EE 419 - Project 11

Fun Filtering Times (FFTs)

1) Windowed Frequency Sampling FIR Graphic Equalizer Design

OK! Time to put all this FIR digital filter design stuff to good use! One day,…while studying for your EE419 Final Exam,…you notice that the "audio experience" you are having while listening to pirated songs downloaded from questionable internet sites on your computer is a bit "underwhelming" (to say the least). Between the poor resampling and lossy compression of the music; along with the tiny, tinny speakers in your laptop, it just is NOT a very satisfying auditory experience! "If only there was a way to boost up the bass some more,…and bring out the "highs" a bit, …maybe this would sound better!", you think to yourself. So,…you get to wondering if maybe…just maybe…a little spectral manipulation might help matters. Perhaps, Dr. Fourier could rescue you from the doldrums of ordinary computer-generated audio(?!).

Being a "closet audiophile", you know that the typical way to overcome shortcomings in the spectral reproduction of "cheap" audio equipment, is to use a "Graphic Equalizer" to selectively boost or attenuate different "bands" of frequencies (portions of the frequency spectrum) that are under or over-emphasized by the amplifiers and speakers producing the sound (or in the original recording). So,… you valiantly set off on a *quest* to improve your computer-based audio sound experience, while also maybe learning some new things and putting together some old concepts that might even help you get a better grade on that EE 419 Final! [*Orchestra swells…..our hero rides off into the sunset….atop his (or her) trusty 12-speed bike…iTunes at the ready,…earbuds in…DESTINY (or at least Spring Break) awaits!!*]

So,…your mission, should you decide to accept it, is to design a ***Programmable Graphic Equalizer*** that runs on Matlab, using a digital filter to perform the spectral enhancements on **audio data input from WAV (.wav) audio files** (with $\mathbf{f_{sample}}$**=44.1 KHz** for CD quality audio). Since you are only satisfied with "the very best" when it comes to audio equipment, your Graphic Equalizer will have **9 frequency bands** (no mere "treble" and "bass" controls for YOU!!) . A typical division of the audio range for this number of bands, would place the **centers of the 9 ranges at: 62.5 Hz, 125 Hz, 250 Hz, 500 Hz, 1 KHz, 2 KHz, 4 KHz, 8 KHz, and 16 KHz**. [Notice the power of 2 relationship between adjacent frequency band centers. These power of 2 increments would correspond musically to going up 1 "octave" in the pitch of the sounds. This non-uniform band width adds a little to the challenge of designing the equalizer.]

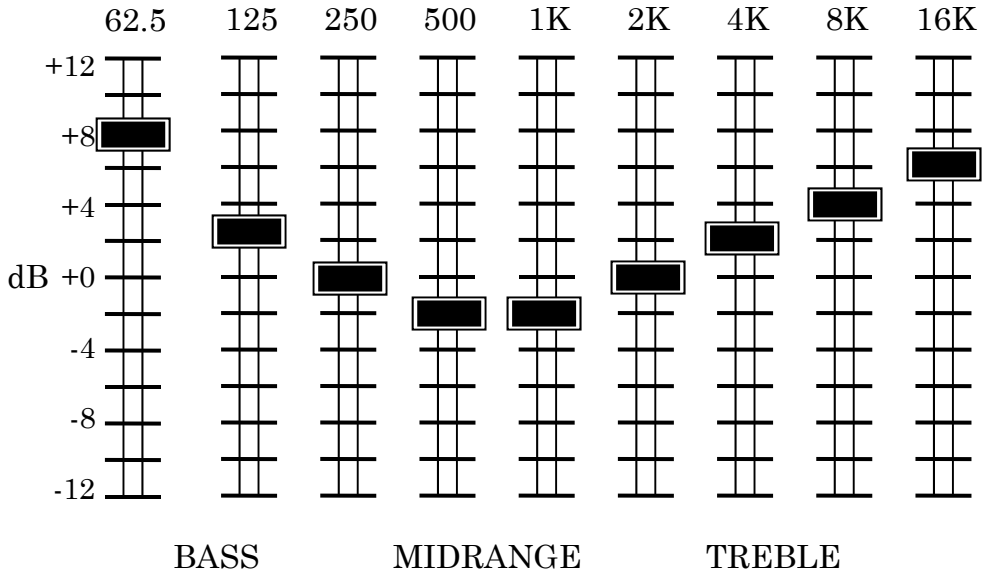
In a typical design for an audio equalizer, the lowest band (62.5 Hz) would generally be implemented as a low-pass filter, with the gain setting for this band applied to all frequencies below the 62.5 Hz center frequency. Likewise, the top band would be

implemented as a high-pass filter, with the same uniform gain applied to all frequencies above 16 KHz (up to the Nyquist frequency). The intermediate bands would typically all be implemented with moderate Q bandpass filters, with overlapping rolloffs to achieve a fairly flat overall response if all the filters were set for 0 dB gain. Alternatively, parallel *shelving filters* that can either attenuate or boost a range of frequencies but that nominally have a gain of 1 outside of the altered frequency band would be used to created the equalizer. [Your Blandford & Parr textbook has a good description of such a device in Section 9.4.8.]

Since you ***really*** don't want to design **9**! different digital filters for this "little" project, you decide to try a shortcut method: using **FIR Design by Frequency Sampling-With Windowing** to create an arbitrary frequency spectrum shape; even if it means doing so with a really big number of frequency samples. ("It just means more computer processing time and more memory,.....and I'm willing to wait for "perfection", you reason to yourself.) The resulting FIR filter will be looooooooooooong; BUT it will have NO PHASE DISTORTION (which appeals to the "audiophile" in you); and you can apply it to audio data using your superfast FFT-based convolution program (that you created in Project #3).

One additional accommodation to how we experience sound that is usually incorporated into equalizers is the fact that we perceive sound intensity (loudness) on a *logarithmic* scale. Therefore, equalizers usually use a dB-scale for setting the gain of each frequency band; often over a range of -12 dB to +12 dB (with 0 dB being no change in signal amplitude in a given frequency range).

EQUALIZER SETTINGS



So, your Matlab program should accept an array of 9 values (in units of dBs) that the user wants to assign to each of the 9 frequency bands (from lowest frequency to highest). For example, for the settings shown above, your program would be passed an array of equalizer dB values of [8, 2.4, 0, -2, -2, 0, 2, 4, 6] corresponding to the dB boost desired for each of the 9 frequency bands. Your program should then compute the filter coefficients (impulse response) required to implement a frequency response comparable to what was set by the user.
A basic method that you can apply would be:

1. Select the **frequency sample spacing** required to place samples at all of the desired band centers (especially the 1st one). This will establish the number of samples (M) needed. Make sure that you choose M to be odd.
2. Determine which exact frequency sample value indexes are closest to the given band center frequencies. Assign the selected dB gain setting for the corresponding frequency band to a "dB version" of the Frequency Sampling HF magnitude sample value at those band center locations.
3. Perform a **linear interpolation of dB values** for frequency samples in between the ones just set. (Use a straight line slope to fill in the dB values for samples that fall between the "known" ones you set in Step 2.]
4. Convert all the sample dB magnitudes to linear magnitude values, as is needed by the Frequency Sampling design method.
5. Use the **FIR Filter Design by Frequency Sampling method** to convert the linear magnitude sample values (several hundred samples long) to an equivalent length FIR filter's unit sample response (and thus the Bk filter coefficients). If you have not done so already, you may want to create a Matlab function to perform this conversion. Here, Matlab's *fir2( )* function will not help you much, as it can not perform the linear interpolation of dB gain values. If you choose to use *fir2( )*, you will have to provide it all the frequency samples and interpolated magnitude response values that you want for your filter (after performing interpolation on the dB values and conversion to linear gains); in the proper format for fir2( )'s **F** and **A** arrays. If you supply it only the 9 equalizer frequencies and gains (plus the F=0 and F=1/2 points), it will linearly interpolate the linear magnitude values (not the dB values) between these points.
6. You can reduce some of the "ringing and ripple" in the frequency response of the resulting filter by **applying a window** to your filter's unit sample response (filter coefficients). While windowing can smooth the frequency response and reduce the ringing in the magnitude response, it will also widen your transition widths. The result is that large gain changes between adjacent frequency bands will not be fully achieved if the windowing is too aggressive. Therefore, you should use a window that only has a small impact on transition widths, such as the Tukey window. Also, since you don't want to lose frequency resolution, use a window that is the same length as the unit sample response determined for your filter by Frequency Sampling. (Do not truncate the filter length any shorter when you window it.)
7. Use your Matlab filter analysis program to look at the resulting frequency response; to see if you achieved something like what was requested.
8. Apply the designed equalization filter to music data samples read in from an audio file, using your fftconv( ) fast convolution by FFT program. Audio files in .wav format are provided on the class PolyLearn site for your testing and debugging.

Zarathustra.wav	Stereo (2 channels) – full length recording (87 sec)
Zarathustra1.wav	Mono (1 channel) – < half length excerpt (39 seconds)
Zarathustra2.wav	Stereo (2 channels) – same half length excerpt (39 seconds)
ZaraExcerpt.wav	Stereo (2 channels) – quarter length excerpt (19.6 seconds)

9. Write the equalized data back out to an audio .wav file (with a different name).
10. Listen to the results using Matlab's soundsc( ) function, AND by playing back the .wav file using Windows Media Player or other similar media utility (iTunes, *etc.*). Can you hear the effects on the music? (It helps to compare it "back to back" with the original, unequalized .wav file.) Make sure that your wav file output is not distorted by signal clipping. If this is a problem, normalize your output sample values so that the largest signal samples are between -1.0 to + 1.0.

OTHER INSTRUCTIONS:

- No,…you may NOT use the Parks-McClellan or Yule-Walker Matlab functions for this.
- Watch out for the orientation of your arrays, especially the data read from wav files. You may have to transpose the data to work with your programs.
- Most wav files for music contain two audio tracks (left and right channels). Your program should **handle either single channel or 2-channel .wav files.** For 2-channel files, you will need to separate out the 2 data sets, individually equalize them, then recombine them before writing back out to the new .wav file.

2) ADD AN ECHO EFFECT TO YOUR EQUALIZER

The characteristics of the sound that we hear when we listen to music is dependent on the acoustics of the room that we are listening in (and the room in which the sound was recorded if we are listening to prerecorded music). Every room has a unique "impulse response" and "frequency response" that depends on the shape and size of the room; the materials in the walls, floor, and ceiling; and the contents of the room (including their shapes, sizes, and materials as well). All these factors affect how the sound reflects or is absorbed by the surfaces in the room. Many materials will absorb or pass through higher frequencies and reflect lower ones, which changes the frequency content of music as it bounces off surfaces. Larger, solid structures will cause uniform reflections that bounce around and arrive later at a listener's ear; which would be perceived as faint echoes. Together these echoes and the frequency-shaping characteristics give a room its unique sonic character. (Acoustic engineers and architects go to great lengths designing concert halls and theaters to achieve uniform and pleasing sound reproduction throughout the room they are designing.)

Many of the sonic characteristics of a room can be compensated for, or deliberately reproduced in a music recording, if the room's impulse and/or frequency response is known. This can actually be measured using special equipment. The frequency shaping characteristics of the room can be compensated for (or replicated) using the filters in an equalizer, like the one you just designed. The surface reflections that cause echoes and reverberations can also be mimicked using digital filters.

Conceptually, an echo is just a delayed version of the original sound that is generally reduced in amplitude. We could represent the sound we hear (y[n]) for an input sound (x[n]) with a single echo (e[n]) in a digital (sampled) sound system as:

$$y[n] = x[n] + e[n] = x[n] + \alpha\, x[n - D]$$

where D is the delay time (in # of sample periods] between when the primary sound x[n] arrives at our ear and when the echo arrives; and α is the attenuation (fractional gain) in the amplitude of the echo compared to the original sound. Multiple echoes would produce additional terms of the same sort:

$$y[n] = x[n] + e_1[n] + e_2[n] + e_3[n] = x[n] + \alpha_1\, x[n - D_1] + \alpha_2\, x[n - D_2] + \alpha_3\, x[n - D_3]$$

where the delays D_k are generally not uniformly spaced, and the amplitudes α_k decrease exponentially with delay time.

The form of the difference equation shown above should look familiar to you – it is nothing more than an FIR filter! The only unique thing here is that the delays are generally fairly long (10's to 100's of milliseconds), and so will span a large number of sample periods (not with terms like x[n-**2**], but terms more like x[n-***2000***]!) This sort of FIR filter is sometimes called a "tapped delay line", since it includes several long delay elements connected in series with a few B_k multipliers tapping off delayed signals that are summed to create the multiple echos.

For the listener, the delay times of the first, strongest series of echoes give our minds cues about the size of the room we are in. The longer the early echo delays, the further away the reflecting walls are, and so the larger the perceived room space is.

Your challenge is to add such an echo reproducing feature to your equalizer. It should add an **arbitrary number of attenuated echoes** using delay times D_k (specified in milliseconds) and fractional gains ($|\alpha_k| \leq 1$) that are specified in arrays. (The program should add as many echoes as there are gains/delays specified.)

The prototype for your m-file to compute and implement the echo should appear as:

```
[echo_filter_hn]= echo_filter(Dk_delays_msec,alphak_gains,Fsample);
```

Finally, use fast convolution to combine the echo filter's effects with the equalizer's filtering using as few extra computations as possible; either as part of the equalizer code or as an added function. (Be sure to apply the echo to both channels of the song.)

TO RECEVE CREDIT:

- **Demonstrate your results!** Process the Zarathustra.wav (full stereo recording) file provided on PolyLearn using the Equalizer Gain settings and echo delays specified below. *(Yes,…they sound awful, but that is OK.)*

  Equalizer Settings (dB): [+12, -6, -12, -12, -6, +12, -6, +12, -6]

  Echo Delays (msec): [250 400 520 660 750 1220]
  Fractional Gains: [0.7 0.6 0.5 0.33 0.2 0.8]

  - **Compute and upload a processed .wav file with the output results to PolyLearn.**

- **Finally, create and demonstrate an *improved version*** of the full, stereo (2-channel) wav file **using your choice of equalizer settings and reverberation echo delays/gains.** Your settings should be chosen **to sound as good as possible** when played through the non-optimal (cheap) speakers we have in the lab or on your laptop computer.
  - **Compute and upload a processed .wav file with the output results to PolyLearn.**

- **Turn in all Matlab code** used for creating and implementing the Equalizer, the delay filter, combining their response responses, and processing a .wav file input (provide

all of the Matlab functions your created). Place all of the Matlab functions that you need into a **single m-file**. (Yes,…you *can* put multiple functions in the same m-file! You can not, however include scripts with functions – just turn any scripts you need into functions without return values.)

- Be sure to include ALL functions in the m-file that are needed to run your equalizer and reverberation systems, without any other non-standard Matlab functions. For example, if you call your own *show_filter_responses( )* or *fftconv( )* functions, be sure to include these in your m-file as well. The m-file must be able to function completely by itself in someone else's computer.
- **Include comments at the top of the m-file with the command(s) needed** to run your function(s) to produce the special test case results.
- **Be sure that your functions are well documented with comments.**
- **Upload your combined function m-file** to PolyLearn.

- **Turn in a brief report, uploaded to PolyLearn in PDF or MSWord format, that includes the following:**
  - **Turn in a frequency response plot** (at least magnitude response) using a dB scale showing the filter frequency response produced by your Equalizer program **using the test case settings given above.**
  - **Document the equalizer and echo settings** you used to create your "good" results, **and explain why you chose those settings** (what were you trying to accomplish?).
  - **Turn in a frequency response plot** (at least magnitude response) using a dB scale showing the filter frequency response produced **when implementing the "good" settings that you chose.**
  - **Include** a copy of your **m-file text with your report**.
  - **Indicate which one of this quarter's lab projects was the "Most Helpful/Interesting" and why; and which was the "Least Helpful/Interesting" and why?**

    Proj 2: Developing a Filter Analysis Program in Matlab
    Proj 3: DFT/FFT Signal Processing – Fast Convolution
    Proj 4: Block Processing and FFT Spectrum Analysis – Touchtone Decoder
    Proj 5: Correlation Detection – Hunt for Red October
    Proj 6: Efficient IIR Implementations and Quantization Effects
    Proj 7: FIR Design By Windowing with "C" Implementation
    Proj 10: IIR Filter Design Comparison
    Proj 11: Audio Signal Processing

Matlab - Supplemental Information

Reading, Saving and Playing a *.wav File in Matlab

a) Microsoft *.wav files may be read as vectors by using the function *audioread*. A vector may be played by the computer as an acoustic sound by using the Matlab functions *sound* or *soundsc*. It may also be saved in the format of Microsoft's *.wav file using the Matlab function *audiowrite*. The sampling rate is specified. Otherwise, the vector will be saved as an 8-bit word at the rate of 8000 samples per second. You can find more information about commands for reading, writing, and playing *.wav files using Matlab's Help menu.

```
>> help audioread
>> help audiowrite
>> help soundsc
>> help audioplayer
```

```
[Y, FS]=audioread(FILENAME) reads an audio file specified by the
    string FILE, returning the sampled data in Y and the sample rate
    FS, in Hertz.

    Y is returned as an m-by-n matrix, where m is the number of audio
    samples read and n is the number of audio channels in the file.
    (If there are 2 audio channels, Y will be a 2-dimensional array.
    One channel's data will be read into Y(:,1), and the 2nd channel
    data will be in Y(:,2)). Y is of type double, and matrix elements
    are normalized values between -1.0 and 1.0.

[Y, FS]=audioread(FILENAME, [START END]) returns only samples START
    through END from each channel in the file.


audiowrite(FILENAME,Y,FS)  writes data Y to an audio file specified
    by the file name FILENAME, with a sample rate of FS Hz.
    Stereo data should be specified as a matrix with two columns.
    Multi-channel data should be specified as a matrix of N columns.

    The file format to use when writing the file is inferred from
    FILENAME's extension.  Supported formats are as follows:

    Format         File Extension(s)   Compression Method
    ------------------------------------------------------
    Wave           .wav                None
    MPEG-4 Audio   .m4a, .mp4          AAC
    FLAC           .flac               FLAC (Lossless)
    Ogg/Vorbis     .ogg, .oga          Vorbis
```

The following program creates a vector x from the sound file cht5.wav. The sampling rate in cht5.wav is 22050 Hz. The program then plays it at that rate and also at a new (slowed down) sample rate of 11025 Hz; and saves it as a new file *cht5-slo.wav* using the new sampling rate.

```
[x,Fs] = audioread('cht5.wav');
soundsc (x, Fs) ;
soundsc (x, Fs/2) ;
audiowrite('cht5-slo.wav',x, Fs/2);
```