

Supplement to Chapter 5 of *The Science of Digital Media* – Digital Audio Processing

Mathematical Modeling Exercise – Digital Audio Processing > Creating FIR and IIR Filters with MATLAB¹

Modeling Environment: MATLAB

Introduction:

Digital audio filters are divided into two main categories: FIR (finite-impulse response) and IIR (infinite-impulse response) filters. Equation 1 describes an FIR filter of order N .

$$\mathbf{y}(n) = \mathbf{h}(n) \otimes \mathbf{x}(n) = \sum_{k=0}^{N-1} \mathbf{h}(k) \mathbf{x}(n-k)$$

where $\mathbf{x}(n-k) = 0$ if $n-k < 0$

Equation 1

$\mathbf{h}(n)$ is just a vector of multipliers – i.e., coefficients – to be applied successively to sample values. The number of coefficients is the order of a filter.

In contrast, to describe an IIR, we need a mask of infinite length, given by this equation:

$$\mathbf{y}(n) = \mathbf{h}(n) \otimes \mathbf{x}(n) = \sum_{k=0}^{\infty} \mathbf{h}(k) \mathbf{x}(n-k)$$

where $\mathbf{x}(n-k) = 0$ if $n-k < 0$ and k is theoretically infinite.

Equation 2

Finding the values $\mathbf{h}(n)$ for an infinitely long mask is impossible, but Equation 2 can be transformed to a more manageable difference equation form.

$$\mathbf{y}(n) = \mathbf{h}(n) \otimes \mathbf{x}(n) = \sum_{k=0}^{N-1} a_k \mathbf{x}(n-k) - \sum_{k=1}^M b_k \mathbf{y}(n-k)$$

Equation 3

In Equation 3, N is the order of the forward filter and M is the order of the feedback filter. Thus, the output sample from an IIR filter is determined by present and past input samples as well as past output samples. Notice from Equation 1 that output samples from an FIR filter are determined by present and past input samples only.

¹This material is based on work supported by the National Science Foundation under Grant No. DUE-0340969. This worksheet was written by Grey Ballard and Jennifer Burg (burg@wfu.edu).

The task in this exercise is to use MATLAB's built-in functions to determine what the coefficients $h(n)$ should be, so that $h(n)$ changes a digital signal in the desired way. The equations above all describe convolutions in the time domain. If we want to design a low-pass filter, it is very difficult to choose coefficients in the time domain that will filter out high frequencies. Instead, we design and implement our filters in the frequency domain and then use the inverse Fourier transform to generate the output sample in the time domain.

Tips for Using MATLAB:

When loading sound wave files (.wav files) into MATLAB with the *wavread* function, make sure your files are in the current directory. The wave files you use may have been created somewhere else – in recorded as saved with Audacity, Audition, or Sound Forge, for example – or they may be uncopyrighted files that are freely available on the web. After you load a wave file into MATLAB, it's just an array of values. You can operate on these values in any way you want.

After you've processed audio samples in MATLAB – mixing wave files or filtering the samples, for example – and you want to save them again as a wave file, you can use *wavwrite* command. This will save the data as a *wave* file in the current directory—you only need to specify the array and the desired filename.

When plotting samples and filters, you may have to set your axes to different limits in order to understand the graphs. For example, if you plot a one second sound clip sampled at 8000 Hz, the graph will show all 8000 points. To get a feel for the wave, zoom in using the axis command:

```
>> axis([1 100 -1 1]);
```

This will change the limits for the current figure. Also, to plot multiple samples on one graph, use MATLAB's hold function to add to the current figure instead of replacing it.

```
>> plot(input,'blue');  
>> hold on  
>> plot(output,'red');
```

When using the *wavplay* function, make sure to specify the sampling rate—the default is 11025 Hz.

Exercise 1

Generate an input sample using a sound editing program like Audacity, Audition, Logic Pro, or Sound Forge. Load the sample values into MATLAB. We will need three different tones to mix in order to create our input sample. Using an 8000 Hz sampling rate, we will generate one second samples of a low A note (440 Hz), a high A note (880 Hz), and a much higher frequency

(2000 Hz). To generate a tone in Audacity, for example, you first create a new audio track and make sure it is set to Mono with 8000 Hz sampling rate and 16-bit depth. Then generate a tone with Waveform set to Sine, Amplitude set to 1.00, and the desired frequency. Make sure the sample is only one second long. Then export it as a *.wav* file. Label the files by frequency: *440.wav*, *880.wav*, and *2000.wav*.

Now we will load the wav files into MATLAB and mix them. Use the *wavread* function to load the files into arrays, labeling the variables *lowA*, *highA*, and *highest*. For example,

```
>> lowA = wavread('440.wav');
```

To hear the sample, use the *wavplay* function and specify the sampling rate:

```
>> wavplay(lowA,8000);
```

MATLAB stores sound samples with normalized amplitude (from -1 to 1). Therefore, after we add the three samples together, we have to normalize so that the mixed amplitudes will lie between -1 and 1. Use this command to mix the samples:

```
>> mixed = (lowA+highA+highest) / 3;
```

Play the mixed sample and try to hear the different frequencies. Our goal will be to filter out the highest frequency. First, we will use a classical IIR filter known as a Butterworth filter. The *butter* function will help you to create the frequency domain coefficients of a low-pass (by default) Butterworth filter. It takes two arguments: the order of the desired filter, and the normalized cutoff frequency. Note: the Nyquist frequency (1/2 the sampling rate) is normalized to 1, so the normalized cutoff frequency should lie between 0 and 1. The *butter* function returns two arrays of coefficients in the frequency domain, so use the function in this way:

```
>> [a,b] = butter(n,Fn);
```

where *n* is the order and *Fn* is the normalized cutoff frequency. With a simple low-pass filter, an order of 6 will work fine. Choose your *Fn* so that the A notes pass through the filter but the highest frequency is filtered out. Now that we have our coefficients, we can implement our filter using the filter function. The filter function takes the coefficients and the input sample as arguments:

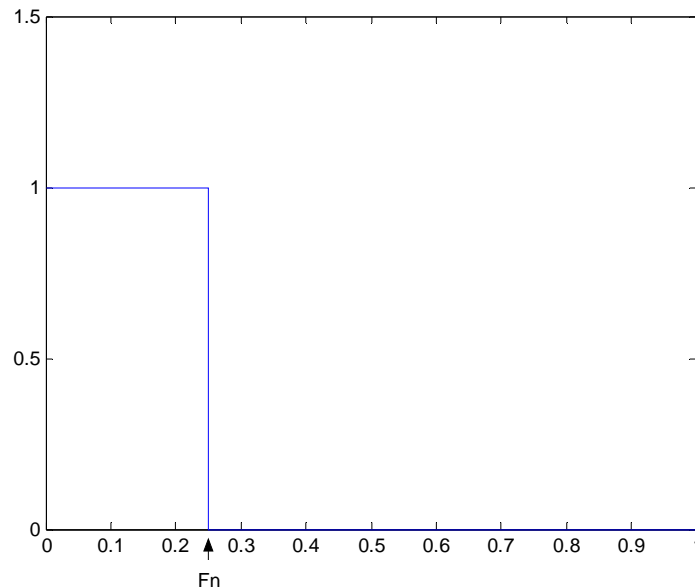
```
>> output = filter(a,b,input);
```

Play the filtered sample and compare it to the mixed sample. You should be able to notice the absence of the highest frequency in the output sample. Now create a mix of the *lowA* and the *highA* samples. This sample should

sound like the filtered sample. To compare them visually, plot them on the same graph and note any differences in the two samples.

Exercise 2

Now we will create the same low-pass filter, but we will design it ourselves as an IIR filter. First, we need to visualize our ideal frequency response graph. For a low-pass filter, the ideal frequency response should like this:

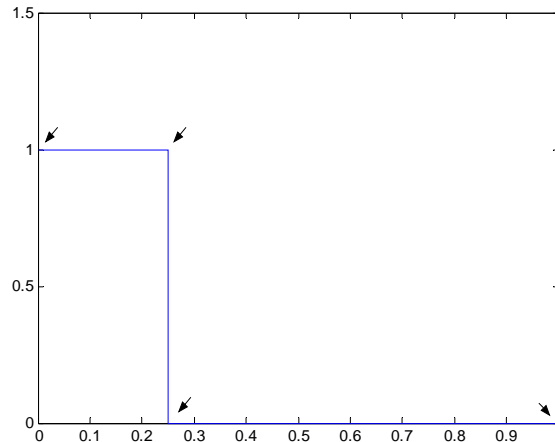


Frequency response of an ideal low-pass filter

Here, the x-axis represents normalized frequencies, and F_n is the cutoff frequency. We can store this graphical information in a pair of arrays: one will store different frequencies, and the other will store the corresponding magnitudes. For our low-pass filter, we only need four element arrays:

```
>> f = [0 Fn Fn 1];  
>> m = [1 1 0 0];
```

Note that by storing the cutoff frequency magnitude twice, we can draw the vertical line in the graph. The arrows below indicate the four points we've stored in the arrays.



Now that we have an ideal response, we use the *yulewalk* function in MATLAB to determine what coefficients to use to approximate the ideal response. The *yulewalk* function takes as arguments the order of the filter and the arrays that represent the ideal response.

☛ **Aside:** The *yulewalk* function in MATLAB is named for the Yule-Walker equations, a set of linear equations used in AR modeling (auto-regression).

```
>> [a,b] = yulewalk(n,f,m);
```

Again, an order 6 filter will be sufficient for the low-pass filter. Use the same filter function as above to perform a filter on the mixed sample with the new coefficients. Compare (audibly and visually) the filtered sample to a mix of *lowA* and *highA* using the *wavplay* and *plot* functions. Are there any differences between the filtered sample created by the Butterworth filter and the filtered sample created with your direct design filter?

Exercise 3

Now we will design our own low-pass FIR filter using the same method as in Exercise 2. The finite counterpart to the *yulewalk* function is the *fir2* function. The *fir2* function takes the order and the ideal response as arguments and returns a single array of frequency domain coefficients. Note that the *yulewalk* function returned two arrays. This is because the second array corresponded to the feedback part of the filter. Recall that IIR filters depend in part on past outputs, whereas FIR filters only depend on inputs.

```
>> c = fir2(n,f,m);
```

We need to use a higher order filter using the FIR, so create a 30th order filter. Because we want to approximate the same ideal frequency response, we can use the same *f* and *m* arrays as before.

Implement the filter using the *filter* function, but use a 1 in the second argument because there is no feedback part to our filter. Plot the output sample against the sample you expect to get. Do they sound the same?

Exercise 4

Now let's try a more practical example. Suppose you wanted to record the sound of a buzzing fly. You use the best microphone you can find, but there is still unwanted noise in the sample you capture. Our goal will be to filter out the unwanted frequencies in order to capture a better sample of buzzing flies. Open the sample saved as *buzzbeep.wav* in the MATLAB environment and play it (the sample rate is 44100 Hz). (You should be able to get this file at the same website where you picked up this worksheet.) Can you hear the beeping of a construction truck in the background? That frequency is about 1250 Hz. In this exercise, create an IIR notch filter using the direct design method from Exercise 2, then filter the sample so that the beeping is attenuated but the fly buzzing is preserved. You may need to iterate your filter two or three times to completely attenuate the beeping. (Hint: you will need 6 points to define your ideal frequency response.)

There's also a chirping bird in the background. See if you can identify its frequency and filter it out also. (Try looking at the spectral view to identify the frequency.)