channel

December 9, 2021

1 A Telephone Channel Simulator

In this notebook we will develop a filter that simulates the effect of a standard telephone channel.

```
In [9]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import IPython
        import scipy.signal as sp
        from scipy.io import wavfile
```

Let's read in an audio file, which we will use as the test signal; the implicit sampling rate will be the internal "clock" of our simulation.

```
In [10]: SF, s = wavfile.read('num9.wav')
         IPython.display.Audio(s, rate=SF)
```

```
Out[10]: <IPython.lib.display.Audio object>
```

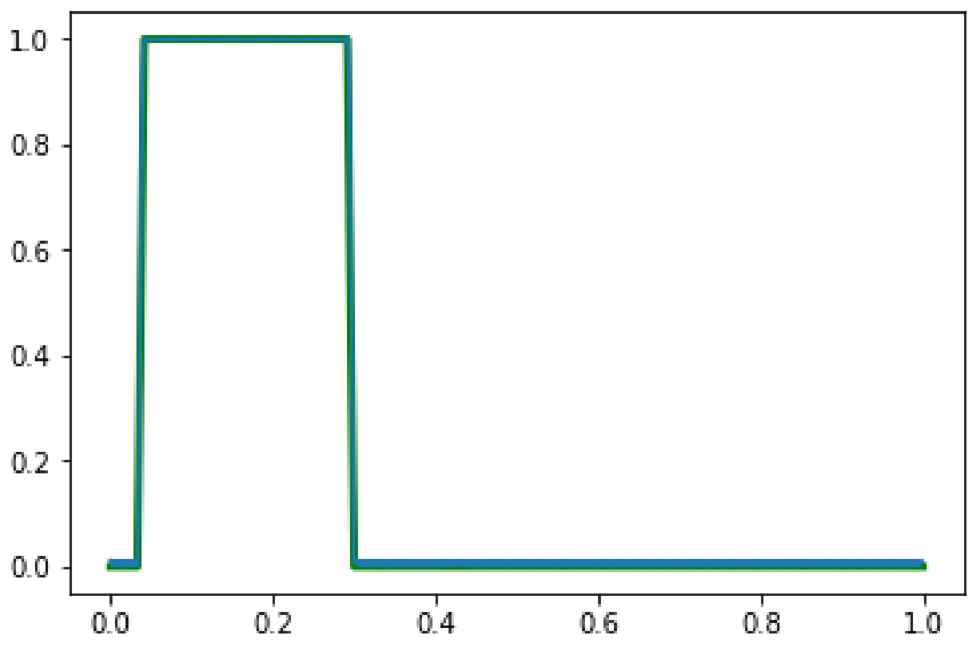
A standard telephone channel acts as a passband approx between 500Hz and 3.5KHz; we can design an optimal FIR design to simulate it.

The scipy Remez algorithm can be used either by specifying real-world frequencies and the sampling rate, or by passing a normalized frequency band vector where the highest frequency is mapped to 0.5 (strange choice). Here we take a more reasonable approach: we use normalized frequencies so that π corresponds to one. This is achieved by normalizing the real-world frequencies by the sampling rate FS and passing a "operational" frequency of 2 to the Remez algorithm:

```
In [11]: # normalized frequencies: x = f / FS
         # band start
         wa = 500.0 / (SF/2)
         # band stop
         wb = 3500.0 / (SF/2)
         # transition band width
         wd = 100. / (SF/2)

         # we need a long filter to ensure the transitions are sharp
         M = 600;
         h=sp.remez(M, [0, wa-wd, wa, wb, wb+wd, 1], [0, 1, 0], [1, 10, 1], Hz=2, maxiter=50)
```

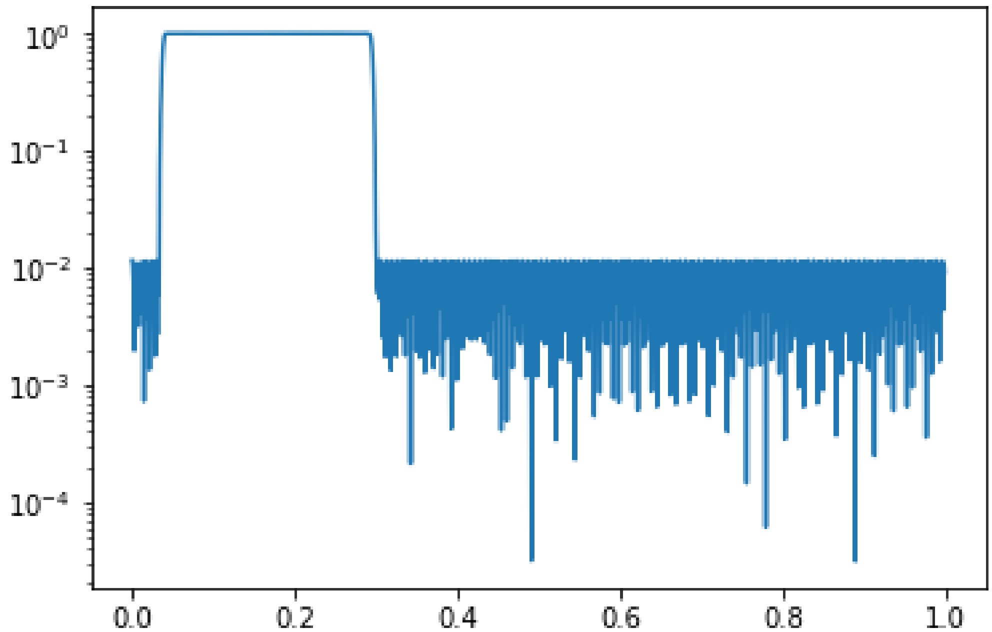
In [12]:
```
# plot frequency response, ideal vs actual
# plot ideal
plt.plot([0, wa-wd, wa, wb, wb+wd, 1], [0, 0, 1, 1, 0, 0], 'green', linewidth=3.0)
plt.hold(True)

# designed filter:
w, H = sp.freqz(h,worN=1024)
# freqz returns a vector of abscissae between 0 and pi
plt.plot(w/np.pi, abs(H));
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:4: MatplotlibDeprecationWarning: py
    Future behavior will be consistent with the long-time default:
    plot commands add elements without first clearing the
    Axes and/or Figure.
  after removing the cwd from sys.path.
/opt/conda/lib/python3.6/site-packages/matplotlib/__init__.py:917: UserWarning: axes.hold is dep
  warnings.warn(self.msg_depr_set % key)
/opt/conda/lib/python3.6/site-packages/matplotlib/rcsetup.py:152: UserWarning: axes.hold is depr
  warnings.warn("axes.hold is deprecated, will be removed in 3.0")
```



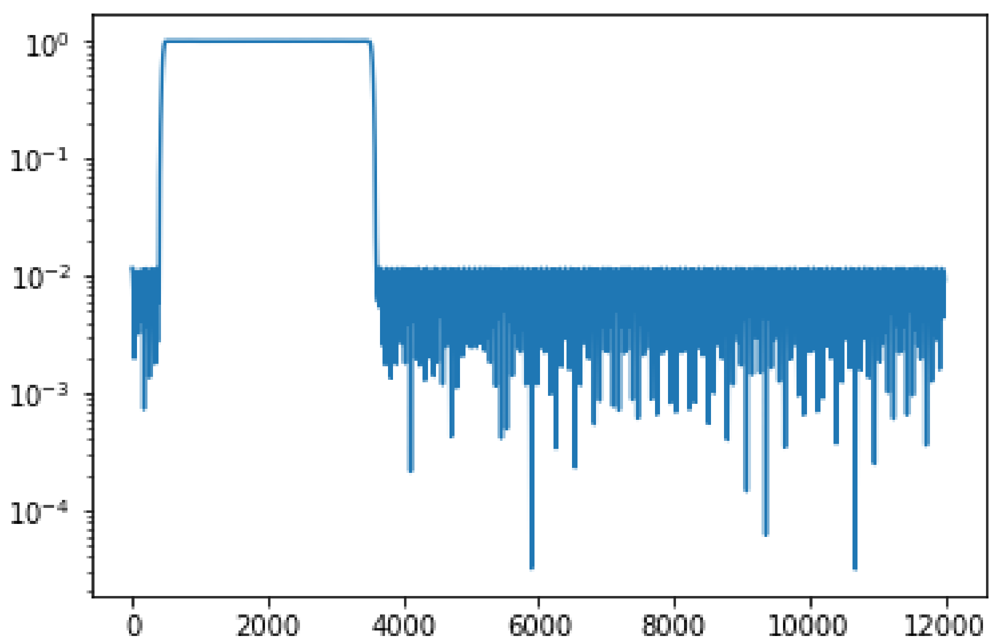
Pretty good fit. Another common way to look at frequency responses is to use a log scale; this allows us to see the attenuation in the bandstop more precisely (and we can see the ripples of the minimax filter):

In [13]:
```
plt.semilogy(w/np.pi, abs(H));
```



With a simple rescaling of the x-axis, we can plot the same response as a function of the real-world frequency:

```
In [14]: plt.semilogy(w/np.pi * (SF/2), abs(H));
```



OK, we can now pass the audio signal through the "telephone" channel and hear the result:

```
In [15]: y = np.convolve(s, h, 'same')
         IPython.display.Audio(y, rate = SF)
```

```
Out[15]: <IPython.lib.display.Audio object>
```

We can now look at the effects of filtering in the frequency domain. Since the signal doesn't have a lot of high frequency content, the acoustic effects of the channel is primarily that of removing the low frequencies up to 500Hz.

```
In [16]: # check the spectrum: first the signal
         # keep the same points for all plots
         N = len(w)
         S = np.abs(np.fft.fft(s, 2*N));
         S = S[0:N];

         plt.semilogy(w/np.pi * (SF/2), S/N, 'cyan')
         plt.semilogy(w/np.pi * (SF/2), abs(H), 'red');

         Y = np.abs(np.fft.fft(y, 2*N));
         Y = Y[0:N];
         plt.semilogy(w/np.pi * (SF/2), Y/N);
```