Filtering Periodic Signals

Version 1.8: Nov 11, 2008 5:09 pm US/Central

UW EE235 TA UW EE235 TA

Abstract

This development of these labs was supported by the National Science Foundation under Grant No. DUE-0511635. Any opinions, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1 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.

2 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]).
- `wavread`, reads in WAV file. The sampling rate of the WAV file can also be retrieved, for example, `[x, Fs] = wavread('filename.wav')`, where `x` is the sound vector and `Fs` is the sampling rate.

All of the sounds for this lab can be downloaded from the Sound Resources[1] page.

3 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 (more on that in EE 341). There are a couple 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`,

*http://creativecommons.org/licenses/by/2.0/

[1] "Sound resources" <http://cnx.org/content/m13854/latest/>

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

```
≫ N = length(x);
≫ pfreq = [0:N/2]*Fs/N;
≫ Xpos=X(1:N/2+1);
≫ plot(pfreq,abs(Xpos));
≫ figure;
≫ freq = [-(N/2-1):N/2]*Fs/N;
≫ plot(freq,abs(fftshift(X)));
```

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 vector that is complex-valued, since some changes that you make in the frequence 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.

4 Low-pass Filtering

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

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

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));  % centered frequency vector
≫ H = a ./ (a + i*w);       % centered version of H
≫ plot(w*Fs/N,abs(H))
```

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
```

> 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));
≫ sound(x, Fs)        % original sound
≫ sound(y, Fs)        % low-pass-filtered sound
```

The filter reduced the signal 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)))
```

Exercise 1
Low-pass Filtering with Sound Use `wavread` 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 version of the sound. Plot their frequency content (Fourier transforms) as well.

Exercise 2 OPTIONAL: Low-pass Filtering

1. Create an impulse train as the input signal `x(t)` using the following MATLAB command,

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

2. Use the low-pass filter defined earlier to low-pass the impulse train. Choose a cutoff of 20.
3. Plot the two signals `x(t)` and `y(t)` separately using the subplot command. These should be plotted versus the time vector. Label the axes and title each graph appropriately.
4. Look at the plots. Can you explain what is happening to the spike train?

5 High-pass Filtering

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

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

A real high-pass filter typically has low but non-zero values for $|H_L(\omega)|$ 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.

Exercise 3
High-pass Filtering with Sound 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 version of the sound. The filtered signal may be to be scaled so that both have the same range on the Y-axis. Plot their frequency responses as well.

6 Sound Separation

Exercise 4 Sound Filtering

- 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. There is not enough time to bring the drummer and trumpet player back in the studio to rerecord the track.
- Click here to download the mixed.wav sound[2] (`Fs` = 8000 Hz). The mixed sound is created from bassdrum.wav, hatclosed.wav, and shake.mat.
- Try to do something easy but approximate first, and then, if you have more time, see how clean you can get the sound. 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 try to eliminate the trumpet sound, leaving only the drums left in the sound.
- Hint: use high-pass and low-pass filtering.
- Another hint: If you want a more powerful filter, you can try using multiple a/(a+jw) terms in series. Each extra term raises the order of the filter by one and higher order filters have a faster drop-off outside of their passing region.

Exercise 5 BONUS PROBLEM: Sound Filtering

- Imagine you recorded a trumpet and rainstick together, so that you have the signal, `mixedsig = shake + 10*rainstick`.
- It turns out the producer thinks the rainstick is too new-age and wants it out of the recording. Pretend you do not have the original signals shake or rainstick. Can you take the signal `mixedsig` and process it to get (approximately) only the trumpet sound (shake) out? Try to do something easy but approximate first, and then if you have more time, see how good a reproduction of shake you can get. You may find it helpful to look at Fourier domain of the sounds, but you may not use rainstick.mat or shake.mat in your solution.

[2] http://cnx.org/content/m14481/latest/mixed.wav