Lab 5
Filtering Periodic Signals

EE 235: Continuous-Time Linear Systems
Department of Electrical Engineering
University of Washington

1 Introduction

In this lab, you will look at the effect of filtering signals with a frequency-domain implementation of an LTI system. This is accomplished by multiplying the Fourier transform of the input signal with the frequency response of the system. In particular, you will apply filters to sound signals, and then 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.

2 Useful MATLAB Commands

You may need to use the following commands. As always, familiarize yourself with their syntax before using them.

- `fft` – Fast Fourier Transform command. Use the default syntax.
- `ifft` – the Inverse Fourier Transform.
- `fftshift` – displays frequencies symmetrically centered around 0.
- `sound` – plays a sound unscaled (clips input to [-1,1]).
- `soundsc` – plays a sound scaled (scales/normalizes input to [-1,1]).

[1] Last revision: Tue May 18 17:05:24 PDT 2010
[2] http://www.creativecommons.org/licenses/by/2.0

3 Transforming Signals to the Frequency Domain and Back

An exact continuous-time Fourier transform cannot be done by the computer exactly, so MATLAB does a digital approximation instead. The approximation uses the Discrete Fourier Transform (DFT), which you will study in depth when you take EE 341. There are two important differences between the continuous Fourier transforms you are working with here in EE 235 and the discrete Fourier transforms you use in MATLAB.

The continuous Fourier transform you've seen is smooth and has an infinite range: ω can take on any value within $(-\infty, \infty)$. However, the DFT has a 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` sound, the sampling frequency is `Fs = 8000` so the frequency range is [-4000,4000] Hz (multiply by 2π to compute ω in radians/sec). 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`, which stands for the Fast Fourier Transform (FFT). This is a particular algorithm for computing the Discrete Fourier Transform rather efficiently. You'll learn the details in EE 341.

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

The `fft` command in MATLAB returns an uncentered result: the frequencies start at 0. To view the frequency content in the same way you are used to seeing it in class, you need to plot only the first half of the result. This gives you the positive frequencies only. You can also use the MATLAB command `fftshift` which toggles between centered and uncentered versions of the frequency domain.

The code below will allow you to view the frequency content both ways:

```
>> N = length(x);
>> 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 fftshift
```

Exercise 1:
• Why does `fftshift` swap the halves of `X` in order to center the frequencies for display?

Note that because the Fourier transform of the signal is complex-valued, `abs` is used in the `plot` to view the magnitude.

Exercise 2:
- Verify that the Fourier transform is complex-valued by examining the value of `X(2)`.
- Is `X(1)` complex or real? Why?

Look at the frequency content of a few other signals. Note that the `fall` signal happens to have a length N that is even, 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);`

You usually want to get back to the time domain after modifying a signal in the frequency domain. The MATLAB command `ifft` will accomplish this task:

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

The `real` command is needed because the inverse Fourier transform returns a vector that is complex-valued, because some of the operations that you can make in the frequency domain produce complex outputs. 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 explicitly get rid of them if you want to use the `sound` command to listen to your signal.

4 Low-Pass Filtering

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

$$H_L^{ideal}(\omega) = \begin{cases} 1 & \text{if } |\omega| \leq B \\ 0 & \text{if } |\omega| < B \end{cases}$$

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 the frequency ω increases. The simplest (and least effective) low-pass filter is given by:

$$H_L(\omega) = \frac{\alpha}{\alpha + j\omega}$$

Here α is the cutoff frequency. This filter can be built with a simple RC circuit.

This low-pass filter can be implemented in MATLAB using what we know about the Fourier transform. Remember that multiplication in the Frequency domain is the same operation as convolution in the time domain. If our both our signal and filter are in the frequency domain, then we can simply multiply them to produce the (frequency) output of the system:
$y(t) = x(t) \star h(t)$
$Y(\omega) = X(\omega)H(\omega)$

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

The cutoff of the low-pass filter is defined by the constant α. The low-pass filter equation defines the filter `H` in the frequency domain. Because the definition assumes the filter is centered around $\omega = 0$, the vector ω is defined to be centered at 0.

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

>> 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 as you are used to seeing it, but it needs to be shifted in order match the form that the `fft` gave for `x`:

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

You now have the output of the system in the frequency domain, so the next step is to transform it back to the time domain, using the inverse FFT. The sound can then be played (remember to specify `Fs`, the sampling frequency):

```
>> y = real(ifft(Y));
>> sound(x, Fs)           % original sound
>> sound(y, Fs)           % low-pass-filtered sound
```

4.1 Low-Pass Filtering of Sound

Exercise 3:
• Download `castanets44m.wav`[3].
Filter it with a low-pass filter that has a cutoff frequency $\alpha = 500 * 2\pi$.
Show plots at each intermediate step.
• Use `sound` to play the original and modified castanets. What is the main difference?

The filter reduced the signal amplitude, which you can hear when you use the `sound` command but not with the `soundsc` command, because `soundsc` does automatic scaling. Sometime– e.g. for plotting purposes – you may want to amplify the output signal so that it has the same magnitude as the original:

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

Exercise 4:
• Replay the input and output castanets with `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?

Exercise 5:
• Choose a different cutoff frequency α to make a different low-pass filter.
• Filter `castanets` with your new filter.
Show plots at each intermediate step.
• How does the output compare to that of the output using the first filter?
Use `soundsc` to listen to all three versions (input, first and second filter outputs).

4.2 Low-Pass Filtering of Impulses

Exercise 6:
• Create an input signal `x` that is an impulse train, as follows:

```
>> x = [ repmat([zeros(1, 99) 1], 1, 5) zeros(1,99)];
```

• Use a low-pass filter with $\alpha = 20$ to low-pass the impulse train. Call the result `y`.
• Plot each of the two signals `x` and `y` against time, using `subplot`.
Label the axes and title each graph appropriately.
• Explain what the low-pass filter is doing to the impulse train.

5 High-Pass Filtering

An ideal high-pass filter eliminates low frequency components entirely, and is defined as the complement of a low-pass filter:

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

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:

$$H_H(\omega) = 1 - H_L(\omega) = 1 - \frac{\alpha}{\alpha + j\omega}$$

Much like for a low-pass filter, α is the cutoff frequency.

5.1 High-Pass Filtering of Sound

A high-pass filter can be implemented in MATLAB much the same way as the low-pass filter.

Exercise 7:
• Perform high-pass filtering on the `castanets44m` sound that you downloaded previously.
Use $\alpha = 2000 * 2\pi$ plus at least one other cutoff frequency.
• Play the original and the two high-passed versions of the sound.
Remember to scale the filtered signals so that they all have the same amplitude.
How do the sounds compare?
• Plot the frequency responses. Are they what you expect?

6 Sound Separation

As an example of when to use filters, let's say that Kick'n Retro 235 Inc. recorded a session of a trumpet and drum kit together for their new release. The boss doesn't like the bass drum in the background and wants it out. Unfortunately, there was a malfunction in the mixing board and instead of having two separate tracks for the drums and the trumpet, the sounds mixed together in one track. In order to get this release out on time you will have to use some filtering to eliminate the bass drum from the sound. Of course, there is not enough time to bring the drummer and trumpet player back in the studio to rerecord the track.

Exercise 8:
• Download the `mixed.wav` sound, which has a sampling frequency of 8000 Hz.
This mixed sound was created from `bassdrum.wav`, `hatclosed.wav`, and `shake.mat`.
• Remove the bass drum from `mixed`, and save the result as `mixed-minus-drum.wav`.
Do something easy but approximate first, and then go back to clean it up.
You may find it helpful to look at the Fourier domain representation of the sounds, but you may not use the individual sounds in your solution.
• Now remove the trumpet from `mixed`, and save the result as `mixed-minus-trumpet.wav`.

If you want a sharper filter, you can try using multiple a/(a+jw) terms. Each extra term raises the order of the filter by one; higher order filters have a faster drop-off outside of their passing region. You will see these in detail in EE 341.

6.1 Bonus Problem

This is a bonus exercise. You do not need to do it in order to finish Lab 5.

Say a trumpet and rainstick are recorded together, so that the mixed sound is
`mixedsig = shake + 10*rainstick;`
The producer, who hates Yanni, thinks the rainstick is too new-age and wants it out of the recording. Suppose you do not have the original samples `shake` or `rainstick`. Can you take the signal `mixedsig` and process it such that the output only sounds like the trumpet sample `shake`?

Again, try to do something easy but approximate first, and then improve your system iteratively. You may find it helpful to look at Fourier domain of the sounds, but you may not use `rainstick.mat` nor `shake.mat` in your solution.

You can also try this task with any other complicated pair of sounds that you find.

EOF