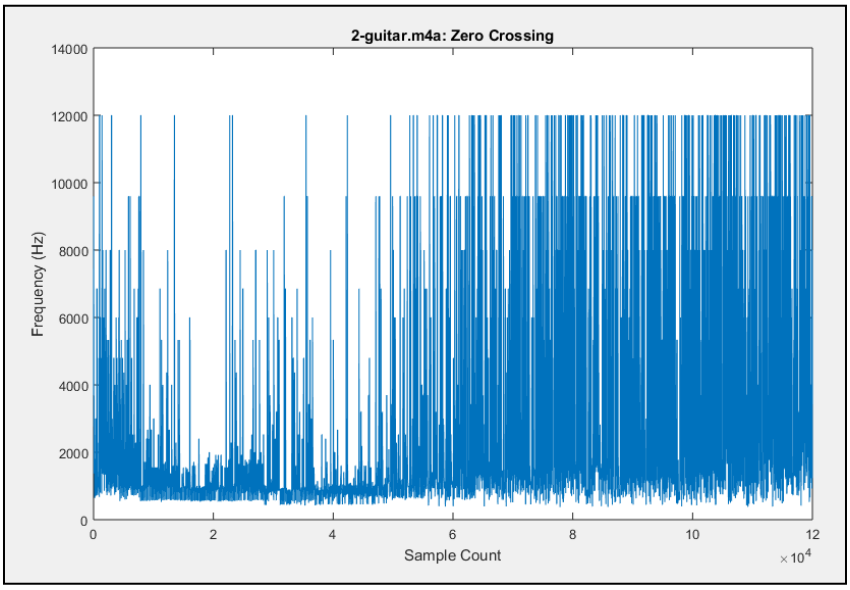


Figure 9. Zero-crossing frequency estimate of acoustic guitar.

Below is a closeup of the **zcross2** output; the purpose here is to show that there appears to be a consistent minimum frequency detected (not quite 440Hz; but there are two periods on the right that appear close to 440Hz).

There also appears to be some consistent detection of a frequency component around the 1kHz area: one guess might be that the sum of the frequencies of the fundamental (440Hz) plus the perfect fifth above the fundamental, E-659.25Hz (which might be the most significant overtone) is around 1099Hz.

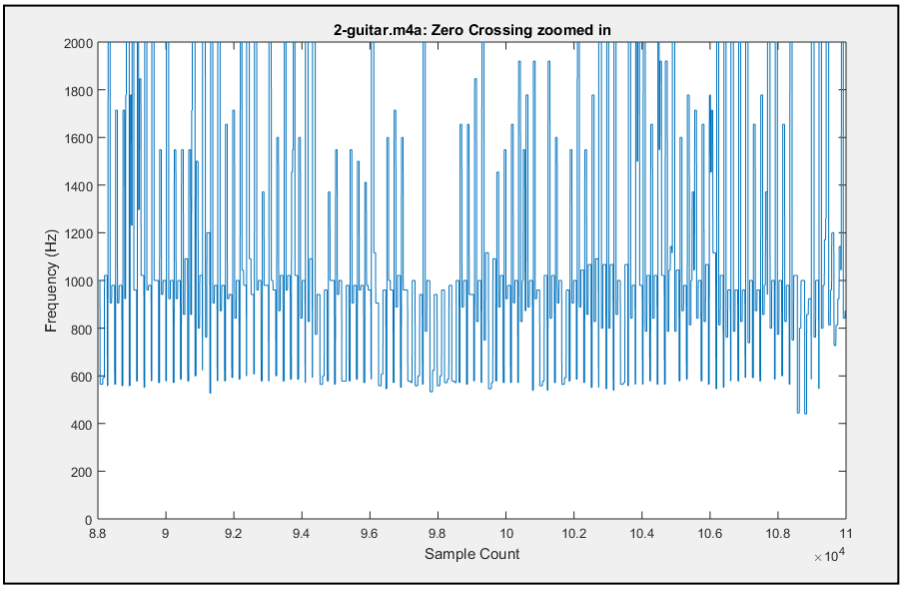


Figure 10. Close-up of zero-crossing frequency estimate of acoustic guitar.

4-keys.m4a is a single 440Hz tone played on a child's toy keyboard; the signal is shown below. The note begins at approximately sample 40,000; recording "hiss" is again significant.

As we might guess, the electronic tone generated by this keyboard has a more controlled envelope than the more rough triangular shape generated by the acoustic guitar:

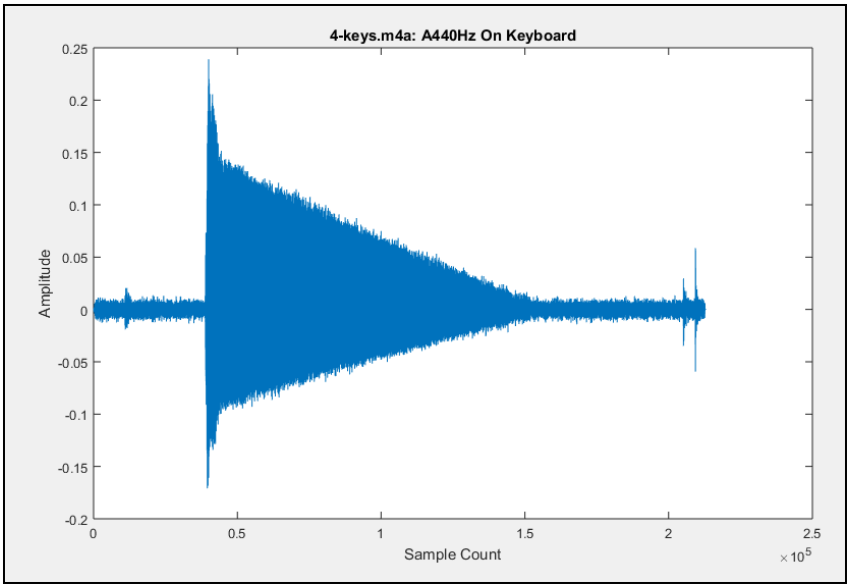


Figure 11. A440Hz played on toy keyboard with a low-quality microphone.

The truncated **zcross2** output, shown below, illustrates consistent detections around 440Hz, 700Hz, and somewhere above 1100Hz:

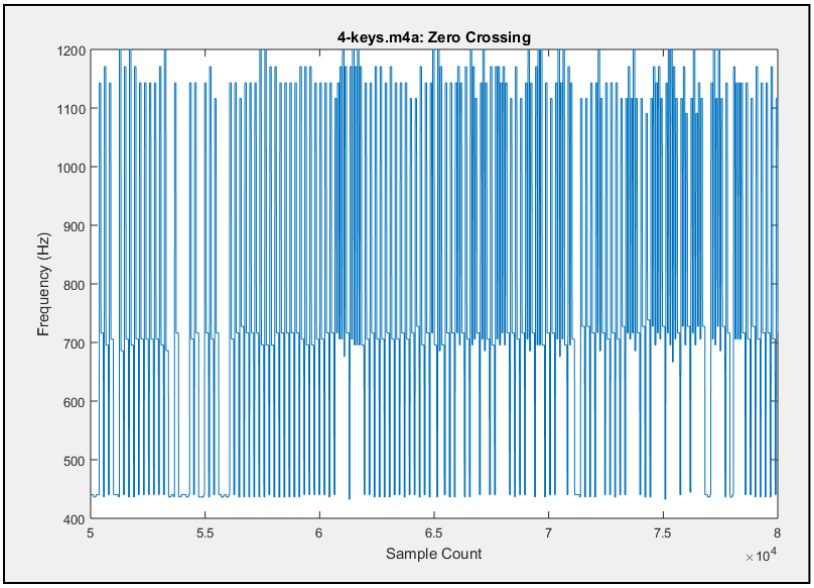


Figure 12. Zero-crossing frequency estimate of toy keyboard.

And a close-up of the **zcross2** output shows a definite trend along 440Hz, as well as around approximately 436Hz:

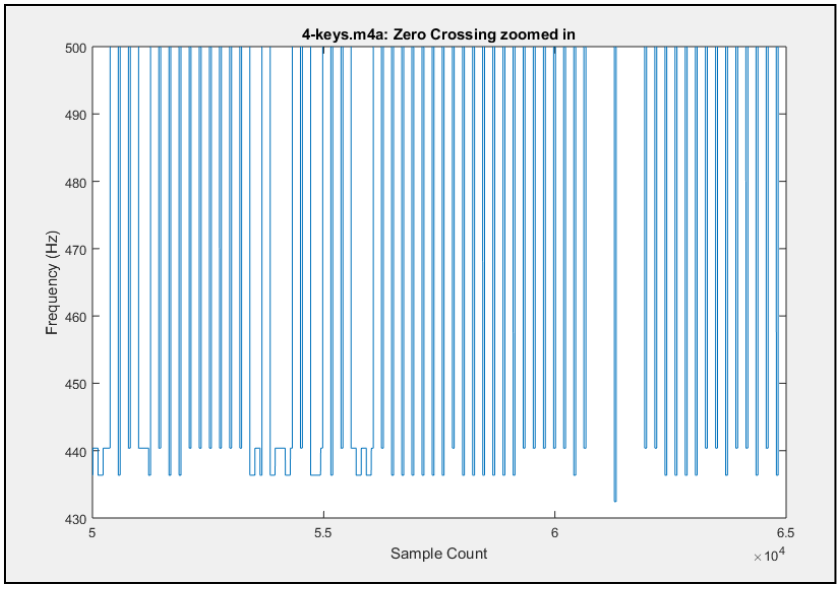


Figure 13. Close-up of zero-crossing frequency estimate of toy keyboard.

2.1.5. Pitch Change Instrument Results

Recall that the five-tone melody used earlier is: A, B, A, G#, A corresponding to frequencies: 440, 493.88, 440, 415.30, 440.

3-guitar.m4a is the five tone melody played on acoustic guitar; the plot below shows the **zcross2** frequency estimate results (the plot prior to the 100k-sample time, and after the 300k-sample time, is ambient noise):

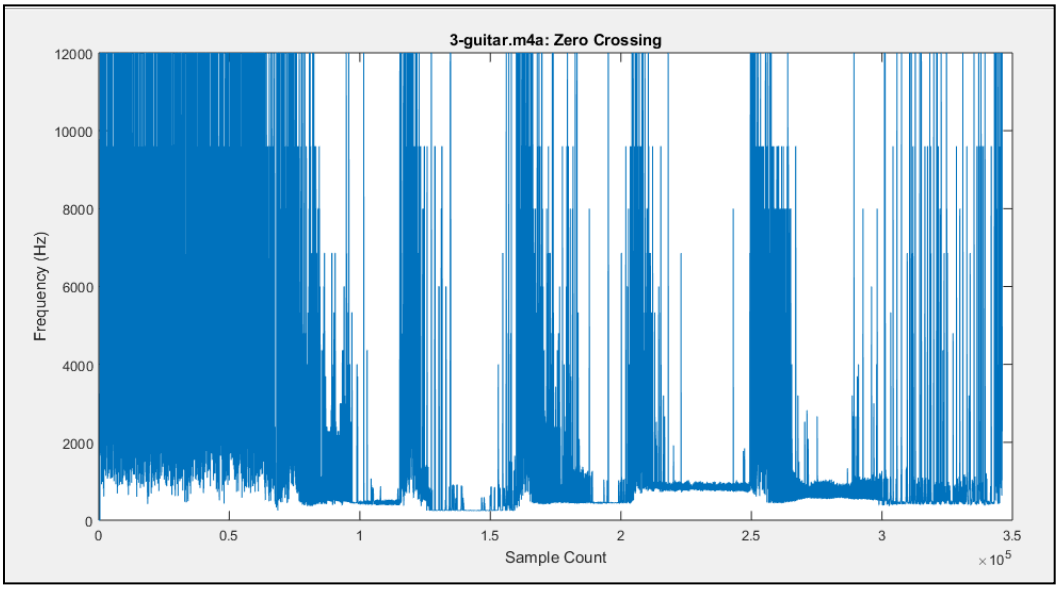


Figure 14. Zero-crossing frequency estimate of five-tone melody played on acoustic guitar.

A closeup of the results, below, shows some pattern; and the minimum frequencies detected are within the same order of magnitude as the original tones; but there generally is no accuracy to the frequency identification:

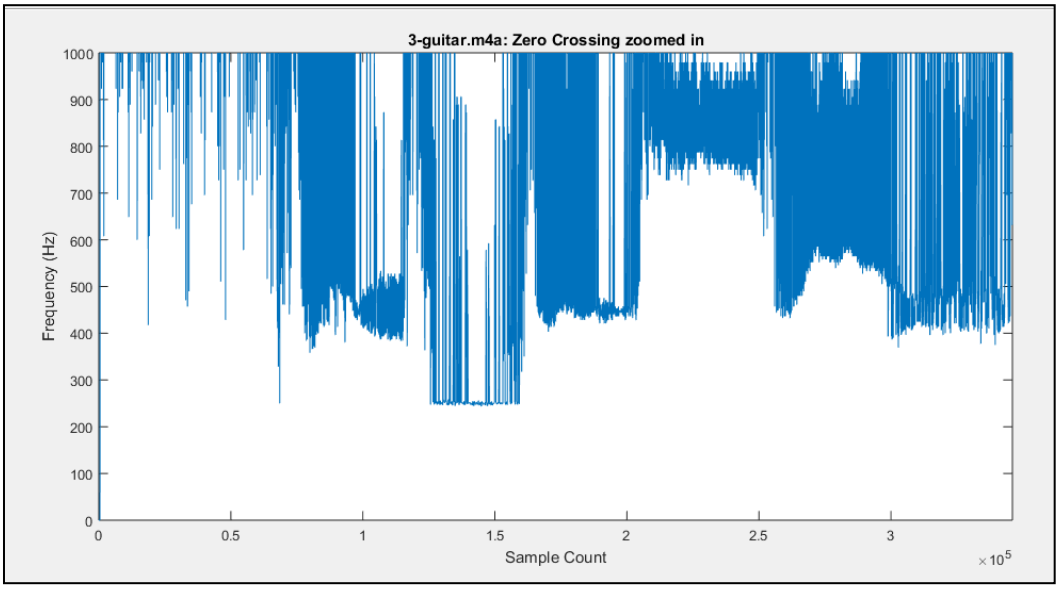


Figure 15. Close-up of zero-crossing frequency estimate of five-tone guitar melody.

5-keys.m4a is the five tone melody played on the toy keyboard. Below is the **zcross2** plot of the keyboard tones, where the samples prior to approximately 4,000-sample point and after 30k-sample point is ambient noise:

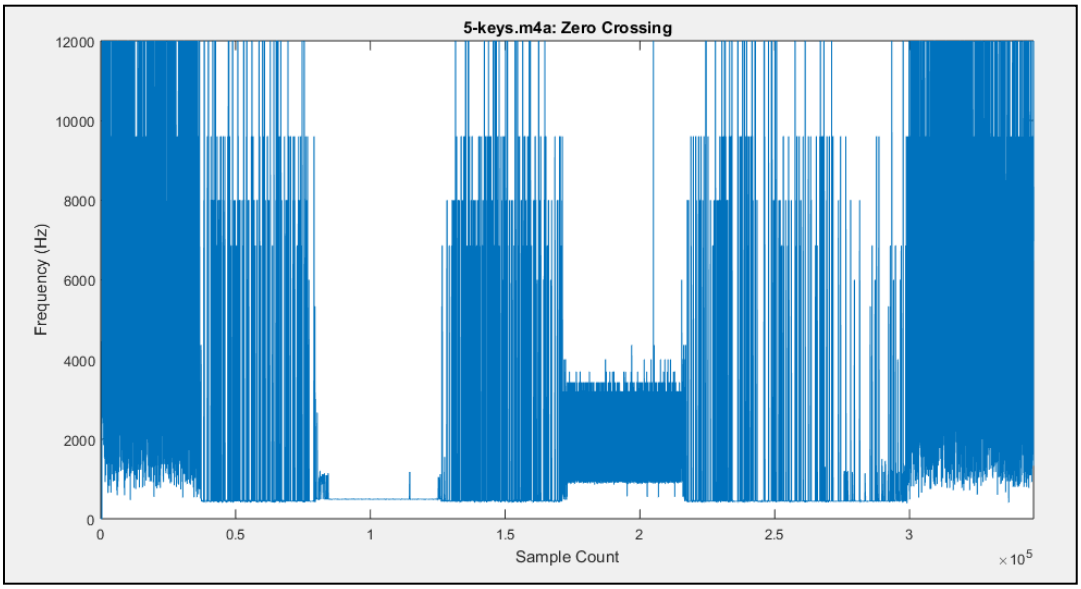


Figure 16. . Zero-crossing frequency estimate of five-tone melody played on keyboard.

And below is a closeup of the minima. We observe that the A and B frequencies (440Hz, 493.88Hz) appear consistent and well-defined; the G# tone (415.30Hz) is, however, detected as an approximate 1kHz minima:

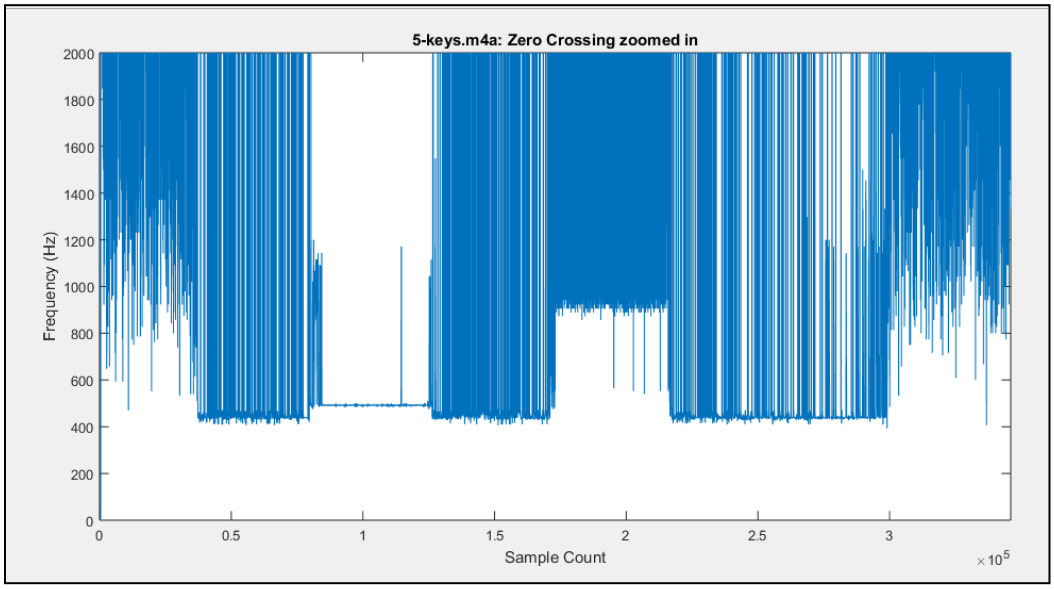


Figure 17. Close-up of zero-crossing frequency estimate of five-tone keyboard melody.

We theorize that, while very inaccurate for instruments as compared to a pure sinusoidal input, the zero-crossing algorithm nonetheless shows some minor success with instruments that (presumably) have a strong fundamental component as compared to its overtones (which is what we assume the electronic keyboard demonstrates).

2.2. Adaptive Linear Prediction

The algorithm defined in [1] is based on the linear prediction property of sinusoids: given a discrete-time sinusoidal signal s_n

$$s_n = \alpha \cos(\omega n + \phi)$$

it can be shown[3] that s_n follows the linear prediction property

$$s_n = 2\cos(\omega)s_{n-1} - s_{n-2}$$

where

$$s_{n-1} = \alpha \cos(\omega(n-1) + \phi)$$

[3] Shown in section 8

$$s_{n-2} = \alpha \cos(\omega(n-2) + \phi)$$

The approach in [1] then derives an expression for the Mean Square Error (MSE), differentiates with respect to the frequency ω to obtain the gradient expression, and then obtains the recursive LMS equation for estimating ω.

A brief summary of algorithm is as follows:

Given a discrete-time noisy sinusoid with time sequence $\{x_n\}$ modeled as

$$x_n = \alpha \cos(\omega n + \phi) + q_n \equiv s_n + q_n$$

the estimated signal $\hat{s}_n$ can be predicted using the two previous samples

$$\hat{s}_n = 2\cos(\hat{\omega}) x_{n-1} - x_{n-2}$$

The error function is

$$e_n \equiv x_n - \hat{s}_n$$

the Mean Square Error (MSE) function $E[e_n^2]$ is derived as

$$E[e_n^2] \equiv 4\sigma_s^2 \left(\cos(\hat{\omega}) - \cos(\omega)\right)^2 + 2\sigma_q^2 \left(2 + \cos(2\hat{\omega})\right)$$

where $\sigma_s^2 = \dfrac{\alpha^2}{2}$ is the signal power.

The MSE expression is simplified as follows: assuming the expression $\left(2 + \cos(2\hat{\omega})\right)$ is a constant, the MSE can be expressed as a scaled expression of $E[e_n^2]$, denoted by $E[\zeta_n^2]$, as follows:

$$E[\zeta_n^2] \equiv \frac{E[e_n^2]}{2(2 + \cos(2\hat{\omega}))}$$

$$= \frac{2\sigma_s^2 \left(\cos(\hat{\omega}) - \cos(\omega)\right)^2}{2 + \cos(2\hat{\omega})} + \sigma_q^2$$

and minimizing the expression $E[\zeta_n^2]$ is considered equivalent to minimizing $E[e_n^2]$.
The instantaneous value of $E[\zeta_n^2]$ is ζ_n^2:

$$\zeta_n^2 = \frac{e_n^2}{2(2+\cos(2\hat{\omega}_n))}$$

Differentiate the above with respect to $\hat{\omega}_n$ to obtain the stochastic gradient estimate:

$$\frac{\partial \zeta_n^2}{\partial \hat{\omega}_n} = \frac{2\sin(\hat{\omega}_n)}{\left[2(2+\cos(2\hat{\omega}_n))\right]^2} e_n\left[(x_n + x_{n-2})\cos(\hat{\omega}_n) + x_{n-1}\right]$$

$$\frac{\partial \zeta_n^2}{\partial \hat{\omega}_n} \approx e_n\left[(x_n + x_{n-2})\cos(\hat{\omega}_n) + x_{n-1}\right]$$

The product term $\frac{2\sin(\hat{\omega}_n)}{\left[2(2+\cos(2\hat{\omega}_n))\right]^2}$ is positive for the range $\omega \in (0, \pi)$, so it does not affect the sign of the gradient estimate and can be considered as incorporated into the step size factor μ. Therefore the LMS update equation is given by

$$\hat{\omega}_{n+1} = \hat{\omega}_n - \mu e_n\left[(x_n + x_{n-2})\cos(\hat{\omega}_n) + x_{n-1}\right]$$

This recursive equation is the basis for the MATLAB **dfe** function and the subsequent **dfe.c** code used in the embedded implementation.

Since this algorithm is based on minimizing the mean square error value of the linear prediction error function characterized by the estimated frequency, it is considered a *direct frequency estimation* (DFE) algorithm.

The source code for the MATLAB **dfe** function implementing the DFE algorithm is not much longer than the zero-crossing algorithm; it is given below. Similar to the **zcross** code, this function returns an array of the estimated frequencies based on each input sample:

```
function [F]=dfe(x,Fs)
%
% DFE - Direct Frequency Estimation
%
% "Adaptive Algorithm for Direct Frequency Estimation"
% H.C. So, P.C. Ching
%

N=size(x);
mu=10^-2;           % LMS step size
theta=0.25;         % assume initial frequency pi/4, normalized
```

```
% we need the first two samples, so the loop starts at 3
for n=3:N
    % compute the predicted signal from past two samples
    s_n = 2 * cos( theta ) * x(n-1) - x(n-2);

    % compute the error e_n
    e_n = x(n) - s_n;

    % compute the new angle
    theta = theta - ( mu * e_n * ( ( x(n) + x(n-2))*cos(theta) + x(n-1) ) );
    F(n)= theta / ( 2 * pi ) * Fs;
end

F(n)
```

Listing 4. Adaptive Linear Prediction aka Direct Frequency Estimation (DFE) algorithm in MATLAB.

Note the arbitrarily selected parameters **mu** (the LMS step size) and **theta**, an initial frequency value.

2.2.1. Sinusoid Results

The results of stationary sinusoidal signals are shown below. These results are much more accurate than the **zcross** algorithm.

INPUT FILE	dfe FINAL OUTPUT (Hz)
100Hz_44100Hz_16bit_05sec.wav	100.0001
250Hz_44100Hz_16bit_05sec.wav	250.0000
440Hz_44100Hz_16bit_05sec.wav	440.0004
1kHz_44100Hz_16bit_05sec.wav	1000.0
10kHz_44100Hz_16bit_05sec.wav	1000

Table 2. DFE results.

Plots of the **dfe** output of estimated frequencies are shown below:

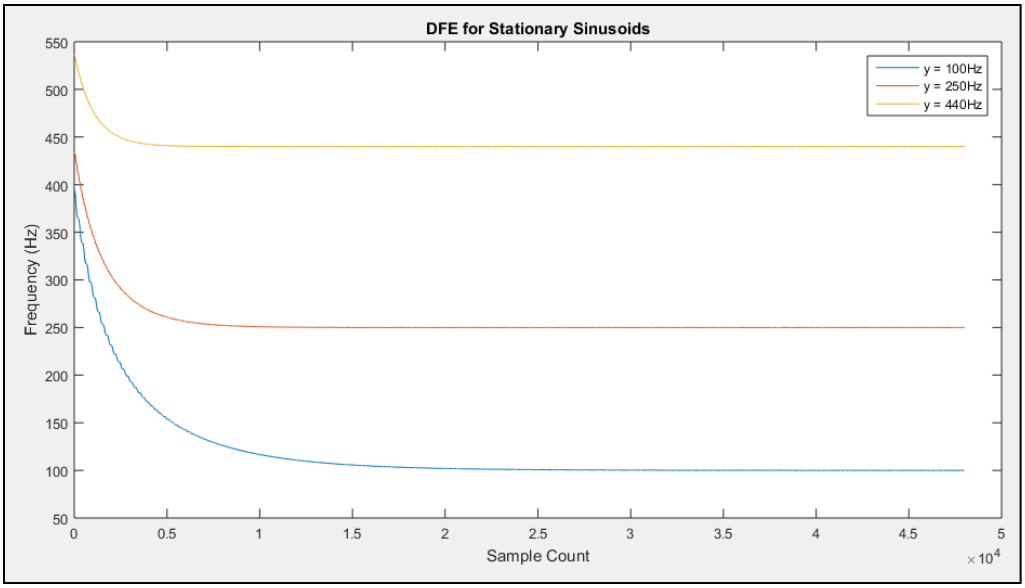


Figure 18. DFE plot for 100Hz, 250Hz, 440Hz.

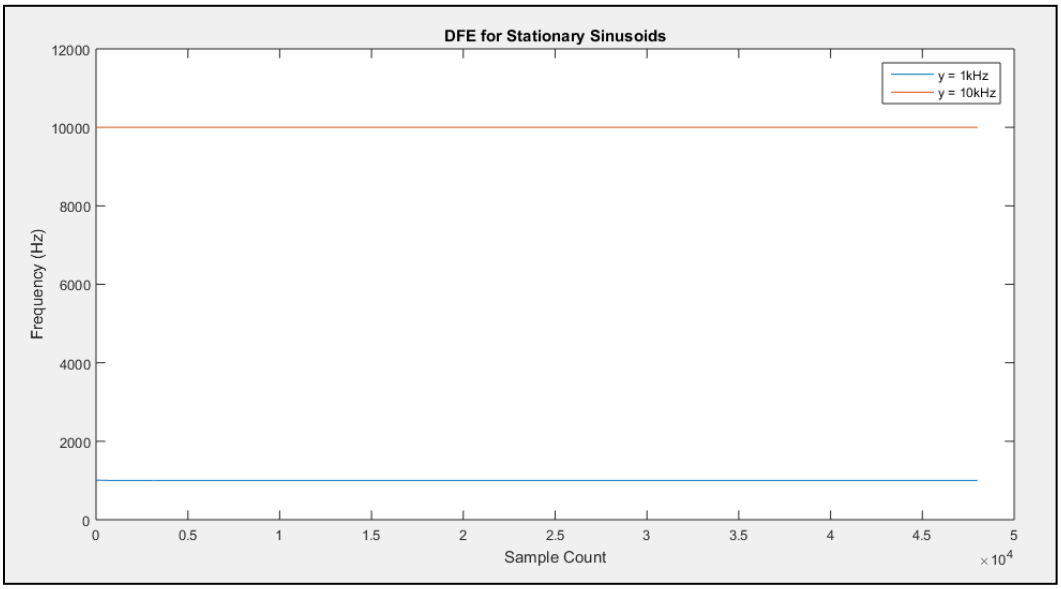


Figure 19. DFE plot for 1kHz, 10kHz.

Observe the relatively slow convergence as compared to the zero crossing algorithm (the first 2,000 samples were truncated, as well; this explains why the 1kHz, 10kHz plots appear to already have converged). The step size **mu** was arbitrarily selected; no attempt was made to improve convergence time.

Observe further, however, that the frequencies appear consistent; there is no variance, as seen in the zero-crossing algorithm plots.

The plot below shows the results of processing the combined 440Hz + 1kHz sinusoids with the **dfe** function. Note that the plot shows a much smaller variance than the zero-crossing algorithm. As **dfe** is intended to divulge the frequency of a single sinusoid, we do not expect the results to reflect either purely 440Hz or 1kHz.

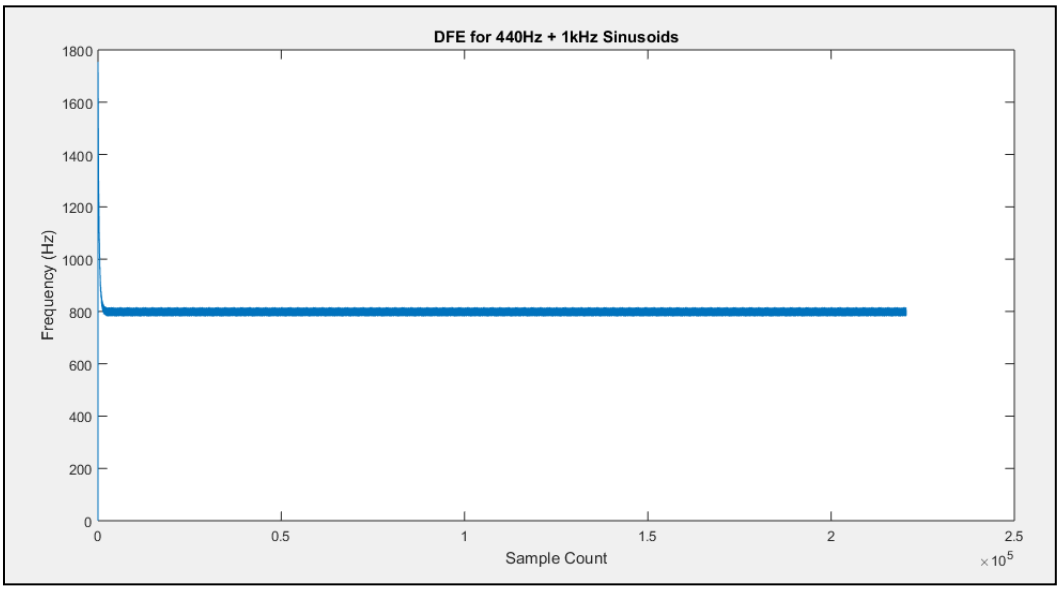


Figure 20. DFE estimated frequencies of combined sinusoids.

The following plot of the five-tone melody as played by pure sinusoids appears quite accurate; note also the convergence curves:

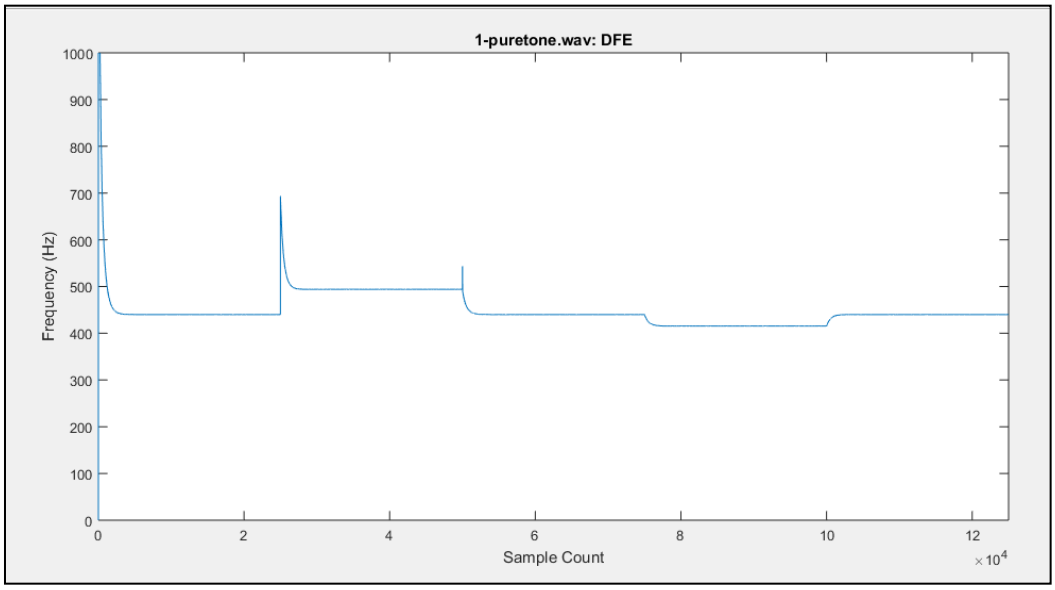


Figure 21. DFE estimate of five-tone melody.

2.2.2. Instrument Results

As with the combined sinusoid signal, we don't expect the instrument results to be very good, due to the complex overtones.

The following results of the A-440Hz on acoustic guitar appear consistent, but incorrect. Perhaps the algorithm is identifying a strong overtone? The frequency appears to be around 1650 Hz; the octave above A-440Hz is A-880Hz; the ratio of 1650Hz to 880Hz is 15:8, which is a major seventh interval. Not sure why we'd identify a major seventh interval; it's possible the guitar isn't tuned to convert pitch, and instead we are identifying the second octave above a (flat) A-440Hz:

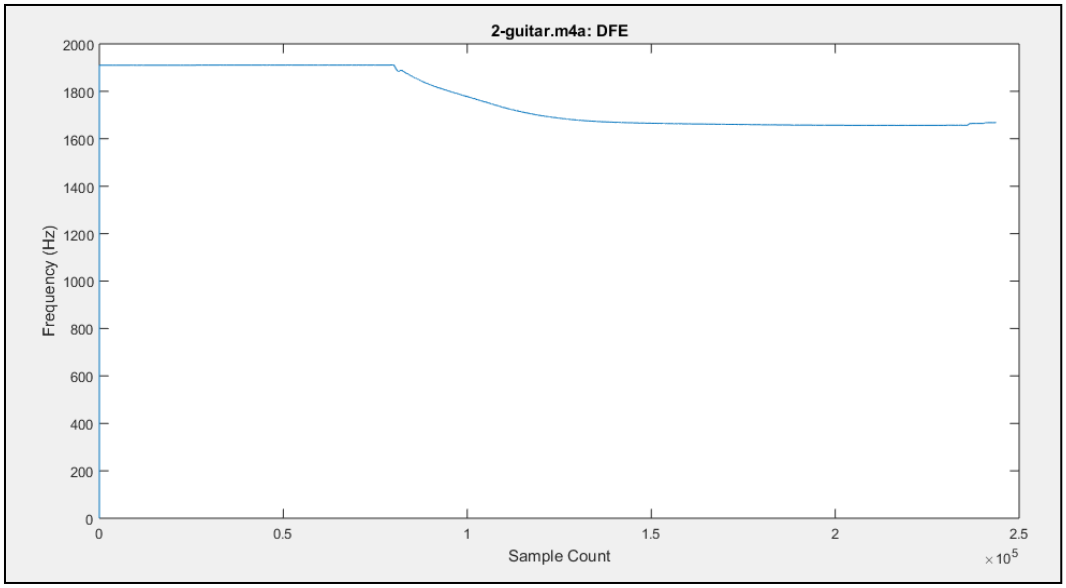


Figure 22. DFE frequency estimate of acoustic guitar.

The plot, below, of A-440Hz on the keyboard results in about 1kHz:

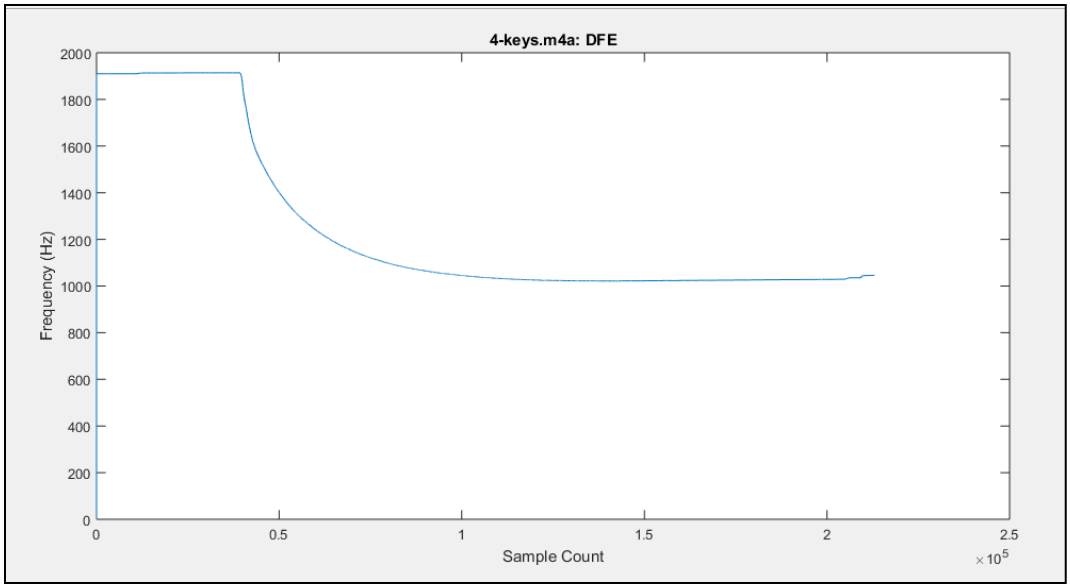


Figure 23. DFE frequency estimate of keyboard.

The five-tone melody played on either instrument is not accurate; the tracking of the notes is not even close. Note however that the frequency ranges appear consistent: the guitar results are around 1500Hz, and the keyboard results are around 1kHz:

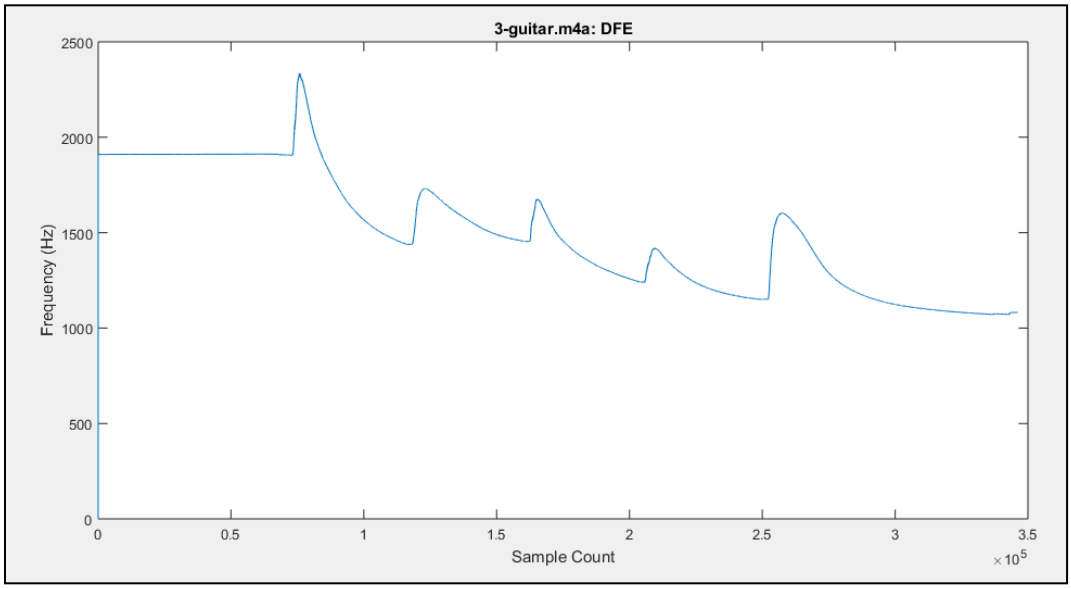


Figure 24. DFE frequency estimate of five-tone melody played on acoustic guitar.

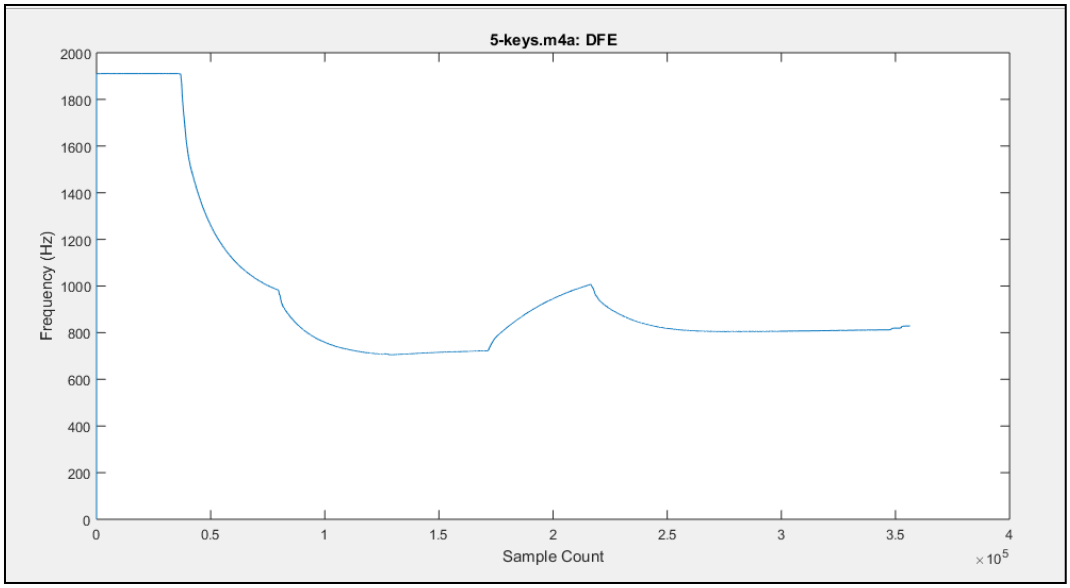


Figure 25. DFE frequency estimate of five-tone melody played on keyboard.

In summary, we see that the **dfe** algorithm fares better than the zero-crossing algorithm when processing pure sinusoidal signals. The instrument results, while incorrect, appear to show some consistency.

2.3. Adaptive Notch Filter

The algorithm defined in [2] is based on adaptively tuning a notch filter to the same frequency as the input signal's fundamental frequency. As the filter is intended to attenuate the signal, the filtered output signal itself is the error signal. The MSE expression is derived, differentiated with respect to the frequency θ to obtain the gradient expression, and the recursive LMS equation is obtained.

The complete algorithm as defined in [2] is designed to consider the signal's harmonic frequencies, as well as the fundamental; a scheme is presented in the paper to select an initial approximate frequency value, based on analyzing the MSE over the total time sequence. For this project, we have simplified the approach to consider only the fundamental, as we are interested in an online, i.e. real-time, design.

A brief summary of algorithm is as follows:

A well-known [3], [4] second-order adaptive IIR notch filter is defined by the transfer function

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1 - 2\cos(\theta)z^{-1} + z^{-2}}{1 - 2r\cos(\theta)z^{-1} + r^2 z^{-2}}$$

where *r* is a value less than but close to 1, which governs the width of the notch.

In the time domain, the transfer function can be expressed as the difference equation

$$y(n) - 2r\cos(\theta(n))y(n-1) + r^2 y(n-2) = x(n) - 2\cos(\theta(n))x(n-1) + x(n-2)$$

rearranging to get an expression for the output *y(n)*, we obtain

$$y(n) = x(n) - 2\cos(\theta(n))x(n-1) + x(n-2) + 2r\cos(\theta(n))y(n-1) - r^2 y(n-2)$$

The objective of the notch filter is to minimize the output power of the filtered signal; without providing the detailed steps of the derivation, the approach in [2] follows the LMS scheme: take the derivative of the MSE:

$$e^2(n) = y^2(n)$$

and equate to zero to obtain the LMS recursion expression

$$\theta(n+1) = \theta(n) - 2\mu y(n)\beta(n)$$

where the gradient function $\beta(n)$ is obtained as follows:

$$\beta(n) = \frac{\partial y(n)}{\partial \theta(n)}$$

$$= \frac{\partial}{\partial\theta(n)}x(n) - \frac{\partial}{\partial\theta(n)}2\cos(\theta(n))x(n-1) + \frac{\partial}{\partial\theta(n)}x(n-2) + \frac{\partial}{\partial\theta(n)}2r\cos(\theta(n))y(n-1) - \frac{\partial}{\partial\theta(n)}r^2 y(n-2)$$

$$= 0 - 2\sin(\theta(n))x(n-1) + 0 + \left(-2r\sin(\theta(n))y(n-1) + 2r\cos(\theta(n))\frac{\partial y(n-1)}{\partial\theta(n)}\right) - r^2\frac{\partial y(n-2)}{\partial\theta(n)}$$

$$= 2\sin(\theta(n))x(n-1) - 2r\sin(\theta(n))y(n-1) + 2r\cos(\theta(n))\beta(n-1) - r^2\beta(n-2)$$

The source code implementing the simplified version of the *Adaptive Notch Filter* (ANF) is the MATLAB **anf** function; it is slightly more complex than **zcross** and **dfe**. Similar to those functions, **anf** returns an array of the estimated frequencies based on each input sample:

```
function [F]=anf(x, Fs)
%
% ANF - Adaptive Notch Filter
%
% Simplified to assume just fundamental, no harmonics
%
% "Novel Adaptive IIR Filter for Frequency Estimation and Tracking",
% Li Tan, Jean Jiang
```

```
N=size(x);
mu=10^-3;       % LMS step size
r=0.85;
beta_prev1=0;   % assume last two gradients are 0
beta_prev2=0;
x_prev1=0;      % assume the last two filter inputs are 0
x_prev2=0;
y_prev1=0;      % assume last two filter outputs are 0
y_prev2=0;


F=zeros(1,length(x));

%
% Part 1: determine the initial value of theta
% JUST GUESS AT A NUMBER since there is only one frequency
theta = .01;

% Part 2: LMS algorithm to find frequency
%
% loop through the whole sample
for n=1:N

    % compute the new filter output
    y = x(n)-2*cos(theta)*x_prev1 + x_prev2 + 2*r*cos(theta)*y_prev1 - (r^2)*y_prev2;

    % compute the gradient function
    beta = 2*sin(theta)*x_prev1 - 2*r*sin(theta)*y_prev1 + 2*r*cos(theta)*beta_prev1 -
(r^2)*beta_prev2   ;

    % compute the new angle
    theta = theta - ( 2 * mu  * y * beta);

    % update previous x's, y's, betas
    x_prev2 = x_prev1;
    x_prev1 = x(n);

    y_prev2 = y_prev1;
    y_prev1 = y;

    beta_prev2 = beta_prev1;
    beta_prev1 = beta;
    F(n) = theta / ( 2 * pi ) * Fs;
end
    F(n)
```

Listing 5. Adaptive Notch FIlter (ANF) algorithm in MATLAB.

As with **dfe**, we have arbitrarily selected the step size **mu** and the initial frequency value **theta**.

2.3.1. Sinusoid Results

Results of **anf** estimating pure sinusoid signals are very good, as was observed with **dfe**: the plots are consistent, and the final results are accurate:

INPUT FILE	anf FINAL OUTPUT (Hz)
100Hz_44100Hz_16bit_05sec.wav	100
250Hz_44100Hz_16bit_05sec.wav	250
440Hz_44100Hz_16bit_05sec.wav	440.0004
1kHz_44100Hz_16bit_05sec.wav	1,000
10kHz_44100Hz_16bit_05sec.wav	10,000

Table 3. ANF results.

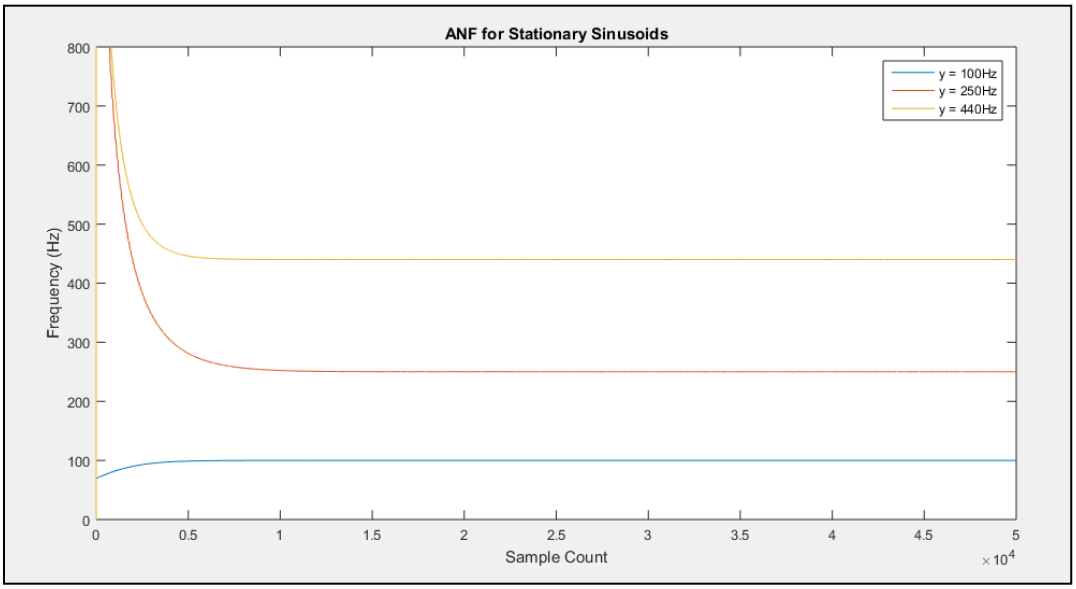


Figure 26. ANF plot for 100Hz, 250Hz, 440Hz.

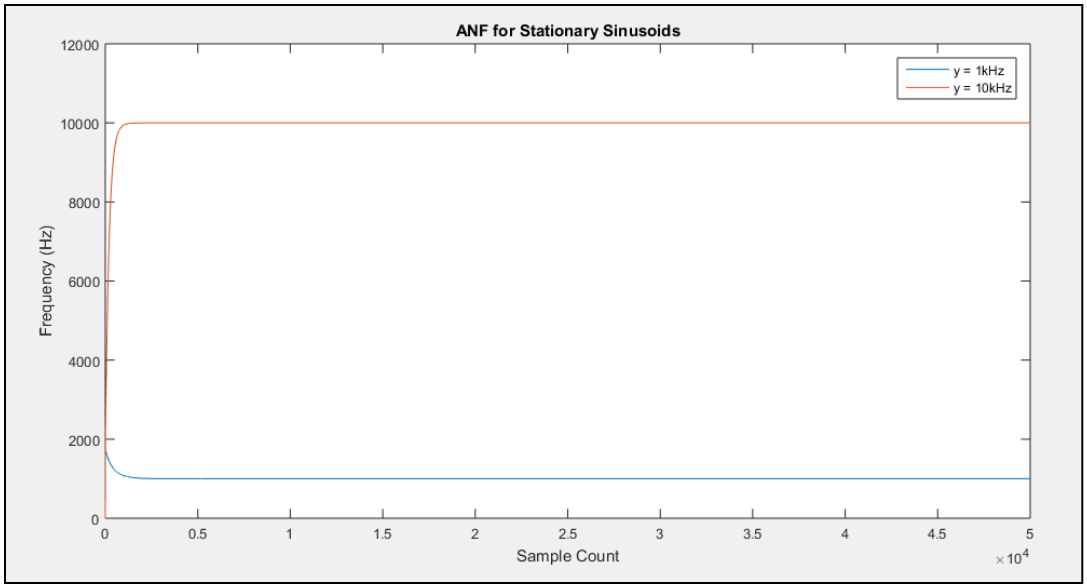


Figure 27. ANF plot for 1kHz, 10kHz.

The combined 440Hz and 1kHz signal results show a wider variance than **dfe**; note however that the mean appears to be approximately the same as the average 440 + 1kHz = 720Hz:

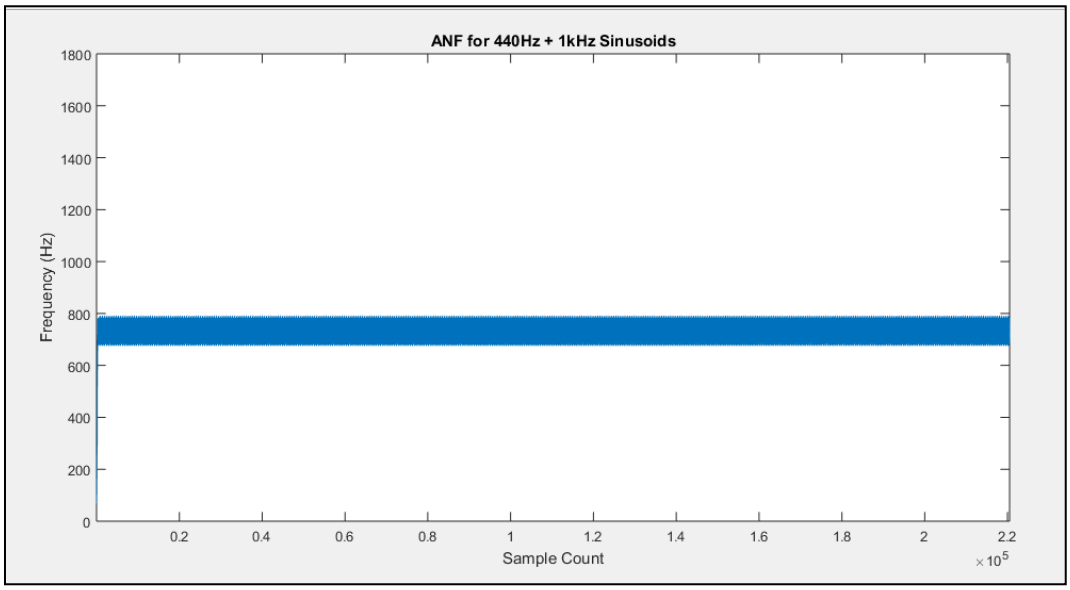


Figure 28. ANF estimated frequencies of combined sinusoids.

The five tone pure sinusoid melody is also accurate; the step size **mu** was arbitrarily selected to be 0.003, an order of magnitude smaller than that used with **dfe**, which may explain the faster convergence:

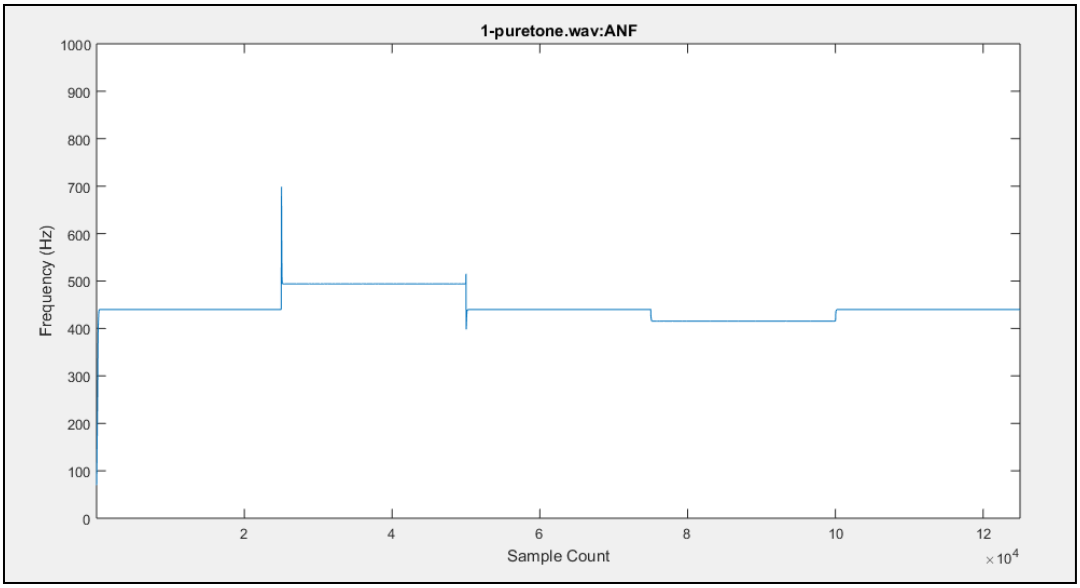


Figure 29. ANF estimate of five-tone melody.

2.3.2. Instrument Results

Similar to the **dfe** algorithm, we don't observe accurate results with **anf**, either with single-tone or melody tones::

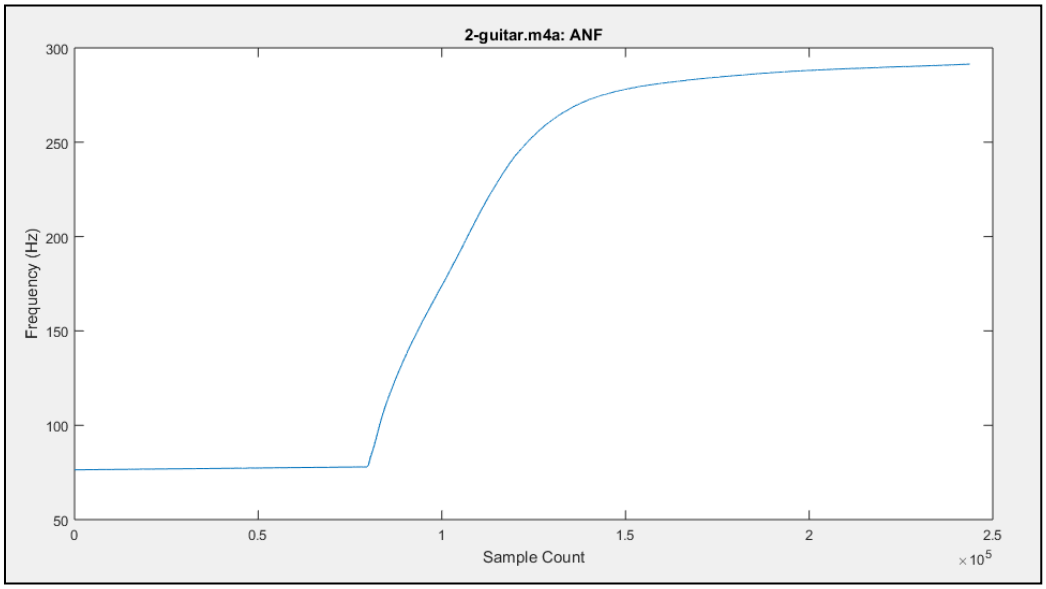


Figure 30. ANF frequency estimate of acoustic guitar.

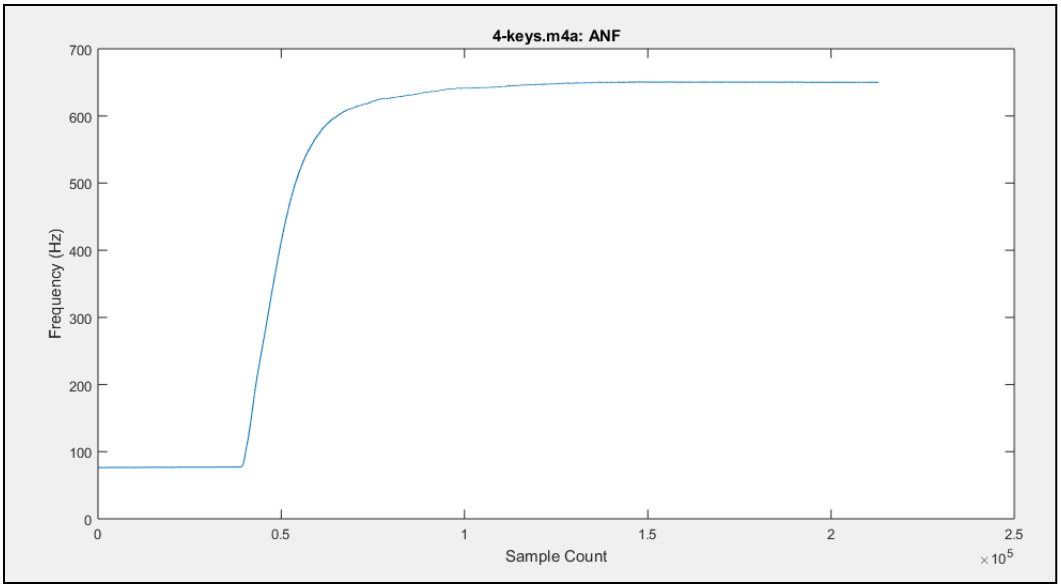


Figure 31. ANF frequency estimtae of keyboard.

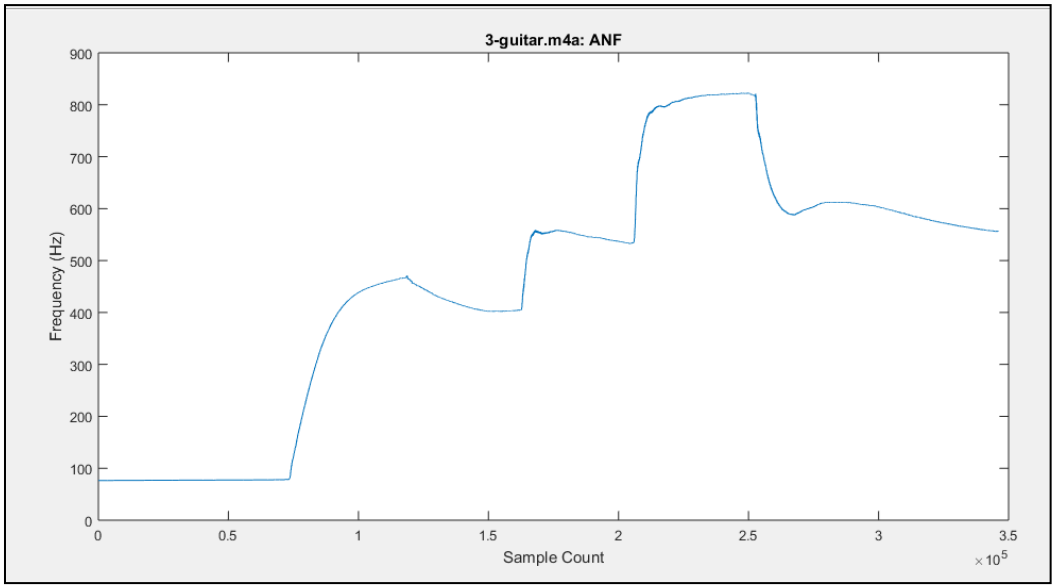


Figure 32. ANF frequency estimate of five-tone melody played on acoustic guitar.

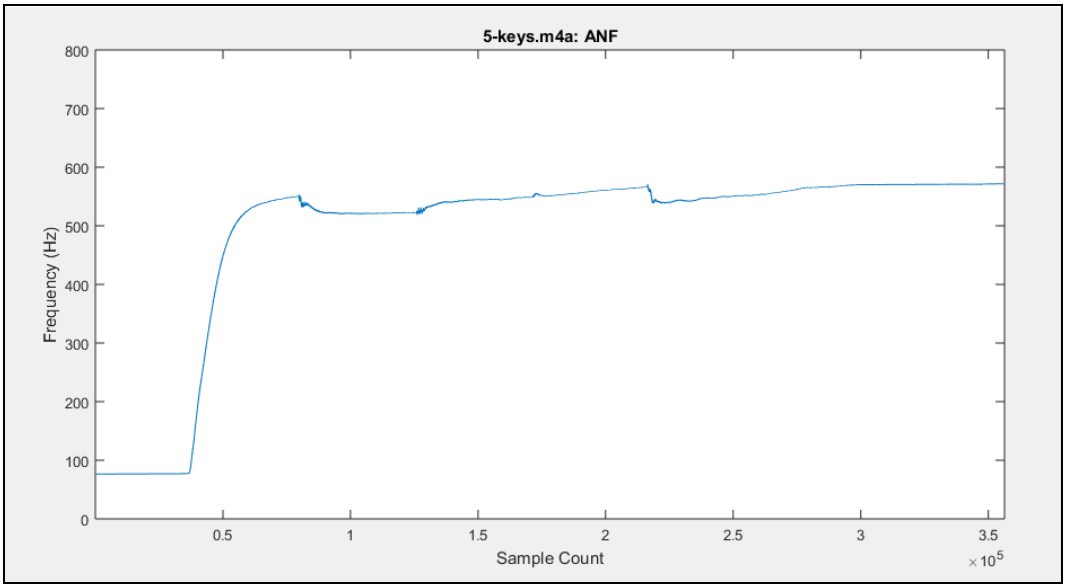


Figure 33. ANF frequency estimate of five-tone melody played on keyboard.

2.4. Summary

We have demonstrated that **dfe** and **anf** algorithms work very well with pure sinusoid signals, but fare poorly with the recorded instrument tones.

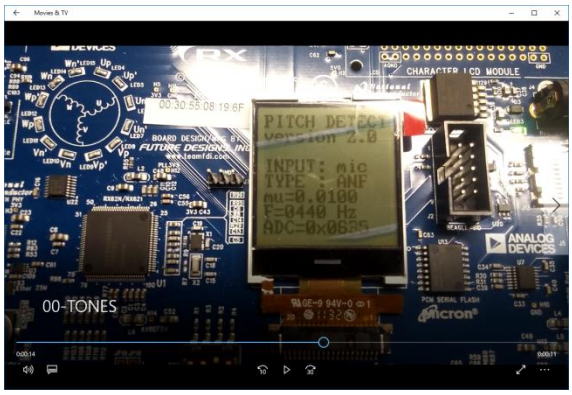
It's quite possible that the poor recording quality of the instruments may have played a significant role in the results; as well, these algorithms may not necessarily be the most ideal approaches when attempting to parse the fundamental from overtone-rich signals.

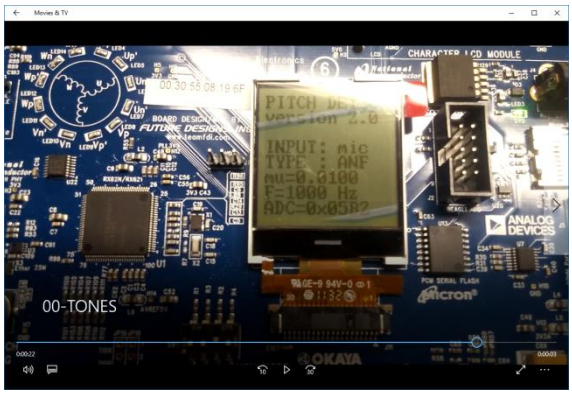
Nonetheless, the implementations for **dfe** and **anf** appear well suited for real-time implementation.

3. Embedded Platform Investigations

The most interesting results with the embedded platform were qualitative, and demonstrated in class; namely, that these algorithms can successfully track a pure sinusoid, while changing frequency.

Below are a couple of snapshots of the embedded platform identifying 440Hz and 1kHz pure sinusoid tones played by the computer:





Unlike the MATLAB processing of recorded instrument audio, the embedded platform showed mild success with single tones played by the guitar and cheap keyboard. Below are a couple of snapshots of the embedded platform successfully identifying a (slightly off-tune) A-440Hz:

The evaluation board implementation provided the option to switch between zero-crossing, DFE, and ANF algorithms, as well as to change the step size **mu** for DFE and ANF within certain ranges.

3.1. Zero-Crossing

Zero-crossing code is shown below; it was implemented within the timer interrupt routine:

```
.
.
.
if ( gFilterMode == MODE_FREQX )
  {
    /* estimate freq based on zero crossings */
    if ( ( gLastVal > 0x07FF ) && ( usADC_Result[5] < 0x07FF ) )
    {
```

```
      freq = FREQ / gCurCount;
      gCurCount = 0;
    }
    gLastVal = usADC_Result[5];
.
.
.
```

The DFE and ANF recursive equations were slightly more complex, and thus were segregated into their own routines.

3.2. Adaptive Linear Prediction (DFE)

Complete listing for the DFE source files is as follows; as with the ANF code, there was also a separate routine to change the step size **mu**, based on user button presses:

```
/*
 * FUNCTION DECLARATIONS
 */
void dfe_init( float _mu, float _angle );
void dfe_reset();
void dfe_update( float _mu );
float dfe_step( float _x );
```

```
#include "math.h"
#include "dfe.h"

/*
 * CONSTANTS
 */
#define PI  (3.1415927)

/*
 * GLOBAL VARIABLES
 */
static float gXprev1 = 0.0;    /* previous two samples */
static float gXprev2 = 0.0;

static float gMu;              /* step size            */
static float gAngle;           /* current estimate     */
static float gAngle_init;      /* initial angle        */

/*
 * DESCRIPTION:
 *   Initialize Direct Frequency Estimation algorithm
 */
void dfe_init( float _mu, float _angle )
{
  gMu = _mu;
  gAngle = _angle;
  gAngle_init = _angle;
  gXprev1 = 0.0;
  gXprev2 = 0.0;
```

```
}

void dfe_reset()
{
  gAngle = gAngle_init;
  gXprev1 = 0.0;
  gXprev2 = 0.0;
}

/*
 * DESCRIPTION:
 *   DFE iteration function
 */
float dfe_step( float _x )
{
  float s,e;

  /* compute the predicted signal from past two samples */
  float cosVal = cosf( gAngle );
  s = 2.0 * cosVal * gXprev1 - gXprev2;

  /* compute the error signal */
  e = _x - s;

  /* compute the new angle */
  /*
  gAngle = gAngle - ( gMu * e * ( ( ( _x + gXprev2 ) * cosf( gAngle ) ) + gXprev1 ) );
 */
  float gAngle_delta = ( gMu * e * ( ( ( _x + gXprev2 ) * cosVal ) + gXprev1 ) );
  gAngle = gAngle - gAngle_delta;

  /* update state variables */
  gXprev2 = gXprev1;
  gXprev1 = _x;

  return( gAngle );
}

/*
 * DESCRIPTION:
 *   Update step size
 */
void dfe_update( float _mu )
{
  gMu = _mu;
}
```

3.3. Adaptive Notch Filter

Complete listing for the ANF source files is as follows:

```
/*
 * FUNCTION DECLARATIONS
 */
```

```
void anf_init( float _mu, float _angle );
void anf_reset();
void anf_update( float _mu );
float anf_step( float _x );



#include "math.h"
#include "anf.h"

/*
 * CONSTANTS
 */
#define PI  (3.1415927)

/*
 * GLOBAL VARIABLES
 */
static float gBetaPrev1 = 0.0;   /* previous two gradients */
static float gBetaPrev2 = 0.0;

static float gXprev1 = 0.0;
static float gXprev2 = 0.0;

static float gYprev1 = 0.0;
static float gYprev2 = 0.0;

static float gMu;              /* step size            */
static float gAngle;           /* current estimate     */
static float gAngle_init;      /* initial angle        */

static float gR = 0.85;

/*
 * DESCRIPTION:
 *   Initialize Direct Frequency Estimation algorithm
 */
void anf_init( float _mu, float _angle )
{
  gMu = _mu;
  gAngle = _angle;
  gAngle_init = _angle;
  dfe_reset();
}

void anf_reset()
{
  gAngle = gAngle_init;
  gXprev1 = 0.0;
  gXprev2 = 0.0;
  gYprev1 = 0.0;
  gYprev2 = 0.0;
  gBetaPrev1 = 0.0;
  gBetaPrev2 = 0.0;
}

/*
 * DESCRIPTION:
 *   DFE iteration function
```

```
 */
float anf_step( float _x )
{
  float beta, y;

  /* compute the new filter output */
  float cosVal = cosf( gAngle );
  float sinVal = sinf( gAngle );

  y = _x - 2 * cosVal * gXprev1 + gXprev2 + 2 * gR * cosVal * gYprev1 - ( gR * gR ) * gYprev2;

  /* compute the gradient function */
  beta = 2 * sinVal * gXprev1 - 2 * gR * sinVal * gYprev1 + 2 * gR * cosVal * gBetaPrev1 - gR
* gR * gBetaPrev2;

  /* compute the new angle */
  gAngle = gAngle - ( 2 * gMu * y * beta );

  /* update the previous values */
  gXprev2 = gXprev1;
  gXprev1 = _x;

  gYprev2 = gYprev1;
  gYprev1 = y;

  gBetaPrev2 = gBetaPrev1;
  gBetaPrev1 = beta;
  gXprev1 = _x;

  return( gAngle );
}

/*
 * DESCRIPTION:
 *   Update step size
 */
void anf_update( float _mu )
{
  gMu = _mu;
}
```

4. Final Thoughts

Both adaptive filter algorithms were demonstrably successful on MATLAB with pure sinusoids; and, amazingly, there was mild success on the embedded platform, with real instruments.

4.1. MATLAB Recorded Instrument vs Live Instrument Results

In the recorded video examples of demonstrating instrument frequency identification, there were good results in identifying the keyboard tones, and mild success with the guitar (some notes were successfully identified, and some were not).

Recall that the MATLAB results for the recorded instrument tones were not successful. We might theorize some reasons:

- quality of the microphones. The embedded platform's microphone may have been of lower quality, and thus acted as a filter for some of the overtones captured by the desktop microphone used for the MATLAB recordings; this filtering of overtones may have caused the embedded platform to be presented with a signal with a more significant fundamental
- recording methodology. Perhaps the desktop recording of the instruments was very close to the instrument, and thus better captured all the signal spectrum; while the live execution with the embedded platform, being performed at a greater distance, attenuated those overtones.

4.2. Evolving To A Real Guitar Tuner

While the implementation of commercial electronic tuner offerings is considered proprietary corporate IP and thus is not readily available for examination, it's possible that some adaptive concepts might be employed. It is certain that more complex audio processing applications use adaptive filters[4].

Our embedded application was a simple demonstration of frequency identification; to evolve to a real guitar tuner, it should be configured with a table of frequencies associated with the guitar's six strings (or even better, a chromatic scale), and provide some qualitative indication of how sharp or flat the currently detected frequency is, to the nearest note in the scale.

5. References

1. So, H.C. and Ching, P.C.: "Adaptive Algorithm for Direct Frequency Estimation" (IEE Proc. Radar Sonar Navig., Vol. 151, No. 6, December 2004)
2. Tan, Li and Jiang, Jean: "Novel Adaptive IIR Filter for Frequency Estimation and Tracking" (IEEE Signal Processing Magazine, November 2009)
3. Li, G.: "A Stable and Efficient Adaptive Notch Filter for Direct Frequency Estimation" (IEEE Transactions on Signal Processing, Vol. 45, No. 8, August 1997)
4. Zhou, J. and Li, G.: "Plain Gradient Based Direct Frequency Estimation Using Second-Order Constrained Adaptive IIR Notch Filter" (Electronics Letters, Vol 40, No. 5, March 2004)

6. Data Files

A MATLAB example of reading a datafile, and producing the frequency estimate array for a given algorithm, is as follows:

```
% read the input file samples into array 'y'
% Fs is the sampling frequency of the input file data
[y,Fs]=audioread('5-keys.m4a');
```

[4] The open-source audio processing software *Audacity* makes use of adaptive filters for a number of features.

```
% process the input samples 'y' using the DFE algorithm
% frequency estimate at each sample returned in array dfe_y
dfe_y = dfe(y,Fs);

% plot frequency estimates
plot(dfe_y);
xlabel( 'Sample Count' );
ylabel('Frequency (Hz)' );
title('5-keys.m4a: DFE');
```

6.1. Sinusoidal tones

The following files contain pure sinusoidal unbiased signals at the indicated frequency and sampling rate. The 5-second duration at the 44.1kHz sampling rate results in 220,500 samples in each file.

```
1kHz_44100Hz_16bit_05sec.wav
10kHz_44100Hz_16bit_05sec.wav
100Hz_44100Hz_16bit_05sec.wav
250Hz_44100Hz_16bit_05sec.wav
440Hz_44100Hz_16bit_05sec.wav
```

The following file combines 440Hz and 1kHz sinusoids.

```
440Hz_1kHz_combined.wav
```

6.2. Instrument Generated Single Tone

A440 played on a Gibson Epiphone EJ200 acoustic guitar, recorded with a low-quality microphone into the built-in mic input of a desktop computer, using the Windows 10 Voice Recorder application. Background hissing noise is significant.

```
2-guitar.m4a
```

A440 played on a children's toy keyboard instrument, using the same recording setup as above. Background noise is again significant.

```
4-keys.m4a
```

6.3. Five-tone melody

Each of the following files contain the five tones played in succession:

A, B, A, G#, A

With A=440Hz and assuming equal temperament tuning, the frequencies are

A : 440Hz

B : 440 * $2^{2/12}$ = 493.88Hz

G# : 440 / $2^{1/12}$ = 415.30Hz

These tones were generated in MATLAB with the above frequencies, and represent pure sinusoidal signals.

`1-puretone.wav`

These tones were generated with the acoustic guitar configuration described earlier.

`3-guitar.m4a`

These tones were generated with the toy keyboard configuration described earlier.

`5-keys.m4a`

7. Source Files

7.1. MATLAB

`zcross.m`	Simple zero-crossing algorithm
`zcross2.m`	Modified zero-crossing algorithm to allow for sample value 0
`dfe.m`	Adaptive Linear Prediction via Direct Frequency Estimation (DFE) [1]
`anf.m`	Adaptive Notch Filter (ANF) [2]

7.2. Embedded Platform

The Renesas RX62N evaluation board included a lot of sample code; fortunately, demonstration of continuous ADC operation was included (although the input was originally the onboard potentiometer).

Digging through the documentation illustrated the I/O register configuration that allowed using the microphone as the input; Googling for further information revealed that an on-board microphone pre-amp component needed to be enabled, as well.

There was much boilerplate and other reused source code components (hardware setup, LCD display, etc); the new code written or modified for this project is as follows:

`PD_main.c`	The original **main_adc_repeat.c** demo code, modified for the Pitch Detection (PD) demonstration
`PD.c`	The original **adc_repeat.c** demo code, modified for the Pitch Detection (PD) demonstration. This contained the main timer interrupt handler code, executing at 10kHz, that read the ADC and executed either the zero-crossing, ANF, or DFE

	algorithms. This code also incorporated zero-crossing.
`anf.c` `anf.h`	The Adaptive Notch Filter code
`dfe.c` `dfe.h`	The Adaptive Linear Prediction aka Direct Frequency Estimation code

8. **Proof of Sinusoidal Predictive Property**

This proof is not given in [1] so it is provided below:

Given a discrete sinusoidal signal

$$s_n = \alpha \cos(\omega n + \phi)$$

its previous two samples are given by

$$s_{n-1} = \alpha \cos(\omega(n-1) + \phi)$$

$$s_{n-2} = \alpha \cos(\omega(n-2) + \phi)$$

We can express s_n as

$$= \alpha \cos(\omega n + \phi) + \alpha\cos(\omega n - 2\omega + \phi) - \alpha \cos(\omega n - 2\omega + \phi)$$

cosine is an even function: $\cos(a) = \cos(-a)$ so we can obtain

$$= \alpha \cos(\omega n + \phi) + \alpha\cos(-(\omega n - 2\omega + \phi)) - \alpha \cos(\omega n - 2\omega + \phi)$$

$$= \alpha \cos(\omega + \omega n - \omega + \phi) + \alpha\cos(\omega - \omega n + \omega - \phi) - \alpha \cos(\omega n - 2\omega + \phi)$$

$$= \alpha \cos(\omega + (\omega(n-1) + \phi)) + \alpha\cos(\omega - (\omega(n-1) + \phi)) - \alpha \cos(\omega n - 2\omega + \phi)$$

$$= 2\alpha \frac{1}{2}\left[\cos(\omega + (\omega(n-1) + \phi)) + \cos(\omega - (\omega(n-1) + \phi))\right] - \alpha \cos(\omega n - 2\omega + \phi)$$

recall the trigonometric identity:

$$\cos(a)\cos(b) = \frac{1}{2}(\cos(a-b) + \cos(a+b))$$

so we can write

$$= 2\alpha\left[\cos(\omega)\cos(\omega(n-1) + \phi)\right] - \alpha \cos(\omega n - 2\omega + \phi)$$

$$= 2\cos(\omega)\alpha\cos(\omega(n-1)+\phi)) - \alpha\cos(\omega(n-2)+\phi)$$

$$= 2\cos(\omega)s_{n-1} - s_{n-2}$$