HARMONIC EQUALIZER

12/11/2006

Group 5 Final Report, 18-551

Eric Boulanger	ehrick@cmu.edu
Kush Jawahar	kjawahar@andrew.cmu.edu
Aditya Raj Kanodia	akanodia@andrew.cmu.edu

Harmonic equalizer

GROUP 5, FALL '06

THE PROBLEM

With the advent of digital recording and editing, the recording industry has sacrificed the rich sound of the past for the speed, ease and in-expense digital has to offer. Many 'audiophiles' directly criticize sampling as the reason for this lack of what many were used to with analog tape, however with the sampling rates now available (96khz) even a trained ear can not distinguish the difference between analog and digital. Furthermore, digital recording has the lowest noise floor, largest dynamic range, and least THD (total harmonic distortion) of any other recording medium. So what is it that people complain about; what is it that recording engineers and producers strive for? If it's not the sampling, it must be something else that's being done differently.

THEORETICAL SOLUTION:

The fact of the matter is when recording digitally, and manipulating the recording with digital filters ("plug-ins") the signal if anything is too perfect. Recordings made in the 70's and 80's got most of their signature 'sound' from what electrical engineers would consider a nightmare: over-saturating tape (purposefully distorting the signal), over-driving tube amplifiers, and etc. It is obvious to us that the human ear does not want to hear the perfect reproduction of a rock band crammed in a small studio room. The magic of an album comes from manipulating analog devices to add desired 'noise'.

If it's un-called for artifacts that the ear likes, then as engineers we must identify what our ears want to hear so that we can deliver. The most sought after sound, especially with recording vocals, is the brilliance and clarity in running the signal through high quality tube amplifiers. This brilliance is as a result of how a vacuum tube distorts; take for example a solid-state op-amp. When an op-amp distorts, the output 'rails' to match exactly its supply voltage. If forced a sinusoidal input of greater amplitude than its supply voltage, the output will greatly resemble a square wave. If there is any "ramping up" (non ideality in the square wave) it would be as a result of the op-amp's slew rate. If this same process was done to a vacuum tube amplifier, the output would still be greatly sinusoidal before clipping. This can be understood as the 'slew rate' of a vacuum tube being far less to that of its solid-state brethren. What the ear picks up on with a distorting tube is since it distorts sinusoidal-ly, all harmonics are present, as opposed to the solid-state square wave type distortion which is only odd harmonics. Odd harmonics are not very musical since they do not correspond to notes in the musical scale which are pleasing (i.e. octaves, 4ths, 5ths).

OUR PROJECT AND SOLUTION:

Why be forced to add tube sound to add upper harmonics when harmonics are already there? We've re-thought and improved upon one of the most powerful devices a recording engineer has at their disposal: the parametric equalizer. Our never-before-attempted version will, with the user's control, be capable of filtering and consequently highlight the upper harmonics a tube normally would be used for.

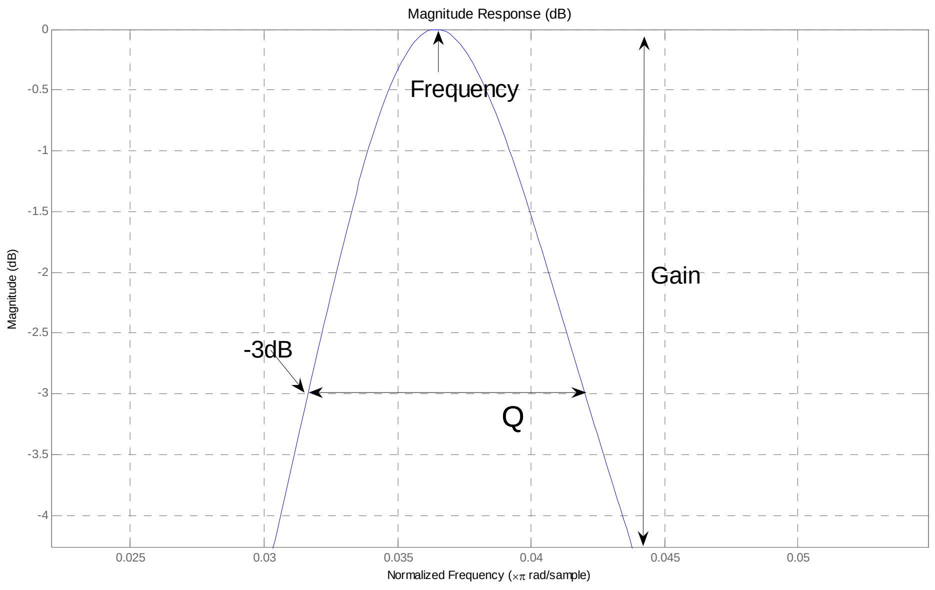
Our device operates very similarly to a standard parametric equalizer which controls gain, frequency, and Q for four bands: low, low-mid, high-mid, and high. We however add to each band controls for gain, Q, and shape which filter the upper harmonics. To describe better in detail, listed below is each control's functionality:

(items in **bold** can be found on a regular parametric eq)

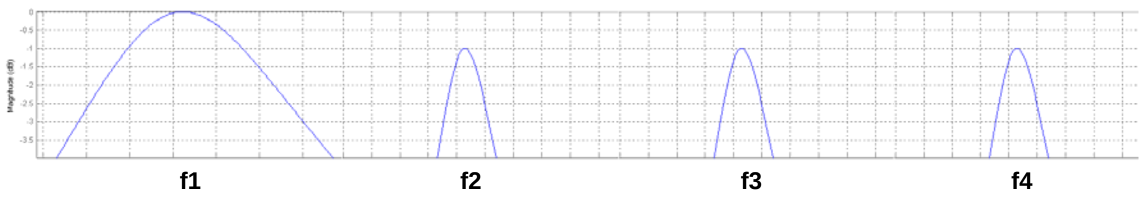
- **Frequency**: defines the fundamental center frequency
- **Q**: controls the q of <u>only</u> the fundamental filter
- **Gain**: controls the gain of <u>only</u> the fundamental filter
- Harmonic Gain: controls the gain of all the harmonic filters
- Harmonic Q: controls the Q of all the harmonic filters
- Shape: alters the gain of individual harmonic filters to "shape" their response

*Note: there is no ‘harmonic frequency’ control; this is because the center frequency of each harmonic filter is determined by the fundamental frequency. Each harmonic is a multiple of the fundamental, so for example the 2nd harmonic of the fundamental frequency = 2*fundamental frequency.

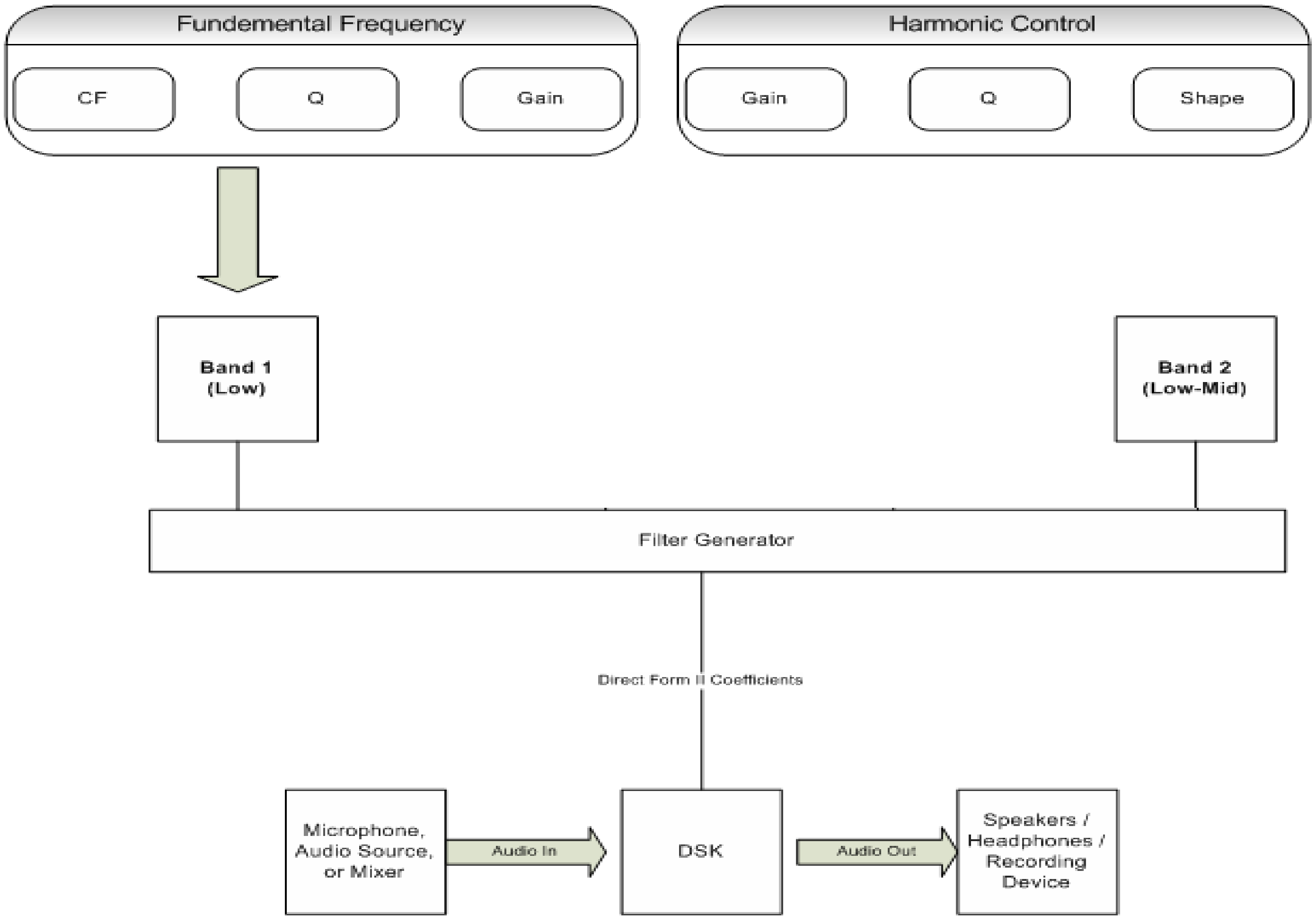
The three basic functions of a bandpass filter controllable by an equalizer:



*Demonstration of what we can control with our device. In this example we can see how the fundamental filter (f1) can have a different gain and q from the harmonic filters (f2, f3, f4). Filters are placed according to the harmonic series: f2 = f1*2, f3=f1*3, f4=f1*4. The frequency “f1” is controlled by the user.*



In order to be fully ‘dynamic’, meaning the user can enter in any combination of settings with each band’s 6 controls, our device accepts commands from the user via a GUI, enters them in to our ‘filter generator’ (calculates the coefficients and specs of each filter) whose output is fed into direct form II filter implementation which filters the incoming audio.



PRIOR 18-551 GROUP PROJECTS

Our project is unique in many ways. Not only does it (as explained above) re-think the operation of a standard EQ. No commercial plug ins incorporate as many filters as our device in one implementation either. We did use the experience of our 18-551 predecessors to decide on what type of pitch tracking algorithm to use.

- Group 9 Spring 2003
- Group 9 Spring 2000
- Group 9 Spring 2002

The above groups were all groups that we studied their pitch tracking algorithms. I guess Group 9 in Spring is a popular assignment for pitch trackers! Below in our pitch tracking segment, we use Group 9, Spring 2003's analysis of auto correlation as a comparison to our method.

OPERATION:

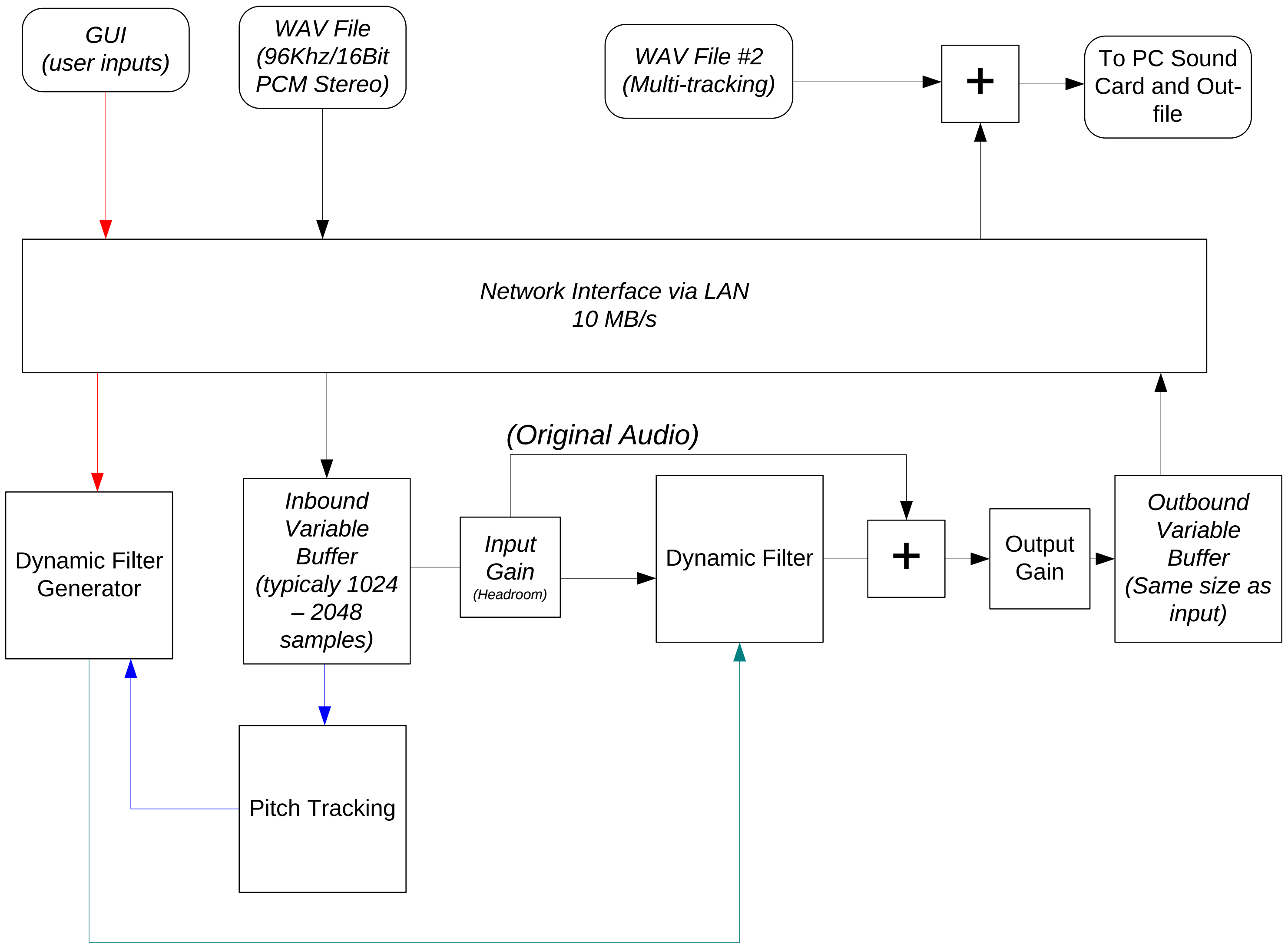
OUR DEVICE CAN OPERATE IN ONE OF TWO WAYS:

1) Digital Plug-in: when audio originates from the PC, is then sent to the DSK for processing, and then sent back to PC for output, and or recording. This scenario would be typically used in the home / project studio situation where (because of its cost efficiency) dsp is at the center of all operations.

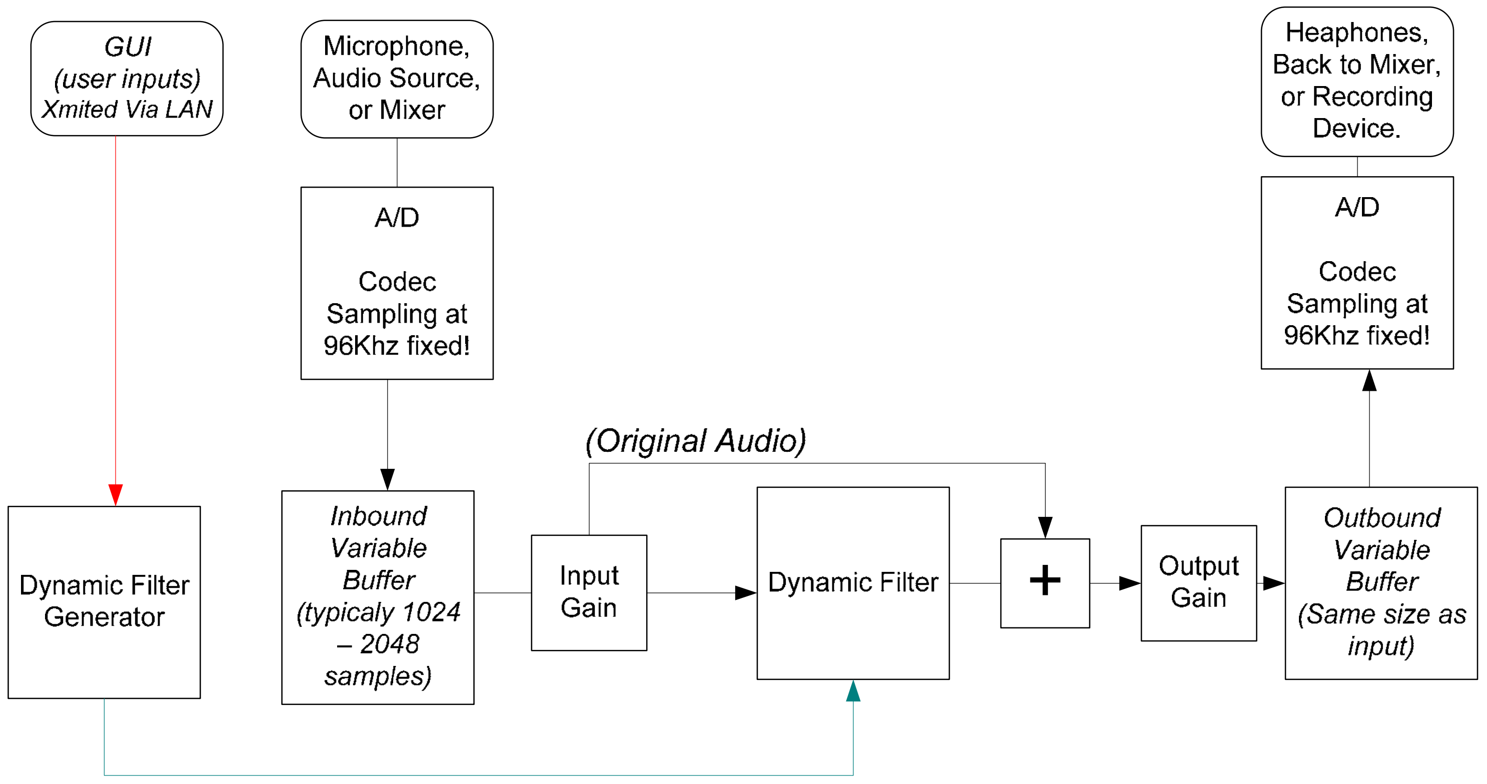
2)Analog Plug-in: when an audio signal is sent to the line in of the DSK, then processed, and sent back out the line out. In this case, all A/D is fixed at 96khz. This scenario would be typically used in the professional recording studio situation where our device would most likely be used only to slightly accentuate an audio track. Routing in such cases usually is completely done in the analog domain with extremely high end audio consoles.

THE NEXT TWO FIGURES ARE THE DATAFLOW DIAGRAMS OF WHAT IS EXPLAINED ABOVE.

Digital Plug-in Dataflow:


GUI (user inputs)
WAV File (96Khz/16Bit PCM Stereo)
WAV File #2 (Multi-tracking)
+
To PC Sound Card and Out-file
Network Interface via LAN 10 MB/s
(Original Audio)
Dynamic Filter Generator
Inbound Variable Buffer (typicaly 1024 – 2048 samples)
Input Gain (Headroom)
Dynamic Filter
+
Output Gain
Outbound Variable Buffer (Same size as input)
Pitch Tracking

ANALOG PLUG-IN DATAFLOW:


GUI
(user inputs)
Xmited Via LAN
Microphone, Audio Source, or Mixer
A/D
Codec Sampling at 96Khz fixed!
Heaphones, Back to Mixer, or Recording Device.
A/D
Codec Sampling at 96Khz fixed!
(Original Audio)
Dynamic Filter Generator
Inbound Variable Buffer (typicaly 1024 – 2048 samples)
Input Gain
Dynamic Filter
+
Output Gain
Outbound Variable Buffer (Same size as input)

BACKGROUND RESEARCH ON FILTERS:

As in any engineering design problem, it is generally not possible to give a precise answer to what is best. No single type of filter and no single design method is best for all circumstances. If we neglect phase considerations, it is generally true that a given magnitude-response specification can be met most efficiently with an IIR filter. However, we must make sure that our system's delay is not excessive to the point that it may become audible.

The chief properties of FIR and IIR filters are outlined below.

IIR Filters

Properties of IIR Filter:

1. *Better magnitude response*
2. *Fewer coefficients*
3. *Less storage is required for storing variables. Fewer taps and thus less hardware than FIR.*
4. *A lower delay*
5. *It is closer to analog models*
6. *A variety of frequency-selective filters can be designed using closed form design formulas.*

A variety of frequency-selective filters can be designed using closed form design formulas. Once the problem has been specified in terms appropriate for a given approximation method(e.g. Butterworth, Chebyshev etc), then the order of the filter that will meet the specifications can be computed and the coefficients of the discrete time filter can be obtained by straight forward substitution into a set of design equations.

These methods are limited to frequency-selective filters and they permit only the magnitude response to be specified. If other magnitude shapes are desired, an algorithmic procedure is required.

Classical IIR Filters

Butterworth

The Butterworth filter provides the best approximation to the ideal bandpass filter response at analog frequencies. Passband and stopband response is maximally flat.

Chebyshev Type I

The Chebyshev Type I filter minimizes the absolute difference between the ideal and actual frequency response over the entire passband by incorporating equal ripple in the passband. Stopband response is maximally flat. The transition from passband to stopband is more rapid than for the Butterworth filter.

Chebyshev Type II

The Chebyshev Type II filter minimizes the absolute difference between the ideal and actual frequency response over the entire stopband by incorporating an equal amount of ripple in the stopband. Passband response is maximally flat. The stopband does not approach zero as quickly as the Type I filter (and does not approach zero at all for even-valued filter order n). The absence of ripple in the passband, however, is often an important advantage.

Elliptic Filter

Also known as Cauer filters, Elliptic filters are equiripple in both the passband and stopband. They generally meet filter requirements with the lowest order of any supported filter type. For a given filter order, passband ripple, and stopband ripple, elliptic filters minimize transition width. This filter has the fastest roll off but very nonlinear phase.

Comb Filter

In signal processing, a comb filter adds a slightly delayed version of a signal to itself, causing phase cancellations. The frequency response of a comb filter consists of a series of regularly-spaced spikes, so that it looks like a comb. In discrete-time systems, the filter implements the following formula: $y[n] = ax[n] + bx[n - \tau] + cy[n - \tau]$ where τ is a constant delay. The comb filter can also be implemented in the continuous-time domain. The frequency response is given by the following:

$$H(\omega) = \frac{a + be^{-i\omega\tau}}{1 - ce^{-i\omega\tau}}$$

FIR Filters

Properties of FIR filters:

1. *A linear phase*
2. *It is unconditionally stable (since no non-zero poles)*
3. *It is insensitive to quantization effects*
4. *No limit cycle issues (i.e., the filter oscillating)*
5. *It is easy to design*
6. *Closed form design equations do not exist for FIR filters.*

The simplest method of FIR filter design is called the window method. Some of the common window functions are:

1. *Rectangular*
2. *Bartlett*
3. *Henning*
4. *Hamming*
5. *Blackman*
6. *Kaiser*

Although the window method is easy to apply, some iteration is necessary to meet a prescribed specification. The Parks-McClellan algorithm leads to lower order filters than the window method does. The design problem for FIR filters is more under control than the IIR design problem. This is because of the optimality theorem for FIR filters which is applicable in a wide variety of applications.

How to choose which filter to use:

Our obvious number one criterion is how the filter sounds, and how close to real time we can get this filter to perform. Hence translated into engineering language, this relates to passband ripple in amplitude, excessive phase shift, and the filter’s order.

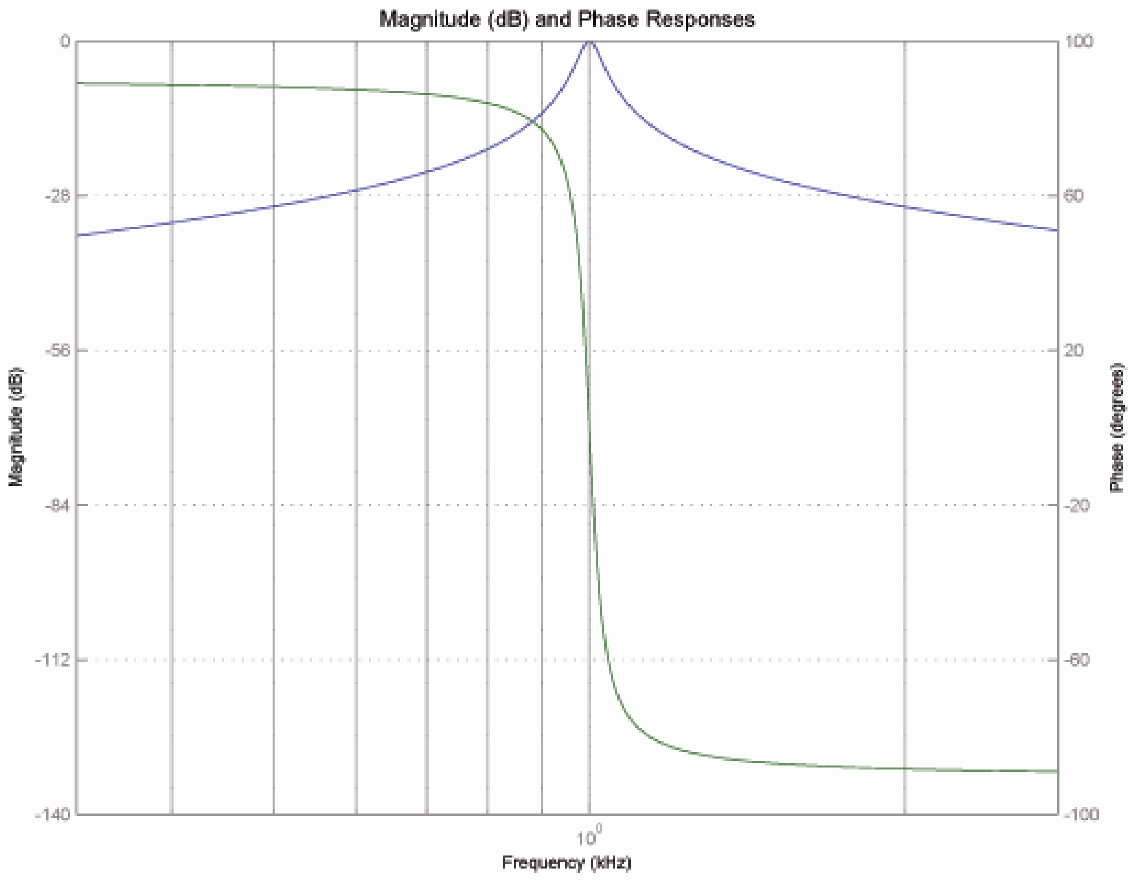
So first IIR vs. FIR:

	IIR	FIR
Phase	Difficult to control, no particular techniques available	Linear phase always possible.
Stability	Can be unstable, Can have limit cycles	Always Stable, No limit cycles.
Order	Less	More
History	Derived from analog filters / circuitry	None, it's the rookie.

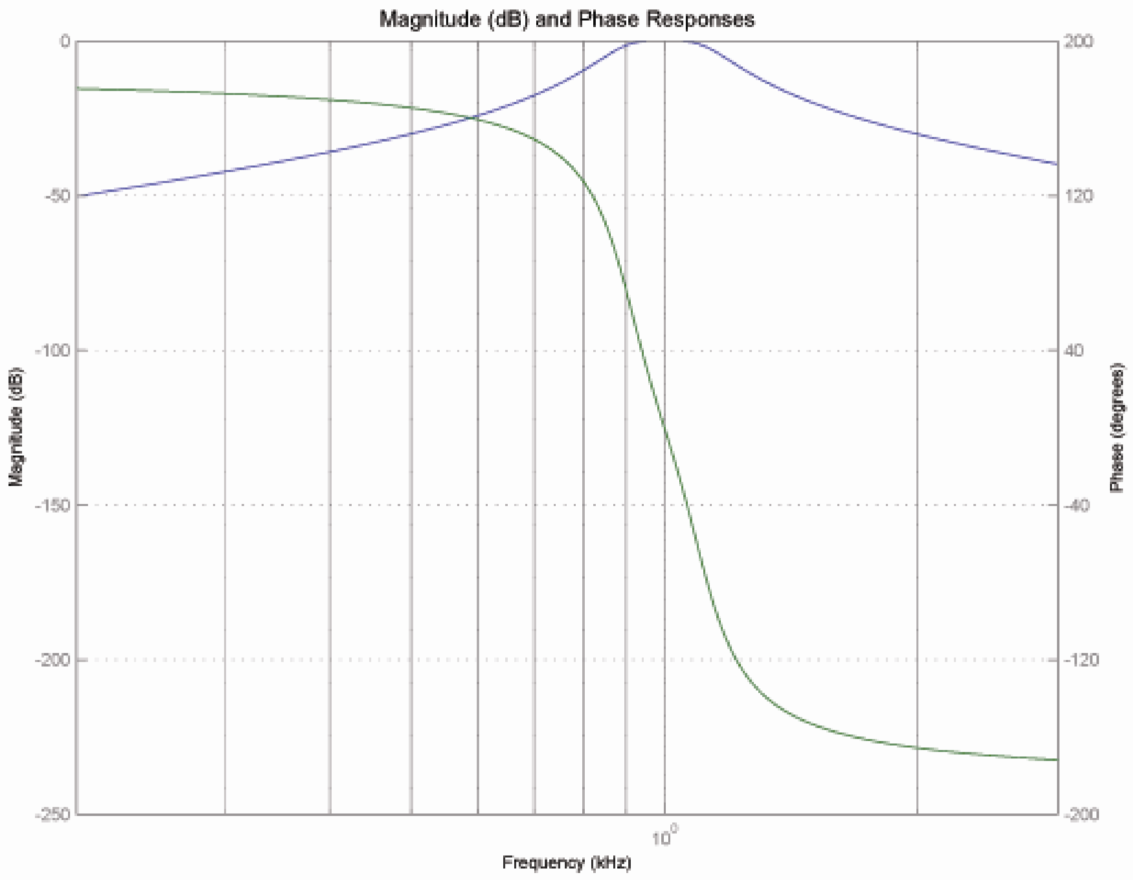
Starting with phase, right away it may seem that a FIR filter should win out over an IIR since linear phase seems like a great way of controlling the group delay of our filter. We've adhered to a maximum of 3ms delay, as to make our device as perceivably real-time as possible. This means our group delay must not be any more than a total of 3ms. We also chose not to use a filter which shifts any more than 180 degrees in the pass band to prevent the possibility of when summing the processed sound with the original to achieve a wet/dry mix, certain frequencies that are meant to be gain-ed would actually be attenuated because of their close to one period phase shift.

So to compare, we'll look at (next three figures) a single bandpass filter centered at 1000hz, a Q of 7, and at unity gain. For suspense we won't say what type of filter we used until later, though we in this case are comparing our filter, a Butterworth IIR, and an equiripple FIR. To do so we will find their Bode plot, (mag in blue and phase in green on top of each other) and then plot / find their maximum group delay.

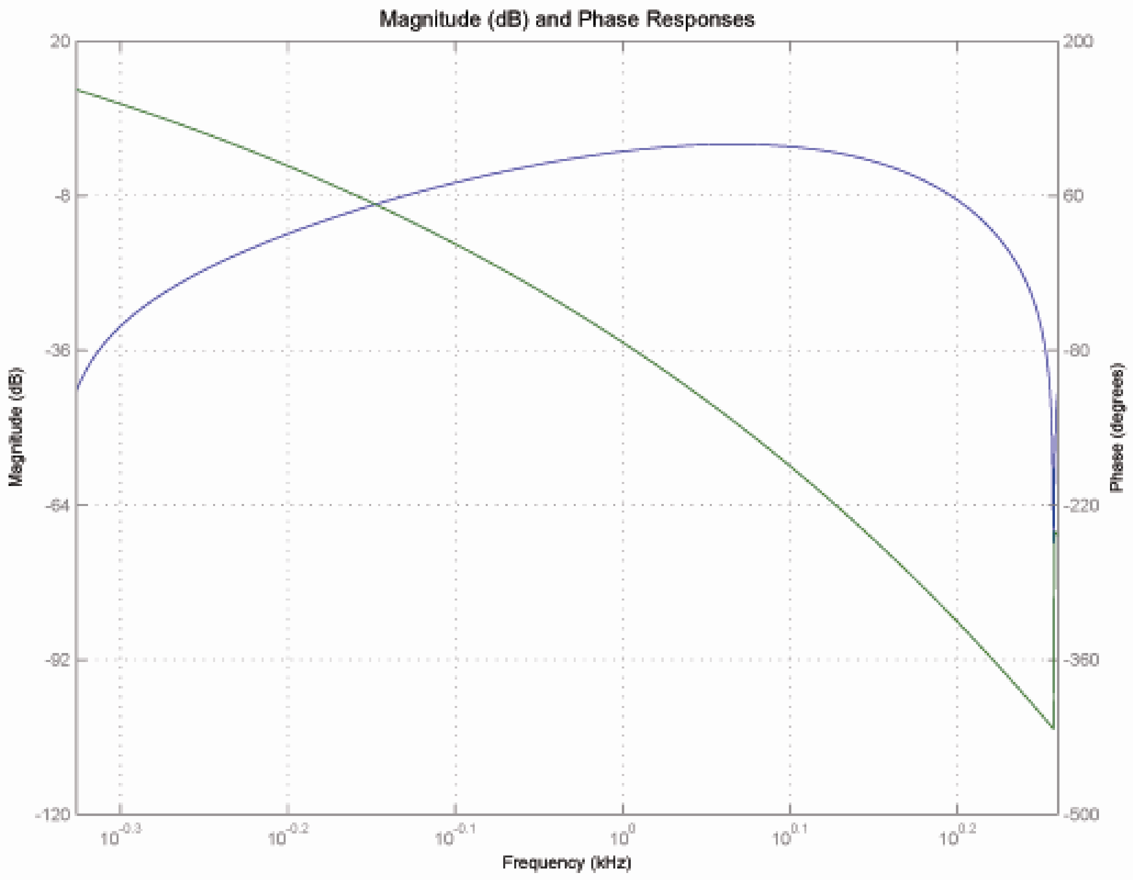
Our Filter's Frequency Response



Butterworth Filter Frequency Response

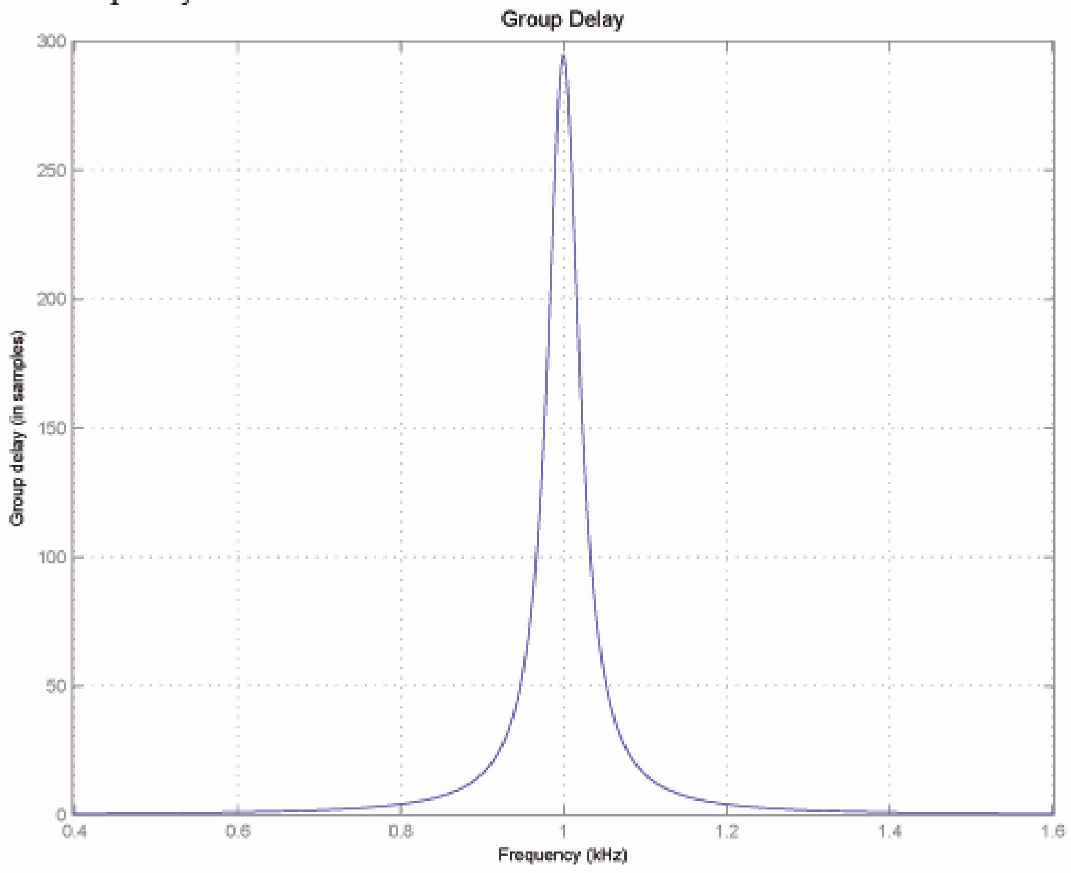


FIR (Equiripple) Filter Frequency Response

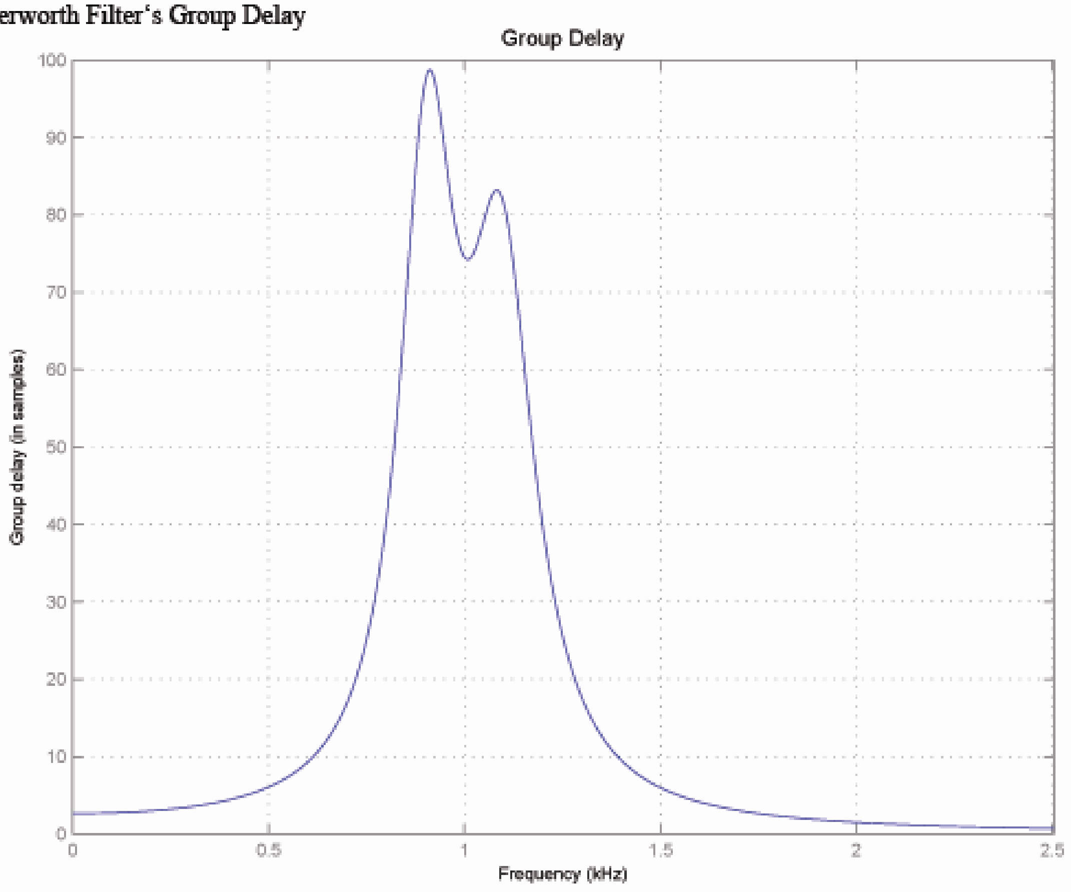


The next two figures are the group delay plots for the two IIR filter examples above. These plots are group delay measured in samples vs. frequency. We did not plot the group delay of the FIR filter since it is trivial. The FIR filter's phase is perfectly linear, hence the group delay (the derivative of the phase vs. frequency) is a constant. This filter's delay is 53 samples.

Our Filter's Group Delay



Butterworth Filter's Group Delay



The group delay of a filter is a measure of the average delay of the filter as a function of frequency. It is the negative first derivative of the phase response of the filter. If the complex frequency response of a filter is $H(e^{j\omega})$, then the group delay is

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega}$$

where ω is frequency and θ is the phase angle of $H(e^{j\omega})$. Average group delay is the average of this result only across the passband.

So now to calculate maximum group delay in seconds, pick the highest value, and:

Seconds = max grpdelay value in samples / sampling rate

Our filter: 300 samples / 44100 samp per sec = 6.8ms,

average = 21.6 samples / 44100 samp. per sec = **0.04 ms.**

Butterworth: 100 samples / 44100 samp. per sec = 2.3 ms ,

average = 40 samples / 44100 samp. per sec = **0.09 ms.**

FIR: The phase is linear, so the group delay is trivial since it is just the negative slope of that phase, hence our result is 53 samples in this example.

Max and Average: 53 samples / 44100 = **1.2 ms.**

In order to decide which filter type to use, we chose the filter with the least average group delay. Since our ears are not sensitive to small changes in frequency, we average the group delay results. Therefore it is clear why we chose our filter, being it has the smallest average group delay, even though it has the largest max delay. The max delay in our filter's case occurred at the frequency of interest, however to be perfectly honest, after feeding audio through each of these examples, we could not tell which was which by listening. Perhaps since everything is under 3ms we cannot perceive the difference since their magnitudes are to the same specification.

Aside from delay, filter order is another very important deciding factor. In the 3 examples above, the FIR filter had an order of 106! This is extremely high compared to the two IIRs, where the Butterworth had an order of 4, and ours had an order of 2. Not only was our filter extremely low order (for one bandpass), when repeated to form the harmonics that could potentially have different gains and Qs, the order still remained at 2 for our operating ranges. This was extremely computationally efficient as we never had to worry about the chance of having different order filters. This was a clear choice to us, though to further demonstrate the true power of our filter vs. an FIR, we plot the frequency response of the FIR filter and our filter as they would have to be used in our device; numerous parallel filters centered at harmonic values. The next two figures are how our filter vs. the FIR filter would operate with our device at 1000hz, unity gain, and a Q of 7 for the fundamental and all harmonics. Our filter actually is picture with a gain of 1dB instead of 0dB(unity). This is only because our code would not allow an input of 0dB since it is intended only to add harmonics; if there was a gain of 0dB, there is no point in using the device since it should do nothing.

THE DECISION:

The decision was clear cut… our filter had better average group delay, and far better filter order. Also when implemented in our device, it was a lot more uniform, and did not develop massive ripples in higher frequencies as the FIR did.

SO WHAT KIND OF FILTER DID WE USE? (drum roll):

Second-order IIR peaking (resonator) digital filter

Why was this so suspenseful / interesting? Referring to our table comparing IIR and FIR filters above, we ended up choosing a filter that is the exact mathematical representation of ‘historical analog filters’. Can we say RLC circuit? Perhaps it’s just coincidence, but it sure is strange that we go through all of this trouble to find out that to get a sound our ear liked…. a sound our ear liked back in the 70’s with all that *analog* gear… we end up with the closest thing digital can represent.

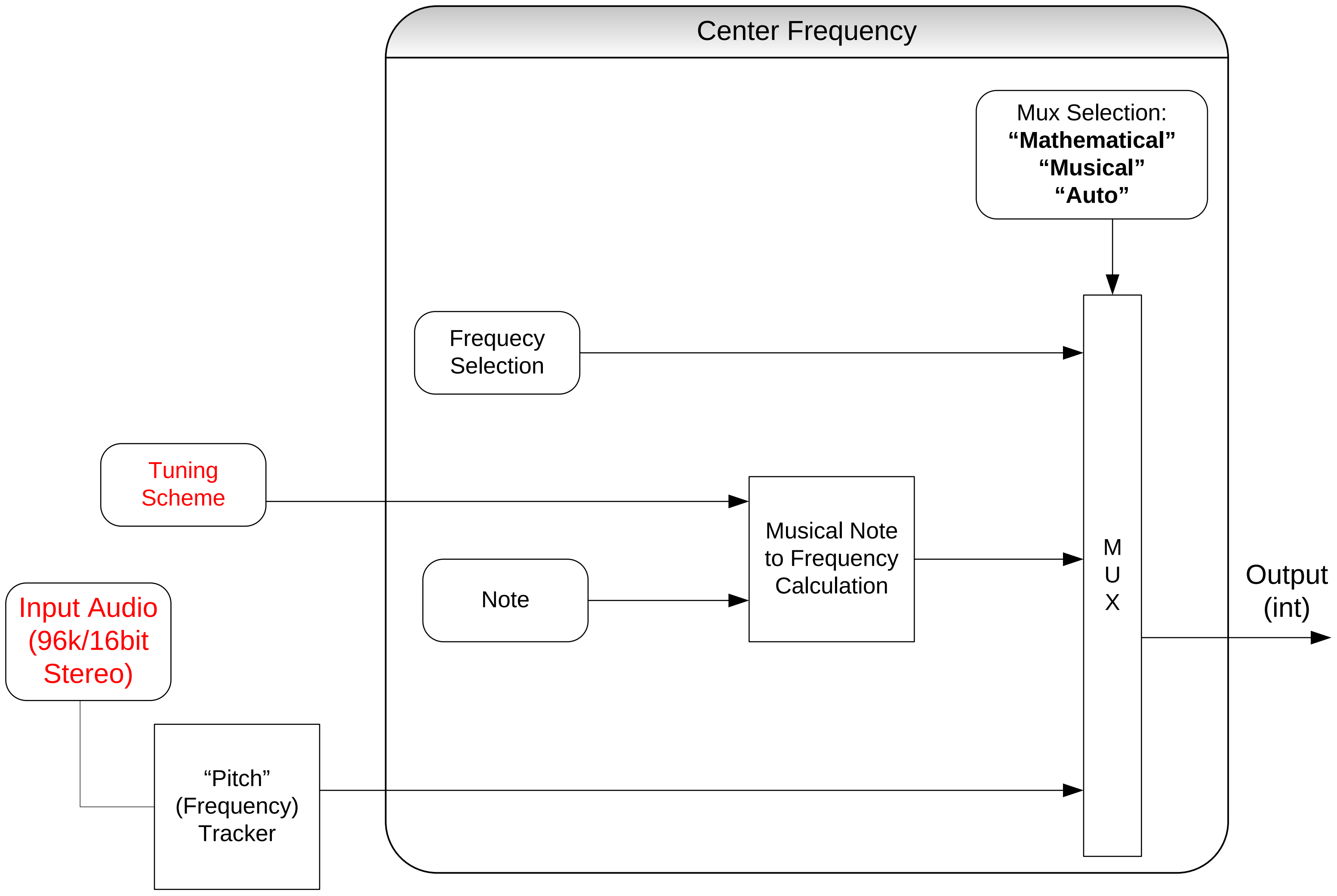
OUR BASIC FILTER FUNCTION:

(IN MATLAB)

```
function [b,a] = filterCode(Fs,CenterFreq,Q,Gain)
%(Sampling Rate , Center Frequency , Q , Gain)

%relationships
Wo = CenterFreq/(Fs/2);
BW   = CenterFreq/Q;
BW = BW/(Fs/2);

%normalize
BW = BW*pi;
Wo = Wo*pi;

Gb   = 10^(-Gain/20);
beta = (Gb/sqrt(1-Gb.^2))*tan(BW/2);
gainc = 1/(1+beta);

%coefficient calculation (Raj Output)
b2  = (1-gainc)*[1 0 -1];
a  = [1 -2*gainc*cos(Wo) (2*gainc-1)];

b=b2*(10^(Gain/20));
```

EXTRA FEATURES:

Our mission in designing this device was to improve upon the use, operation, and sound of equalizers. We therefore found that we did not have to be locked into the traditional methods of equalizer operation, where fundamental frequencies are chosen as frequencies in hz. Musically '1320 hz' makes little sense. So why do we hold recording engineers who are usually of mainly musical backgrounds to our engineering standards of hz? Why not relate what the user is doing to the music; so instead of 1320hz, we say "E". Better yet, why not let the DSK choose for you! The next figure demonstrates how the user has the choice between "Mathematical" (traditional method of picking hz), "Musical" (our method where the user selects a musical note), or "Auto" (where the DSK does the thinking).



At first it may seem trivial selecting a musical note for a frequency since a musical note surely must represent some frequency right? Wrong! The harmonic series governs the way a string vibrates; harmonics are integer multiples of the fundamental frequency. Knowing this, Pythagoras *back in the day* figured out a bunch of perfect ratios which calculate any musical interval is in frequency. This is how to calculate a musical note's frequency in hz. For example: the standard reference frequency is "A 440hz". If I wanted to know the frequency of the E above that A, I would know musically, that is an interval of a fifth. I then would look up on Pythagoras' chart and see that an interval of a fifth is a 3/2 ratio. Hence E then must be 440*3/2 = 660hz.

Here's the problem: Pythagoras sure was right, however he wasn't much of a musician. His method will work fine for anything that's 'in one key'… uses only notes associated with the key of A (the reference standard). If a song changes keys ("modulates") the octaves and subsequent harmonics of this new key will not line up, simply put… the instrument will sound out of tune and horrible. So now fast forward a few years to J.S. Bach's day. One of Bach's most famous compositions were the Prelude and Fugues of "The Well Tempered Clavier". Well tempered is the literal translation from German of what we now call "equal temperament tuning". Sometimes also called 12-tet tuning, 12-tet is the solution to Pythagoras' problem. Since there are 12 notes in one octave, 12-tet evenly divides the frequencies between an octave amongst the 12 notes. This means that the notes aren't perfectly in tune, nor are their harmonics, however one could consider 12-tet an 'average value' of musical notes. This is what allows music to be played on any instrument in any key. . The formula for 12-tet is as follows: 440*2^(x/12).

440 is the same reference value as before, and x is the number of notes away from the reference you are calculating. So just as before, let's find E but now in 12-tet. E is 7 notes above A, so: 440*2^(7/12) = 659.3 hz. In this case there is only a small change however as you go up in frequency or in the musical scale, the intervals get wider and wider.

We need to use this information since our project walks a fine line between 'perfect tuning' and 12-tet. That's why we decided to have the option of choosing between

Pythagoras' or 12-tet tuning. So now if a song is tuned in 12-tet (which it probably is) when the user selects "E" the device will center its filters properly.

How about the lazy recording engineers? Well our device can think for them. If the input signal is monophonic (one instrument playing one note at a time) we've implemented a pitch tracking algorithm which can figure out the fundamental frequency, and set the filters to align to whatever fundamental frequency is being played.

MULTITRACKING

To better demonstrate the possibilities of our device in a professional recording environment, we added two-track capability to the digital plug-in scenario so that one file is processed with our device (sent through the DSK) and one is played at the same time through the sound card. This demonstrates how a single track in a mix can be accentuated without having to make the track's volume unnecessarily loud.

PITCH TRACKING

Pitch Tracking:

The basic idea behind the idea we used for pitch tracking was to use Instantaneous Frequency Distributions (IFD). If we take the IFD of a given sample, then the first maximum is considered to be our fundamental frequency. However, computational considerations require us to estimate the IFD of a given signal rather than compute it directly. In an attempt to do so, we used Charpentier's method, as referenced in our reference section. Our method of computing the IFD is to compute two Short Time Fourier Transforms (STFT). Using these STFTs, we are then able to estimate the magnitude of the derivative of the phase or the IFD of the given signal. The STFTs are related to each other in a manner that makes this process equivalent to the evaluation of phase shifts due to an infinitesimal displacement of the time window.

The first step in implementing this algorithm requires us to compute the STFT of a signal which is represented by:

$$X_k(n) = \sum_{m=0}^{N-1} w^{-km}\, x(n+m)\, h(m)$$

However, calculating the STFTs using this method would require the use of two FFTS which is computational inefficient. So we can instead one FFT of the original signal. Then apply the Henning window using the following formula:

$$X_k(n) = Y_k - (Y_{k-1} + Y_{k+1})/2$$

We can calculate derive the second STFT using the original FFT using:

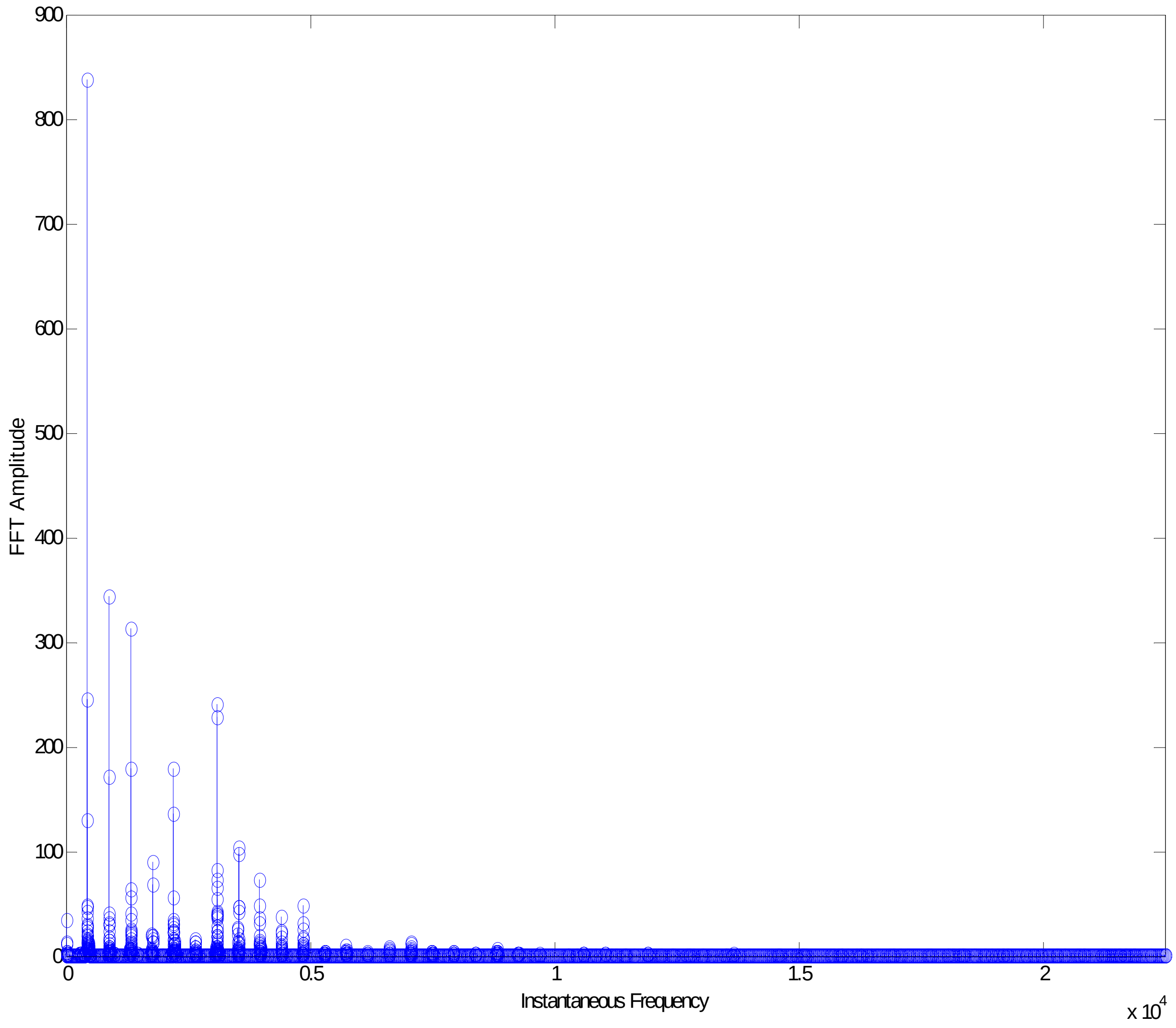
$$X_k(n-1) = w^{-k}\left[Y_k - (wY_{k-1} + w^{-1}Y_{k+1})/2\right]$$

Now that we have our two STFTs, we can calculate the change in phase between two time shifted signals using:

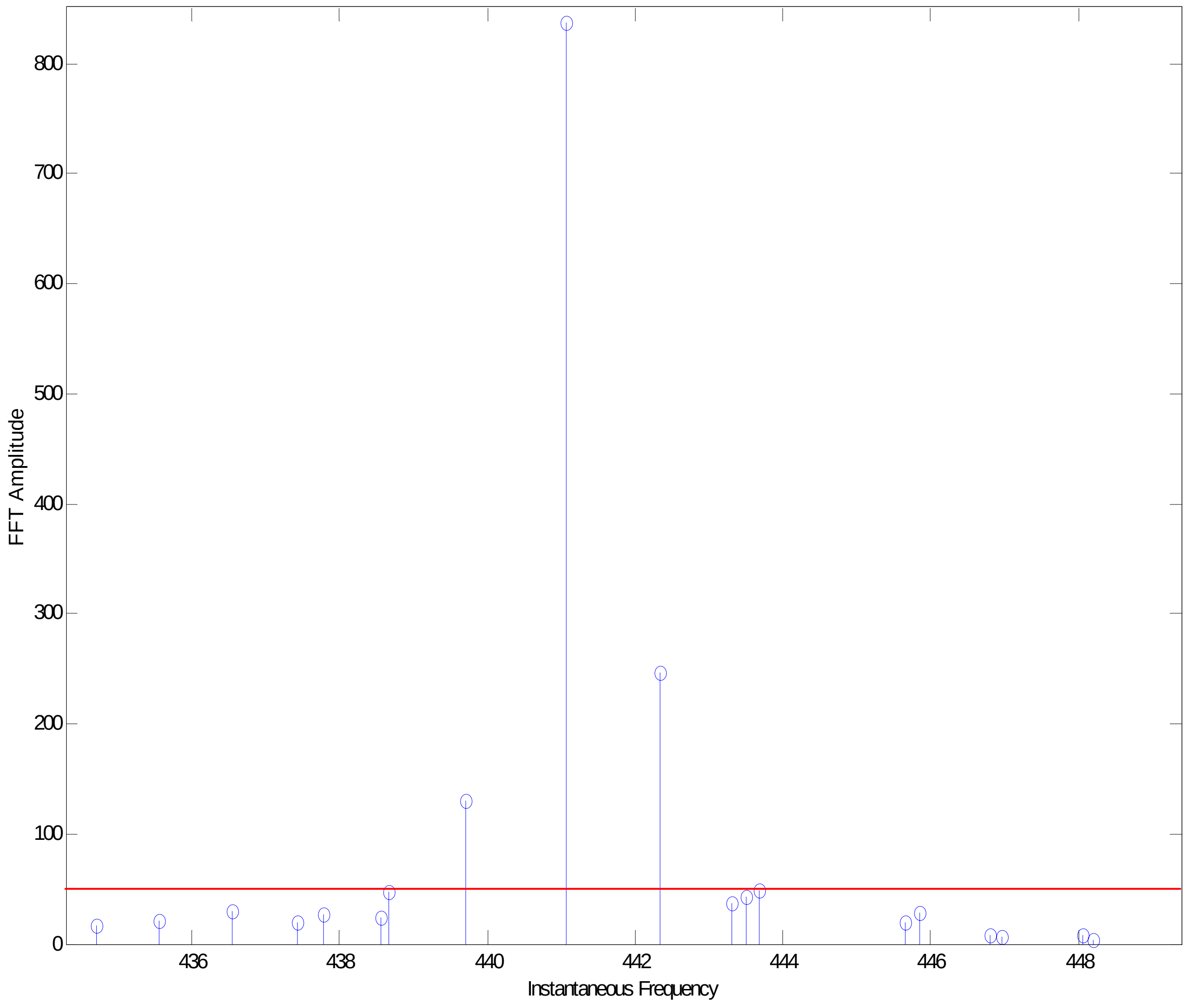
$$\Delta\varphi_k = 2\pi(k/N) - \mathrm{Arg}\left[\frac{2Y_k - wY_{k-1} - w^{-1}Y_{k+1}}{2Y_k - Y_{k-1} - Y_{k+1}}\right]$$

Now that we have our approximation of the IFD, we traverse through this set of frequencies until we hit the first set of three adjacent frequencies whose corresponding magnitude or X_k is above a threshold. The middle of these three frequencies is our fundamental frequency.

A quick example of the STFT vs. IFD of a violin playing an A 440hz.



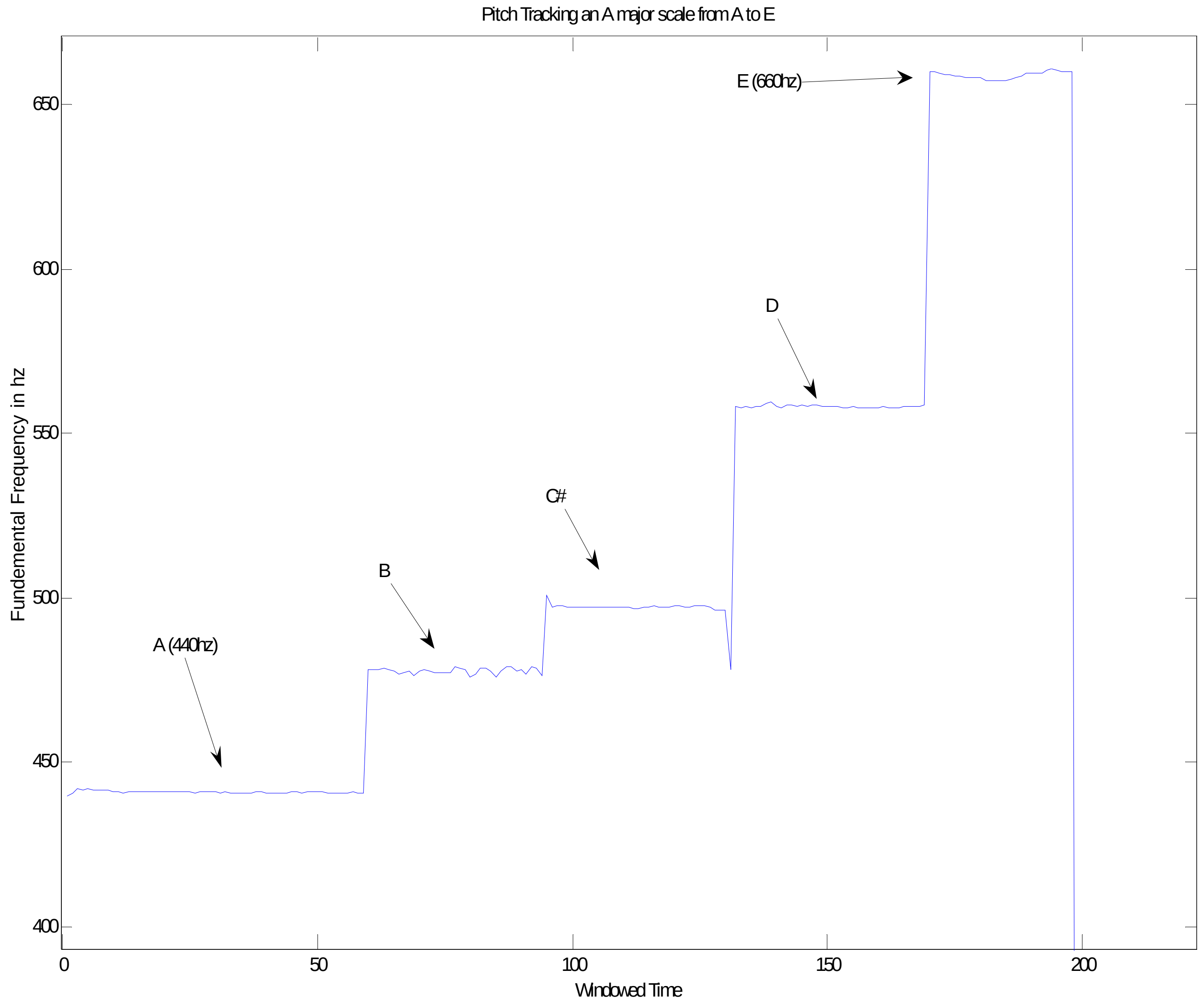
Same plot, but now zoomed in on the fundamental frequency, which is the first spike in the plot above. Note the red line is our threshold, which is how we determine when the fundamental energy is present.



PITCH TRACKING METHOD DECISION

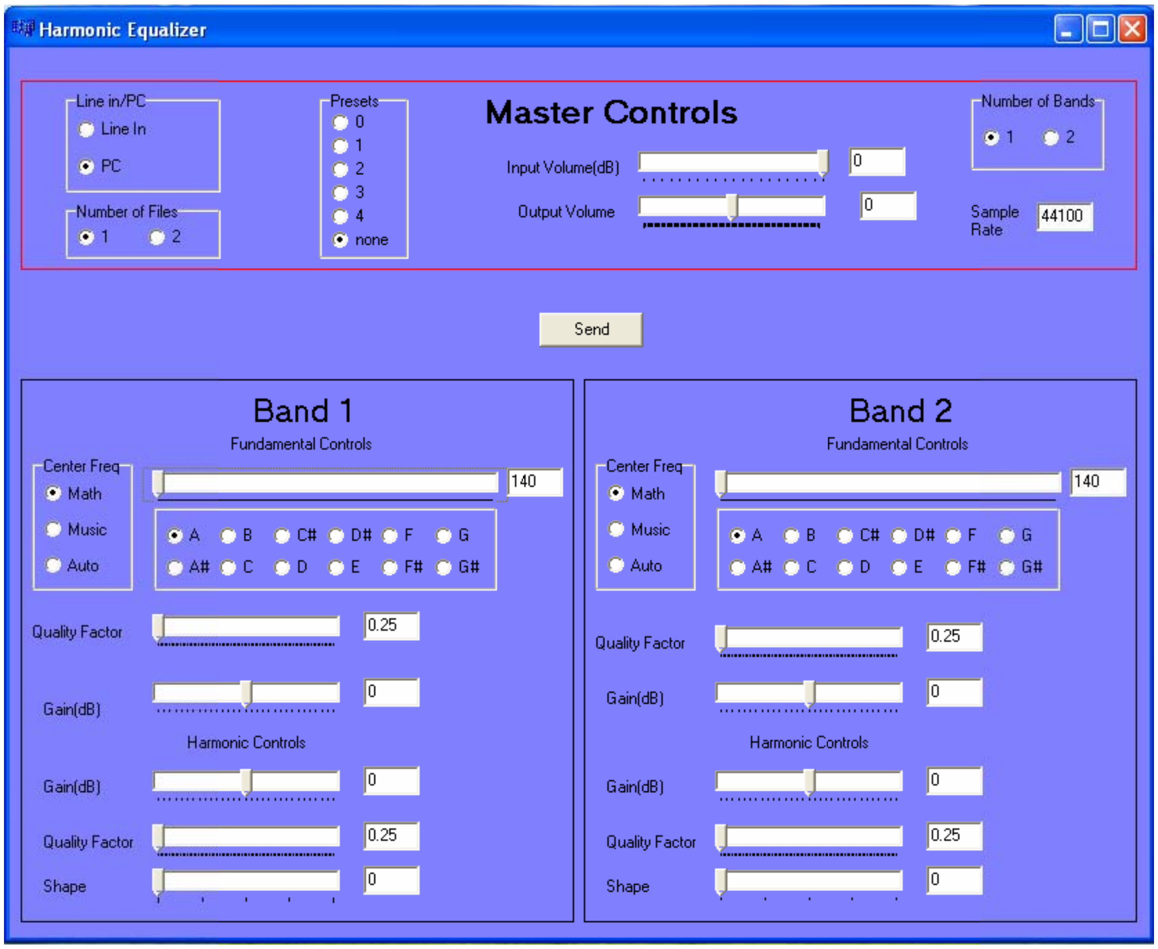
As Group 9 of Spring 2003 found, using the Autocorrelation method to detect pitch was “terribly computationally inefficient”. Autocorrelation involves shifting a FFT multiple times to figure out when both versions match, producing a peak. Since harmonics will line up, this will produce a strong peak at the fundamental frequency. The main benefit to Autocorrelation is it is impervious to noise. The biggest disadvantage aside from inefficiency is it can misinterpret a harmonic as the fundamental. Charpentier’s algorithm is way faster since it only involves one FFT and a few multiplies instead of a FFT and many shifts and multiplies. We chose to use Charpentier since noise was not much of an issue with our high fidelity recordings, and our device needs to be as efficient as possible. In the long run, Charpentier’s method is a very accurate of detecting monophonic pitches, making this an easy decision.

On the next page is an example of our pitch tracking algorithm’s output. A musical scale (consecutive notes) from A to E played on a violin is the input.



Even though we once said our pitch tracking didn’t have to be ‘dead on’ since the filter’s Q would always be wider than any small pitch tracking error (of +- Input frequency * 2^(1/12) hz), our pitch tracker seems to be right on anyway, so *c’est la vie*.

THE GUI



Line in/PC: Line In [✓ PC]
Number of Files: [✓ 1] 2
Presets: 0 1 2 3 4 [✓ none]
Input Volume(dB): 0
Output Volume: 0
Number of Bands: [✓ 1] 2
Sample Rate: 44100

Band 1 – Fundamental Controls
Center Freq: [✓ Math] Music Auto
140
[✓ A] B C# D# F G A# C D E F# G#
Quality Factor: 0.25
Gain(dB): 0
Band 1 – Harmonic Controls
Gain(dB): 0
Quality Factor: 0.25
Shape: 0

Band 2 – Fundamental Controls
Center Freq: [✓ Math] Music Auto
140
[✓ A] B C# D# F G A# C D E F# G#
Quality Factor: 0.25
Gain(dB): 0
Band 2 – Harmonic Controls
Gain(dB): 0
Quality Factor: 0.25
Shape: 0

Our Graphic User Interface (GUI) was created using Borland C++ builder 6. All of the code on the PC side was then written on various source files within the Borland project. This enabled us to create one single executable file that incorporated all our functionality.

Controls:

- Master Controls
- Band 1 Controls
- Band 2 Controls
- Presets

Selection of Audio Files:

The GUI allows the user to select one or two audio files. This is done by right-clicking on the GUI. The second audio file is to make use of the added on multitrack feature.

How the GUI works:

We first created a basic form onto which we then added components like track bars, radio buttons, file select dialog boxes, text boxes, labels and buttons. These components are used to control parameters of the filter such as gain, frequency, quality etc.

The user can adjust the controls and choose audio files as per his/her wishes. When he/she presses the send button, all the filter parameters from the GUI are captured in a floating point array. A connection is then established between the PC and the DSK as was done in the labs. These parameters are then sent to the DSK where the filter is created dynamically and the audio file is processed.

CODE USED AND WHAT WE HAD TO MODIFY:

We used C code we found online to play the music real time. We had to modify the code so that it would fit our buffer size. Furthermore, the code played music it read from a file where as we need to play music that was being sent to us in packets from the DSK. We also modified the code for communication between the PC and the DSK. This was done so that we could send more 5 commands between the two. The rest of the code that we used was integrated into our program without many changes.

WHAT CODE WE WROTE:

- *Read wav files*
- *Calculates the filters and their properties.*
- *Charpentier's pitch tracking code*
- *GUI*
- *Filtering Code*

SPEED ISSUES:

One large concern for us was the fact that our project had to run real time. This was a monumental task that required us to optimize our code such that it could run with as few cycles as possible. In order to do so, we had to keep the limitations of the DSK and the PC in mind.

The first hurdle in achieving real time processing is the transfer rate between the PC and DSK. Ideally, we would get 2MB/s transfer rate from the PC to the DSK and 10MB/s transfer rate from the DSK to the PC. However, the actual transfer rate between the PC and DSK and the transfer rate between the DSK and PC was not ideal. The reason these numbers are not ideal has to do with the fact that we are only 1024 samples at a time. With such a small transfer, the overheads costs make up a large portion of the total transfer cost.

The other major timing concern for us was the actual processing of the input samples. First we had to find a reasonable level for our sample size. If the sample size was too large, we would take too long to process the whole sample. If the sample size was too small, it would cost more to transfer and process the data per sample. We decided to process 1024 samples at a time on the DSK. This was a nice balance between size and speed.

Our original code for processing the samples was not optimized. The code required approximately 393 cycles to process each sample. When we looked at TI code already available, we found that TI provided us with code for filtering signals through a bi-quad. Using the TI assembly code, we could achieve speeds of almost 4 cycles per bi-quad filter. As we might use up to 14 filters, this would provide us with speeds of 56 cycles per sample. However, despite working with the teaching assistant we were not able to implement the assembly version of this filter. So we opted to use the C version of the same code. The C version of the bi-quad filtering code was still more efficient at filtering our signal than the original code. We were able to get 226 cycles per sample by using the TI C code.

The pitch tracking algorithm takes 4.13 million cycles with full optimization in inclusive mode. However, the exclusive average for the pitch tracking algorithm is only 0.386 million cycles. Ideally, the fft function should only take 14,448 cycles to compute the fft. After profiling the pitch tracking algorithm range by range, we found that 3.8 million cycles our of 4.13 million cycles was used by our shifting loop. The reason this requires 3.8 million cycles is that pow(), sqrt(), and cos() require more than 1000 cycles. Since we use these functions frequently in this loop, the required number of cycles is very large. If there was a way to decrease the number of cycles required to run the math.h functions, then we might be able to run pitch tracking in real time as well.

The main functions in our code are pitch tracking, create filter, and run filter. The following table will illustrate the cycles required to run each function under different levels of optimization:

	Pitch Tracking (cycles)	Create Filter (cycles)	Run Filter (cycles)
No Optimization	Took Too Long	100897	55365
Local Optimization(-o1)	Took Too Long	52243	16463
Function (-o2)	Took Too Long	52177	16450
Full Optimization (-o3)	4.13 MM	52149	16421

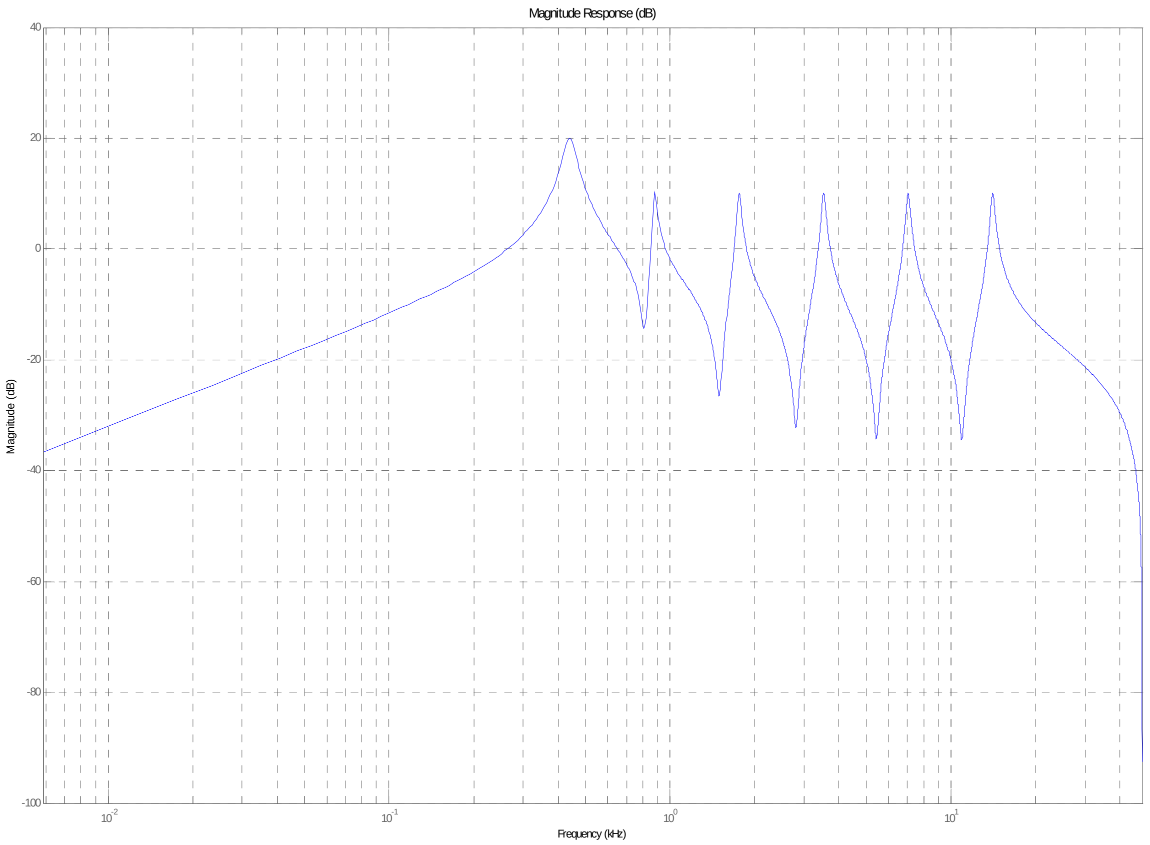
Another concern we had regarded memory. Our initial algorithm required a circular buffer of size 2048. We split it into two buckets of size 1024 each. We would bring in 1024 samples into the first bucket and process it. We then sent the processed sample back out and brought the next 1024 samples into the second bucket. The reason for this was the fact that a FIR filter is dependent on data from the past. By using this circular buffer, we could access points we needed that were in the past. However, this method was inefficient and we used the TI filter. This allowed us to cut the required amount of memory in half. This becomes an issue because we attempt to place all of our data on chip. By doing so, we decrease the number of cycles required to access each sample.

Comparison and results

So how do we know if our project is working how we want?

- Input white noise, and FFT that output.
- Our ears.

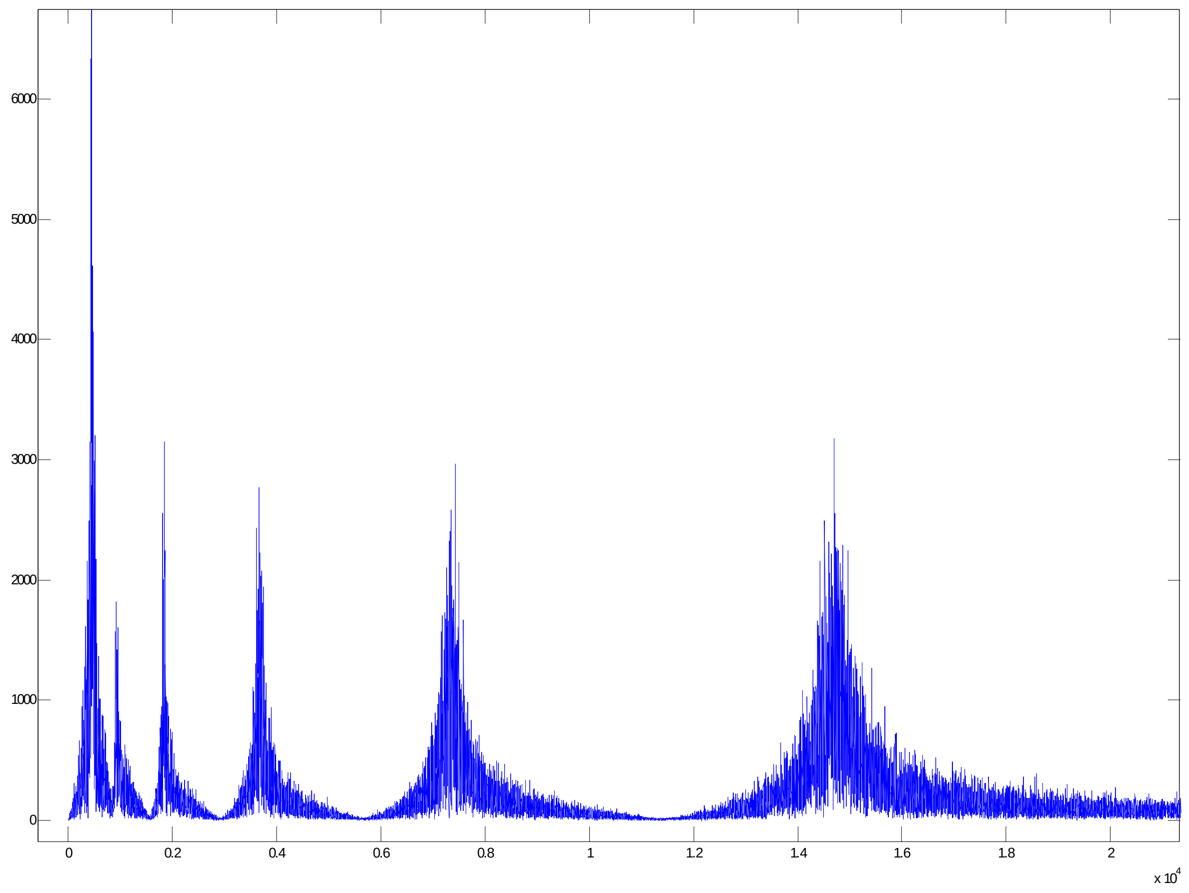
Below is an example implementation of our device:



Fund: 440hz, Q:1, Gain 20,

Harm: Q: 10, Gain 10. Filter Order: 12

After running white noise through our device this is the output:



It is quite clear that our system is responding as expected. Though now for the real test, listening. While picking music for examples, and playing with the filter settings to process the audio examples, there was definetly a learning curve for even us (the ones who built the darn thing). Being personally used to standard EQs, it definitly took some getting used to in order to start producing results. Not to brag, but our filters sounded darned good. Even if we piped in a commercial (already mixed and mastered) cd, we still could get certain qualities out of the track.

After processing three examples, one commercial stereo track, one just piano, one just violing, we took a poll from fellow lab partners to see which version they prefered… processed or unprocessed.

Ear test: 3 different audio samples (25 participants):

	Favored	*Abstain*	*Disliked*
Sample 1 ("Fallen")	*18*	*5*	*2*
Sample 2 (Piano Chords)	*15*	*6*	*4*
Sample 3 (Violin)	*21*	*4*	*0*

Clearly the majority was in our device's favor. But was it? I believe it's more a testament to the fact that I used our device properly. Just as easily I used our device to make the samples sound a certain way, I could have made it sound another way in which more people could have disliked. Though in my professional opinion what differs our project from any other commercial device, to achieve the same sound and result, it would have taken nearly a day to create the same samples. Where as it took me 30 mins to settle on a good result with our device.

CONCLUSION

In conclusion, our project was a huge success. It was a bit bumpy the night of our demo, where a small bug had to be quickly attended to, however nonetheless we got it back working. We would like to acknowledge Prof. Casasent, his TAs: Eric Chu, Rohit Patnaik, and Xinde Hu, and Faculty Assistant Marilyn Patete for all of their countless help and support through out the design and construction of this project. Thank you!

REFERENCES

1. Tutorial: Using the Windows waveOut Interface

Author: David Overton

July 2002

http://www.insomniavisions.com/documents/tutorials/wave.php

This tutorial is designed to help you use the windows waveOut interface for playing digital audio. Through this tutorial I will build a windows console application for playing raw digital audio. When we started out project, we had very little knowledge about how music is stored in a digital format. This tutorial gave us a good background on 'sample', 'sample size' , 'channels' and 'sampling rate' and how these concepts are related to the quality of the music.

The higher the sampling rate, the more like the analogue wave your sampled wave becomes, so the higher the quality of the sound. The larger the sample sizes the higher the quality of the audio. We followed the basic structure of the tutorial and modified it to suit our needs. We changed the buffer size and the sampling rate to hear the effect on the quality of music.

The steps involved were:

- Introduction to Digital Audio
- Opening a Sound device
- Playing a sound
- Streaming audio to the device
- The buffering scheme
- The Driver program

As the audio file was being sent to the DSK for processing, the part of the file that the DSK had finished processing was sent back to the PC and played using the above buffering scheme.

2. Planet-source-code.com

We also found the above tutorial at the following website. The tutorial at insomniavisions.com was more detailed.

http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=4422&lngWId=3

3. Code Guru

Author: Raghuvamshi

January 2005

http://www.codeguru.com/cpp/g-m/multimedia/audio/article.php/c8935/

This article taught us how PCM audio is stored and processed on a PC.

It also explained the structure of wave files and showed us how to work with them using code written in C++. It provided us with a good theoretical background on the structure of wave files and how to process them.

4. Microsoft MSDN

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_playsound.asp

Before we learned about the waveOut interface of windows and how to use a buffering scheme to play streaming music, we looked up the PlaySound function on Microsoft MSDN. The PlaySound function plays a sound specified by the given file name, resource, or system event.

We realized that this function would not be enough for our needs as it did not incorporate a buffering scheme and it required a full wave file to play. Our project starts playing the processed sound file even before processing is complete and thus the PlaySound function was not sufficient for our needs. Initially however, it did give us some insight into how sound is played on a PC and what parameters could be changed.

5. “Pitch detection using the short-term phase spectrum”
by F.J. CHARPENTIER

Published in 1986 by the IEEE
developed at the Centre National d'Etudes des Telecommunications 22301 LANNION FRANCE.

6."Instantaneous—frequency distribution versus time: an interpretation of the phase structure of speech",
by D.H. Friedman, Proc. ICASSP, 1121—1124, Mar.1985.

7. Texas Instruments:

http://focus.ti.com/docs/toolsw/folders/print/sprc121.html#description

- FIR implementation method
- Radix-4 FFT
- Twiddle factors

8. Class Notes:

- DSK to PC networking
- Codec Operation
- Digit Reverse (for FFT)

9. Google:

Couldn’t have done it with out ya! Thanks to Gmail as well!