Lab #4

Filtering Signals in the Frequency Domain

Summary: This development of these labs was supported by the National Science Foundation under Grant No. DUE-0511635.

Introduction

In this lab, we will look at the effect of filtering signals using a frequency-domain implementation of an LTI system, i.e., multiplying the Fourier transform of the input signal with the frequency response of the system. In particular, we will filter sound signals, and investigate both low-pass and high-pass filters. Recall that a low-pass filter filters out high frequencies, allowing only the low frequencies to pass through. A high-pass filter does the opposite.

MATLAB Commands and Resources

- `help <command>`, online help for a command.
- `fft`, Fast Fourier Transform.
- `ifft`, Inverse Fourier Transform.
- `sound`, plays sound unscaled (clips input to [-1,1]).
- `soundsc`, plays sound scaled (scales input to [-1,1]).
- `audioread`, reads in WAV file. The sampling rate of the WAV file can also be retrieved, for example, `[x, Fs] = audioread('filename.wav')`, where `x` is the sound vector and `Fs` is the sampling rate.

Transforming Signals to the Frequency Domain and Back

When working in MATLAB, the continuous-time Fourier transform cannot be done by the computer exactly, but a digital approximation is done instead. The approximation uses the discrete Fourier transform. There are a couple of important differences between the discrete Fourier transforms on the computer and the continuous Fourier transforms you are working with in class: finite frequency range and discrete frequency samples. The frequency range is related to the sampling frequency of the signal. In the example below, where we find the Fourier transform of the "**fall**" signal, the sampling frequency is **`Fs=8000`**, so the frequency range is **[-4000,4000]** Hz (or **2*pi** times that for **w** in radians). The frequency resolution depends on the length of the signal (which is also the length of the frequency representation).

The MATLAB command for finding the Fourier transform of a signal is **`fft`** (for Fast Fourier Transform (FFT)). In this class, we only need the default version.

```
>> load fall      %load in the signal
>> x = fall;
>> X = fft(x);
```

The **fft** command in MATLAB returns an **uncentered** result. The **first half** of X contains the **positive** frequencies, while the **second half** contains the **negative** frequencies. To view the frequency content in the same way as we are used to seeing it in class, you need to plot only the first half of the result (positive frequencies only) OR use the MATLAB command **fftshift** which toggles between centered and uncentered versions of the frequency domain. **Y = fftshift(SS)** rearranges a Fourier transform SS by shifting the zero-frequency component to the center of the array. If **SS** is a vector, then **fftshift** swaps the left and right halves of **SS**. For example:

>>SS = [1, 2+3j , 2-j, 0.5, 0.5, 2+j, 2-3j]; % You can also use i for complex numbers
>>Y = fftshift(SS)

```
Y =

   0.5000 + 0.0000i   2.0000 + 1.0000i   2.0000 - 3.0000i   1.0000 + 0.0000i   2.0000 + 3.0000i   2.0000 - 1.0000i   0.5000 + 0.0000i
```

Now the value 1 (1.0000+0.0000i, DC component) in SS is moved to the center of Y.

Using **fftshift** you can also convert back and forth between the centered and uncentered versions in the frequency domain.

The sample rate **Fs** for the 'fall' sound is 8,000 Hz. The code below will allow you to view the frequency content both ways.

```
>> N = length(x);              % add a command "Fs=8000;' for the 'fall' sound
>> pfreq = [0:N/2]*Fs/N;       % index of positive frequencies in fft
>> Xpos=X(1:N/2+1);            % subset of fft values at positive frequencies
>> plot(pfreq,abs(Xpos));      % plot magnitude of fft at positive frequencies
>> figure;
>> freq = [-(N/2-1):N/2]*Fs/N;  % index of positive AND negative freqs
>> plot(freq,abs(fftshift(X))); % fftshift actually SWAPS halves of X here.
See help.
% Convince yourself of why it does this to match up with freq!
```

Note that we are using **abs** in the plot to view the magnitude since the Fourier transform of the signal is complex-valued. (Type **X(2)** to see this. Note that **X(1)** is the DC term, so this will be real-valued.)

Try looking at the frequency content of a few other signals. Note that the fall signal happens to have an even length, so N/2 is an integer. If the length is odd, you may have indexing problems, so it is easiest to just omit the last sample, as in **x=x(1:length(x)-1);.**

After you make modifications of a signal in the frequency domain, you typically want to get back to the time domain. The MATLAB command **ifft** will accomplish this task.

```
>> xnew = real(ifft(X));
```

You need the `real` command because the inverse Fourier transform returns a complex-valued vector, since some changes that you make in the frequency domain could result in that. If your changes maintain complex symmetry in the frequency domain, then the imaginary components should be zero (or very close), but you still need to get rid of them if you want to use the `sound` command to listen to your signal.

Low-pass Filtering

An ideal low-pass filter eliminates high-frequency components entirely, as in:

$$H_L^{ideal}(\omega) = \begin{Bmatrix} 1 & |\omega| \leq B \\ 0 & |\omega| > B \end{Bmatrix}$$

A real low-pass filter typically has low but non-zero values for $|H_L(\omega)|$ at high frequencies, and a gradual (rather than an immediate) drop in magnitude as frequency increases. The simplest (and least effective) low-pass filter is given by (e.g. using an RC circuit):

$$H_L(\omega) = \frac{\alpha}{\alpha + j\omega}, \alpha = \text{cutoff frequency.}$$

This low-pass filter can be implemented in MATLAB using what we know about the Fourier transform. Remember that multiplication in the Frequency domain equals convolution in the time domain. If our signal and filter are both in the frequency domain, we can simply multiply them to produce the result of the system.

$$y(t) = x(t) * h(t)$$
$$Y(\omega) = X(\omega)H(\omega)$$

Below is an example of using MATLAB to perform low-pass filtering on the input signal `x` with the FFT and the filter definition above.

The cutoff of the low-pass filter is defined by the constant `a`. The low-pass filter equation above defines the filter `H` in the frequency domain. Because the definition assumes the filter is centered around w = 0, the vector `w` is defined as such.

```
>> load fall      %load in the signal
>> x = fall;
>> X = fft(x);    % get the Fourier transform (uncentered)

>> N = length(X);
>> a = 100*2*pi;
>> w = (-N/2+1:(N/2))*Fs/N*2*pi;  % centered frequency vector (rad/s)
>> H = a ./ (a + i*w);            % generate centered sampling of H
```

```
>> plot(w/(2*pi),abs(H))          % w converted back to Hz for plotting
```

The plot will show the form of the frequency response of a system that we are used to looking at, but we need to shift it to match the form that the `fft` gave us for `x`.

```
>> Hshift = fftshift(H);      % uncentered version of H
>> Y = X .* Hshift';          % filter the signal (uncentered)
```

Your filter's response and the FFT spectrum of the signal to be filtered must be consistent, i.e., both should be either centered or uncentered.

Note:

If you are having problems multiplying vectors together, make sure that the vectors are the exact same size. Also, even if two vectors are the same length, they may not be the same size. For example, a **row** vector and **column** vector of the same length cannot be multiplied element-wise unless one of the vectors is transposed. The `'` **operator** transposes vectors/matrices in MATLAB.

Now that we have the output of the system in the frequency domain, it must be transformed back to the time domain using the inverse FFT. Play the original and modified sound to see if you can hear a difference. Remember to use the sampling frequency Fs.

```
>> y = real(ifft(Y));   % you must do 'ifft' on the uncertered version
>> sound(x, Fs)         % original sound
>> sound(y, Fs)         % low-pass-filtered sound in the time domain
```

The filter reduced the signal's amplitude, which you can hear when you use the `sound` command but not with the `soundsc` which does automatic scaling. Replay the sounds with the `soundsc` and see what other differences there are in the filtered vs. original signals. What changes could you make to the filter to make a greater difference?

Note:

Sometimes, you may want to amplify the signal so that it has the same height as the original, e.g., for plotting purposes.

```
>> y = y * (max(abs(x))/max(abs(y)))
```

Now you are ready to do some exercises.

Exercise 1

Low-pass Filtering with Sound

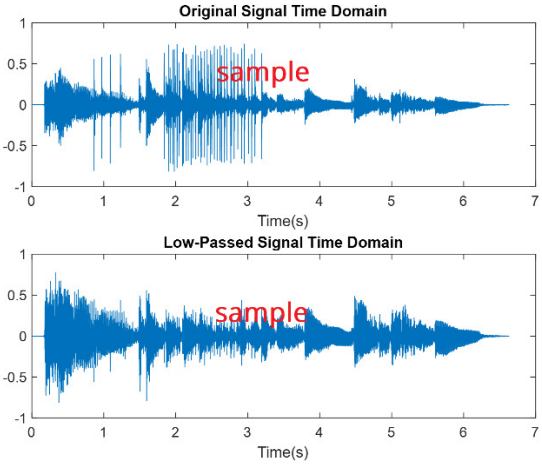
Write your script file **SignalLowPass.m** to perform the low-pass filtering. Use `audioread` to load the sound castanets44m.wav. Perform low-pass filtering with the filter defined above, starting with `a = 500*2*pi`, but also try different values.

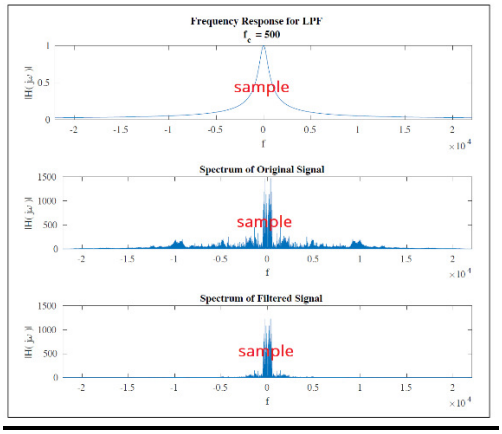
Play the original and the low-passed versions of the sound.

On one figure, plot the original and the low-passed versions of the sound using subplots. Make sure the horizontal axis is time in seconds. You need to create an appropriate time vector to plot it correctly (**t=[0:length(signal)-1)]/Fs**).

On another figure with 3 subplots, plot the frequency content (Fourier transforms) of the original and low-passed signal and make sure they are **centered**. You can apply **fftshfit** to the uncentered spectrum after filtering to create a centered version. On the same figure, plot the frequency response (**abs(H)**) of your low-pass filter.

Please include your code and the above plots in your report.





Exercise 2

High-pass Filtering

An ideal high-pass filter eliminates low-frequency components entirely, as in:

$$H_H^{ideal}(\omega) = \begin{cases} 0 & |\omega| < B \\ 1 & |\omega| \geq B \end{cases}$$

A real high-pass filter typically has low but non-zero values for $|H_H(\omega)|$ at low frequencies, and a gradual (rather than an immediate) rise in magnitude as frequency increases. The simplest (and least effective) high-pass filter is given by (e.g. using an RC circuit):

$$H_H(\omega) = 1 - H_L(\omega) = 1 - \frac{\alpha}{\alpha + j\omega}, \alpha = \text{cutoff frequency}.$$

This filter can be implemented in the same way as the low pass filter above.

High-pass Filtering with Sound

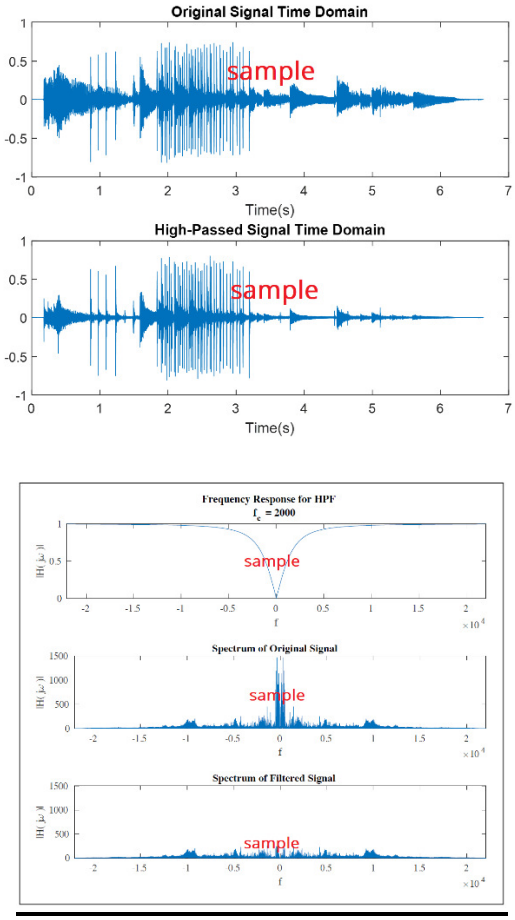
Write your script file **SignalHighPass.m** to perform the high-pass filtering. The high-pass filter can be implemented in MATLAB much the same way as the low-pass filter. Perform high-pass filtering with the filter defined above on the sound castanets44m.wav. Start with `a = 2000*2*pi`, but also try different values.

Play the original and the high-passed versions of the sound.

On one figure, plot the original and the high-passed versions of the sound using subplots. Make sure the horizontal axis is time in seconds.

On another figure with 3 subplots, plot the frequency content (Fourier transforms) of the original and high-passed signal (make sure they are **centered**) and the frequency response (**abs(H)**) of your high-pass filter.

Please include the code and above plots in your report



Exercise 3

Band-pass Filtering

Band-pass Filtering with Sound

It is possible to combine a low-pass filter and a high-pass filter to create a band-pass filter.

Now design a high-pass filter with `a = 500*2*pi` and a low-pass filter with `a = 2000*2*pi`. The two filters can be implemented in MATLAB much the same way as the low-pass/high-pass filter in the above exercises.

There are **two ways** of doing bandpass filtering on the sound castanets44m.wav:

1) filters in cascade, i.e., filtering the audio signal first with the **high-pass** filter $H_H(\omega)$ and then with the **low-pass** filter $H_L(\omega)$;

2) combining them into one single band-pass filter. The frequency response of the bandpass filter should be $\boldsymbol{H_L(\omega) \cdot H_H(\omega)}$ [note: it is NOT their sum]. Write your script file **SignalBandPass.m** to perform the band-pass filtering.

Play the original and the band-passed versions of the sound.

On one figure, plot the original and the band-passed versions of the sound using subplots. Make sure the horizontal axis is time in seconds.

On another figure with 3 subplots, plot the frequency content (Fourier transforms) of the original and band-passed signal (make sure they are **centered**) and the frequency response of your band-pass filter.

Please include your code and the above plots in your report.