Filtering Functions in MATLAB and Python

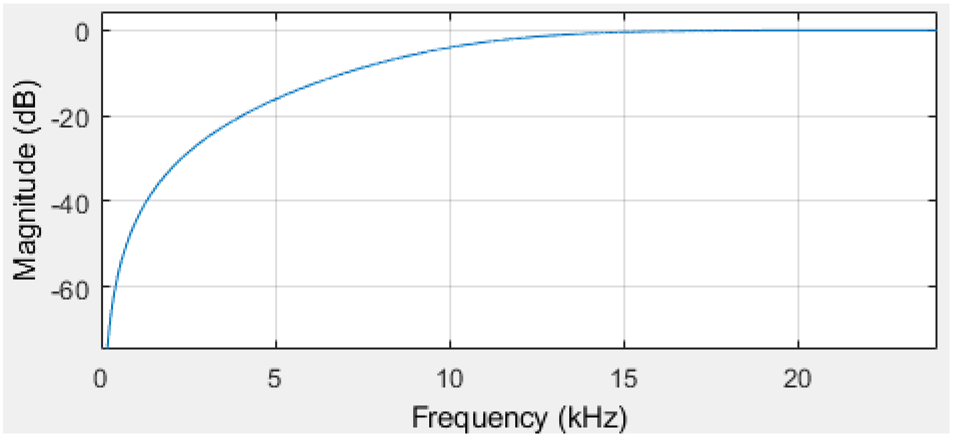
This article describes several digital filter implementations for non-real-time applications using MATLAB and Python as processing tools for data analysis. The demos will be implementing the same filter operation on a wave file(.wav) with two different sounds inside. One sound is a low frequency gong, and the other is a high frequency bird chirp.

Filter Implementations

Essentially, linear filters are weighted sums of previous inputs **(FIR)** and outputs **(IIR)**. For a full definition please see the previous article on filters. This means that implementing filters just takes storage, multiplies, and additions. There are plenty of ways to get this done explicitly, but MATLAB and Python have great function calls that we will be using in the demos below.

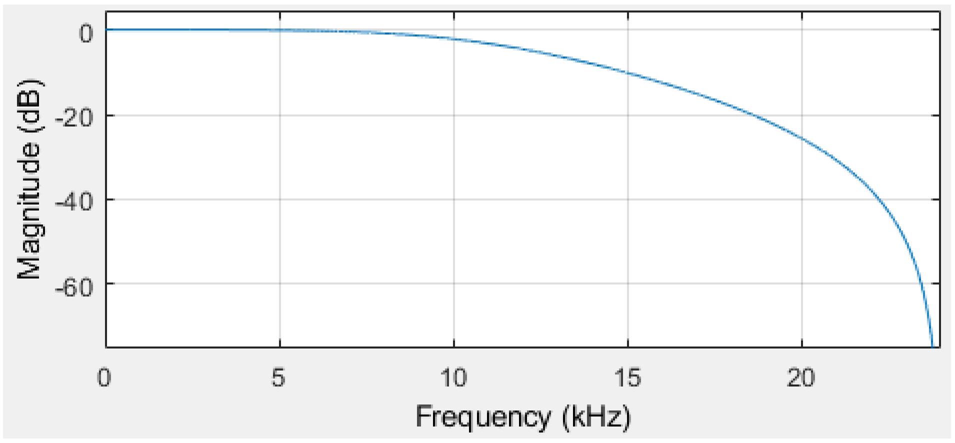
Filter Behavior

Filters have several functions, but the most common is to remove noise and interference. Noise in a signals setting is anything not part of the desired signal. Noise can present itself in several ways, but by identifying what part of the frequency spectrum is affecting the Signal to Noise Ratio **(SNR)** you can choose a filter that improves your data the best.



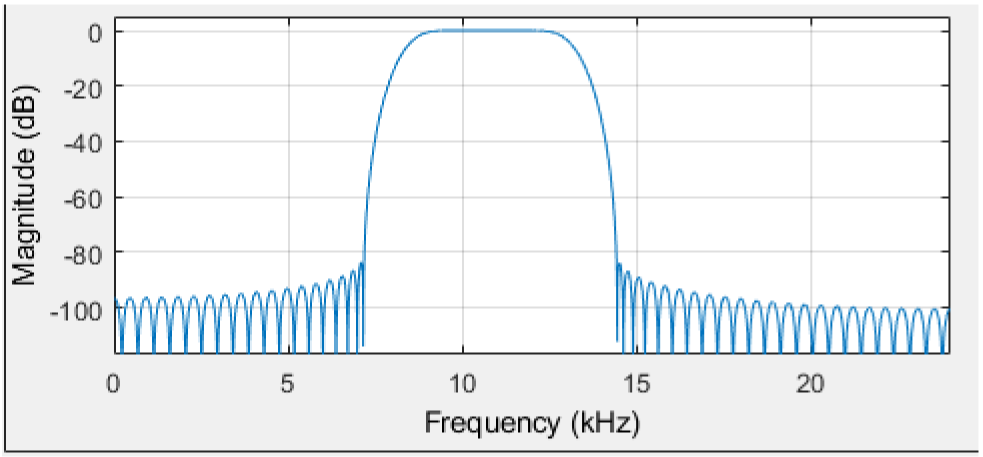
Highpass: $\mathbf{a_0y[n] = b_0x[n] - b_1x[n-1]}$

A filter is said to be highpass if it more easily allows high frequency, fast moving pieces to travel through. The most basic example is above with the current output equal to the current input minus the previous input if a_0, b_0 ,and b_1 are all one. The output does not have anything that was common between the two samples, so the piece of the signal that didn't change within the timeframe of one sample to the next is gone. This allows you to cancel out all frequencies lower than your desired signal.



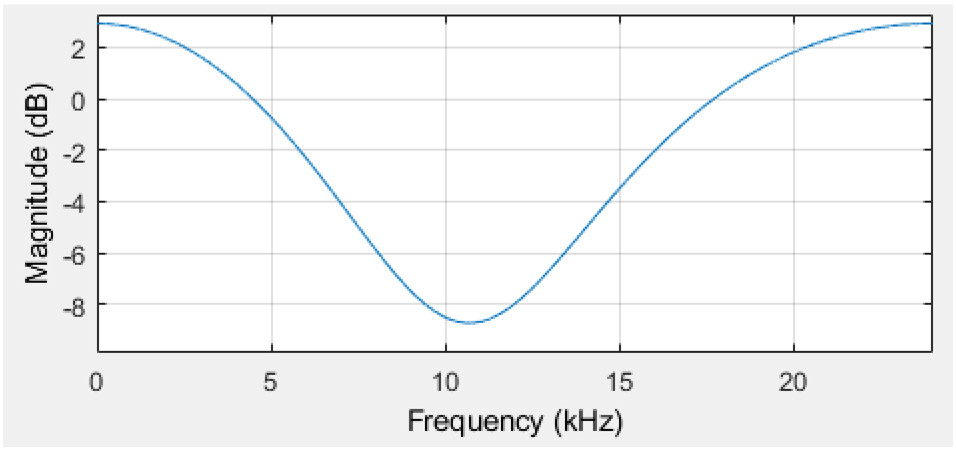
Lowpass: $a_0y[n] = b_0x[n] + b_1x[n-1]$

A filter is said to be lowpass if it more easily allows low frequency, slower moving pieces to travel through. The most basic example is above with the current output equal to the current input plus the previous input if a_0, b_0 ,and b_1 are all one. The output is twice the average of the two samples, so the piece of the signal that changed within the timeframe of one sample to the next is gone. Ideally the input coefficients would be halved, so the output is the same scaling as the input. This allows you to cancel all frequencies higher than your desired signal.

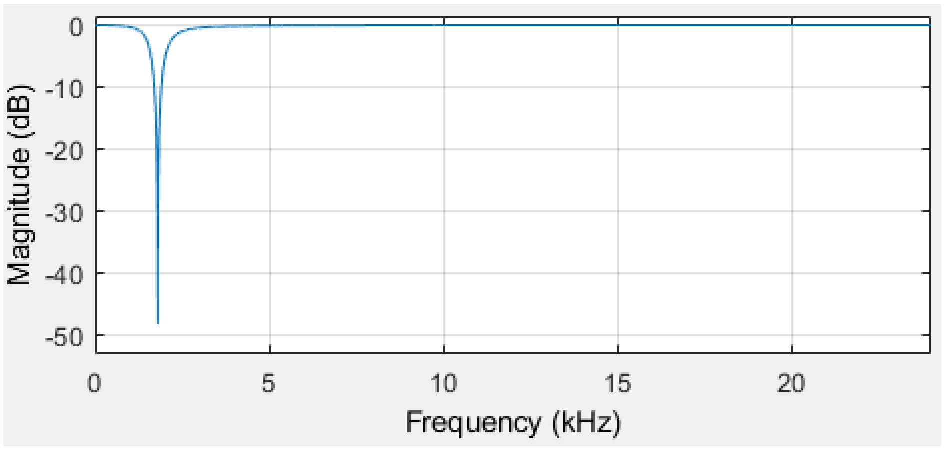


Bandpass: $a_0y[n] = -b_0x[n] + b_1x[n-1] + b_2x[n-2] - b_3x[n-3]$

A filter is said to be bandpass if it more easily allows medium frequency, jugging pieces to travel through. The most basic example is above with the current output equal to the equation if a_0, b_0, b_1, b_2, and b_3 are all one. The output does not have anything that was common between the four samples, and the output does not have anything that was different between any two samples. This allows you to cancel all other frequencies outside of your desired signal.



Bandstop: $a_0y[n] = b_0x[n] - b_1x[n-1] + b_2x[n-2] - b_3x[n-3] + b_4x[n-4]$
A filter is said to be bandstop if it more easily allows a signal's highest and lowest frequencies to travel through. The most basic example is above with the current output equal to the equation if a_0, b_0 , b_1, b_2, b_3, and b_4 are all one. The output does not have anything that was common between two samples or four samples. Both bandpass and bandstop benefit greatly from the use of non-unit coefficients. The examples above do not perform well with all ones. This allows you to cancel out a range of frequencies taking up signal power.



Notch: $a_0y[n] = b_0x[n] + b_1[n-1] - b_1y[n] - b_2y[n-1]$
A filter is said to be notching if it more easily allows all but one frequency to travel through. The most basic example is above with the current output equal to the current input plus the previous two inputs and two outputs. The input and output equations are nearly identical, so the output is zeroed out when the notching frequency is hit. This allows you to cancel out one frequency taking up signal power.

MATLAB DEMO

The first demo is on filtering with MATLAB. The **filter(b, a, x)** function is the focus of this demo. Uncomment the first “sound” and “plotspec” statements to see and listen to the input waveform and spectrum. Then, comment these back out to observe the output of this filter.

filter(b, a, x)
Arguments:

- **(b) Input coefficients**
- **(a) Output coefficients**
- **(x) Input data**

For this example, the coefficients were made using the Parks-McClellan algorithm called with the **firpm** function. This algorithm can be found in FDATool under the name “Least-Squares” in FIR Design Methods.

```
% Modified from EE 445S Professor Brian Evans

%Audio Read gives you the amplitudes(Input) and the sampling frequency(fs)
% of an incoming sound file
[Input, fs] = audioread('twosignals.wav');

Ts = 1 / fs; %Sampling period
fnyquist = fs/2; %fastest frequency represented in Input

%sound(Input, fs); % Listen to the input
%plotspec(Input, Ts); % See its waveform and spectrum

fpass = 1800; % highest unfiltered frequency
fstop = 2000; % lowest fully attenuated frequency
ctfrequencies = [0 fpass fstop fnyquist]; % set cutoffs
pmfrequencies = ctfrequencies / fnyquist; % scale by max frequency
filterOrder = 80; % how many zeros in transfer function
idealAmplitudes = [1 1 0 0]; % lowpass [ 0 1 1 0]bandpass [0 0 1 1]highpass
%filter creation
filterCoeffs = firpm( filterOrder, pmfrequencies, idealAmplitudes );
%filter the input
Filtered = 2*filter(filterCoeffs, 1, Input);
%outputs
sound(Filtered, fs); % Listen to how the filter changes the sound
plotspec(Filtered, Ts); % See how the waveform and its spectrum have changed
```

Python Demo

The second demo is on filtering with Python. The **lfilter(b, a, x)** function in the scipy library is the focus of this demo. Run this code to see how the filter described below works. After running the code, comment out the filter performance graphing, and uncomment the triple quotes ending to watch the filter in action on a dirty input waveform. To run this code you will need Python with numpy, matplotlib, and scipy libraries installed, or you can download the Anaconda development environment.

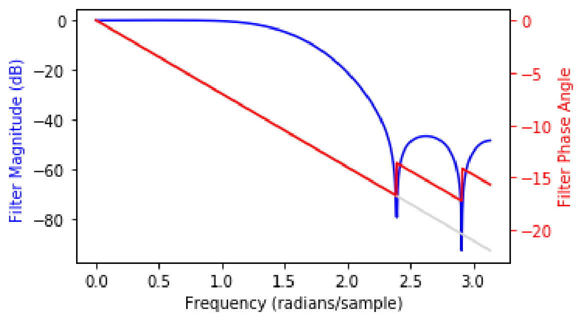
lfilter(b, a, x)
Arguments:

- **(b) Input coefficients**
- **(a) Output coefficients**
- **(x) Input data**

For this example, the coefficients were made using the Windowing algorithm called with the **firwind** function in the scipy library. This algorithm can be found in FDATool under the name "Window" in FIR Design Methods. The frequency behavior is plotted below.

```
# Credit to Foundations and Trends® in Signal Processing

# all the imports you will ever need
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import scipy
import scipy.signal as sig
```



```
# Set numericals
np.set_printoptions(precision=3,suppress=T
rue)

# Set plotting size
matplotlib.rcParams['figure.figsize']=(5.0,1.)

# Design filter
N = 15 #number of coefficients
b = sig.firwin(N,0.5) #0.5 times Nyquist

# Test filter
f, w = sig.freqz(b) #f=radians per sample
```

###[more below]###

```
# Plot filter behavior ~ must be commented out before running the filter below
fig, ax1 = plt.subplots(figsize = ( 5 , 3))
ax1.plot(f,20*np.log10(np.abs(w)),color = 'blue' )
ax1.set_ylabel('Filter Magnitude (dB) ',color = 'blue' )
ax1.set_xlabel( 'Frequency (radians/sample)')
ax2 = ax1.twinx()
ax2.plot(f, -(N-1 ) * f/2 , 'lightgrey' ) #delay of (N-1)/2 samples
ax2.plot(f,np.unwrap(np.angle(w)), 'r' )
ax2.tick_params( 'y' , colors = 'r' )
ax2.set_ylabel( 'Filter Phase Angle' ,color = 'r' );

“””        #Input and filtering operation
t = np .arange(101 )
f1, f2 = 0.05 , 0.35 #cycles per sample
x = np .cos(2*np .pi *f1 *t) + np .sin(2*np .pi *f2 *t)
x += 0.2*np .random .randn(len(t))
plt.plot(t,x, 'lightgrey' ,label = 'orig')
y = sig.lfilter(b,1,x)
plt.plot(t,y,label = 'filtered' )
plt.legend();
“””
```