# EE 242 Lab 4a – Digital Filtering - Various Filters

Wheeler

This lab has 2 exercises to be completed as a team. Each should be given a separate code cell in your Notebook, followed by a markdown cell with report discussion. Your notebook should start with a markdown title and overview cell, which should be followed by an import cell that has the import statements for all assignments. For this assignment, you will need to import: numpy, the wavfile package from scipy.io, simpleaudio/librosa, and matplotlib.pyplot.

In [2]:
```
# We'll refer to this as the "import cell." Every module you import should be i
%matplotlib notebook
import numpy as np
import matplotlib
import scipy.signal as sig
import matplotlib.pyplot as plt
# import whatever other modules you use in this lab -- there are more that you
```

## Summary

In this lab, we will consider different types of digital filters (specifically discrete-time, linear, time-invariant filters) and look at their characterization in both time and frequency. This will give you some insight into how digital filters are implemented and into the properties of different digital filter design algorithms. You'll also learn about some of the signal processing functions available from the signal module in the scipy package which will be useful for designing and implement filters. You will work with examples that show you how filtering can be useful to remove noise and reshape the frequency content of a signal. Specifically, we'll revisit the lab 2 problem of removing noise from signals (or smoothing signals), then explore filter design methods, and finally implement a simple audio equalizer. This is a 2-week lab. It is recommended to work on the first 2 assignments in the first week and the remaining assignments in the second week.

## Lab 4a turn in checklist

• Lab 4a Jupyter notebook with code for the 2 exercises assignment in separate cells. Each assignment cell should contain markdown cells (same as lab overview cells) for the responses to lab report questions. Include your lab members' names at the top of the notebook.

**Please submit the report as PDF** (You may also use : https://www.vertopal.com/ suggested by a student)

## Assignment 1 -- Different Filter Implementations

In this lab, we will be using standard tools to design filters, and we'll want to view them in both the time and frequency domain. In this assignment, you will write and test functions for plotting the frequency response and the impulse response of a system given the filter coefficients {a, b}. This assignment will have three parts, A-C.

**A.** The response that is most often illustrated is the magnitude frequency response on a dB scale. Write a function that takes as input the filter coefficients, an optional flag for plotting both the magnitude and phase of the frequency response, and an optional sampling frequency. The function should generate either a plot of the magnitude or both the magnitude and the phase side-by-side, depending on the flag, with the default being magnitude only. The magnitude of the frequency response should be plotted on a dB scale with a range of [-100,0]. If no sampling frequency is provided, use radians for the frequency axis; otherwise use a Hz scale.

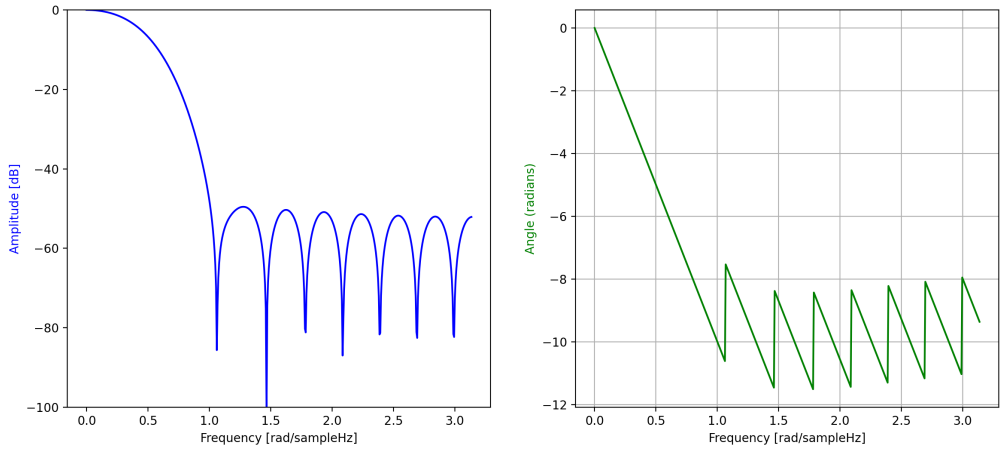
**B.** Write a second function that takes as input the filter coefficients and a desired impulse response length, computes and returns the impulse response, and also plots the impulse response using a stem plot.

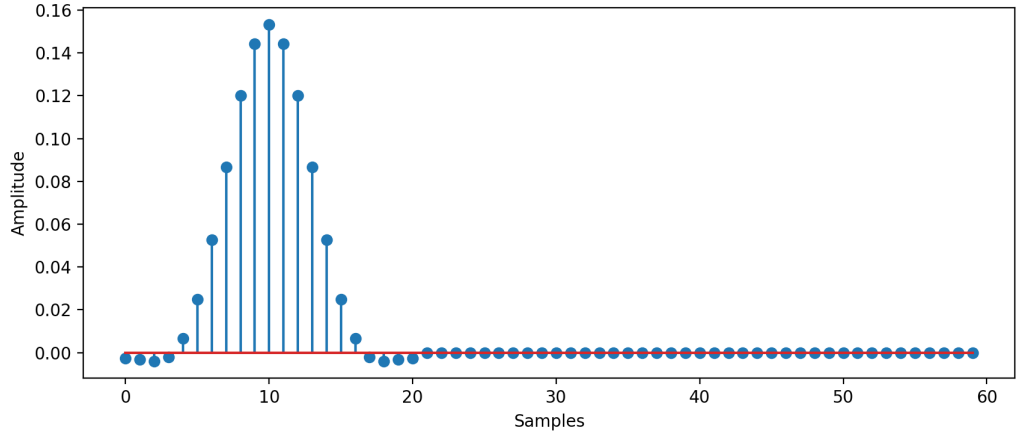
**C.** Test the functions by plotting the magnitude, phase and impulse responses of two lowpass filters with a frequency cut-off of 0.15. One should be an FIR filter designed using the signal.firwin function (order 20) and the other should be an IIR filter with the signal.butter function (order 10).

In [3]:
```
# Assignment 1 - Different Filter Implementations
from scipy import signal

# Part A
def plot_mag_freq_response(b, a, plot_phase = False, fs = None):
    """
    This function plots the magnitude and phase of the frequency response
    of a filter given the filter coefficients a and b.
    """
    # Getting the frequency response
    w, h = signal.freqz(b, a)

    # Checking if a sampling frequency was provided
    if fs:
        # Convert from rad/sample to Hz
        w = w * fs / (2*np.pi)

    # Plotting the magnitude response
    plt.figure(figsize=(14, 6))
    plt.subplot(1, 2 if plot_phase else 1, 1)
    plt.plot(w, 20 * np.log10(abs(h)), 'b')
    plt.ylim([-100, 0])
    plt.ylabel('Amplitude [dB]', color='b')
    plt.xlabel('Frequency [{}Hz]'.format('' if fs else 'rad/sample'))

    if plot_phase:
        # Plotting the phase response on a second subplot
        plt.subplot(1, 2, 2)
        angles = np.unwrap(np.angle(h))
```

```
        plt.plot(w, angles, 'g')
        plt.ylabel('Angle (radians)', color='g')
        plt.grid()
        plt.xlabel('Frequency [{}Hz]'.format('' if fs else 'rad/sample'))

    plt.show()

# Part B
def plot_impulse_response(b, a, impulse_response_length):
    """
    This function computes and plots the impulse response of a filter given the
    filter coefficients a and b and a desired impulse response length.
    """
    # Creating the impulse signal
    impulse = np.zeros(impulse_response_length)
    impulse[0] = 1.0

    # Getting the impulse response
    x = signal.lfilter(b, a, impulse)

    # Plotting the impulse response
    plt.figure(figsize=(10, 4))
    plt.stem(x, use_line_collection=True)
    plt.ylabel('Amplitude')
    plt.xlabel('Samples')
    plt.show()

    return x

# Part C

# Defining the cut-off frequency
fc = 0.15

# Designing a lowpass FIR filter using the signal.firwin function
fir_b = signal.firwin(21, fc)

# Designing a lowpass IIR filter using the signal.butter function
iir_b, iir_a = signal.butter(10, fc)

# Plotting the frequency and impulse responses for the FIR filter
print('FIR Filter:')
plot_mag_freq_response(fir_b, [1], True)
plot_impulse_response(fir_b, [1], 60)

# Plotting the frequency and impulse responses for the IIR filter
print('\nIIR Filter:')
plot_mag_freq_response(iir_b, iir_a, True)
plot_impulse_response(iir_b, iir_a, 60)
```

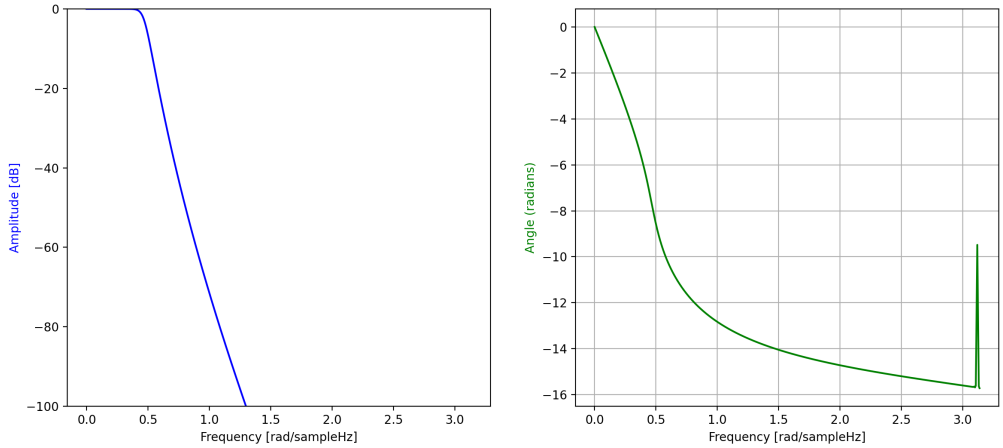
```
FIR Filter:
```

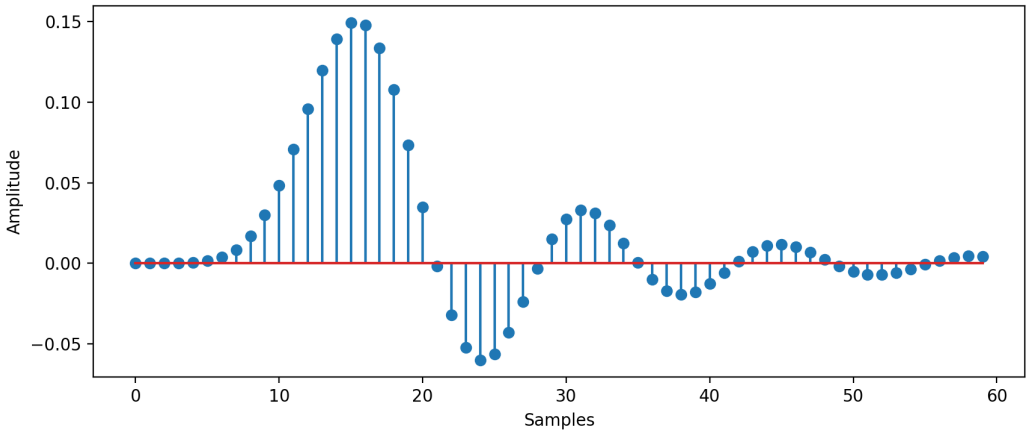



IIR Filter:

```
/var/folders/qz/cx605xfd6hg3dyw77ss7dhc80000gn/T/ipykernel_75542/1389457474.py:52: MatplotlibDeprecationWarning: The 'use_line_collection' parameter of stem() was deprecated in Matplotlib 3.6 and will be removed two minor releases later. If any parameter follows 'use_line_collection', they should be passed as keyword, not positionally.
  plt.stem(x, use_line_collection=True)
```






Out[3]:
```
array([ 1.38467082e-07,  2.35247509e-06,  1.95824370e-05,  1.06901175e-04,
        4.32302231e-04,  1.38698034e-03,  3.69159512e-03,  8.41210571e-03,
        1.67983303e-02,  2.99203978e-02,  4.81788658e-02,  7.08510585e-02,
        9.58621290e-02,  1.19918003e-01,  1.39027308e-01,  1.49312064e-01,
        1.47907733e-01,  1.33714640e-01,  1.07795024e-01,  7.33007142e-02,
        3.49368100e-02, -1.91765228e-03, -3.22305984e-02, -5.22508096e-02,
       -6.01553993e-02, -5.62855508e-02, -4.29430462e-02, -2.38170064e-02,
       -3.18417343e-03,  1.49374707e-02,  2.75074563e-02,  3.29218012e-02,
        3.11460266e-02,  2.35292445e-02,  1.23704807e-02,  3.50952513e-04,
       -1.00395160e-02, -1.69672088e-02, -1.95309716e-02, -1.78184002e-02,
       -1.27579888e-02, -5.82256704e-03,  1.33758904e-03,  7.24927370e-03,
        1.08888053e-02,  1.18282053e-02,  1.02420894e-02,  6.79209445e-03,
        2.42824665e-03, -1.84165373e-03, -5.16375896e-03, -6.99063679e-03,
       -7.15141053e-03, -5.83624446e-03, -3.50965693e-03, -7.79674801e-04,
        1.74476177e-03,  3.57763410e-03,  4.43825583e-03,  4.28101190e-03])
```

## Discussion

Comment on the differences between the two filters in terms of the magnitude, phase and impulse responses. What are the tradeoffs associated with these differences?

## Answer

The magnitude response of FIR filters usually has a sharp cutoff and precise control over the filter shape, but it might require a higher order. IIR filters, on the other hand, can have oscillations near the cutoff frequency due to their recursive nature.

Regarding phase response, FIR filters can have a linear phase, preserving waveform shape within the passband, whereas IIR filters generally introduce phase distortion due to their inherent feedback structure.

The impulse response for FIR filters is finite, settling to zero after a certain time. For IIR filters, it theoretically extends to infinity, though in practice, it becomes negligibly small after some time.

The trade-offs lie in complexity versus performance. FIR filters may provide superior performance with phase linearity and precision, but require more computational resources.

IIR filters, using fewer coefficients, might introduce phase distortion and ringing at the cutoff frequency. The choice depends on specific application requirements.

## Assignment 2 -- Different Filter Implementations for Smoothing Signals

In lab 2A, assignment 2, you experimented with smoothing a noisy signal using a moving average window and a convolution. The convolution used an impulse response h[n] that was a causal version of the moving average window. In this problem, you will implement the smoothing function using the both convolution and the signal.lfilter command, to see that they give the same result. This assignment will have three parts, A-C.

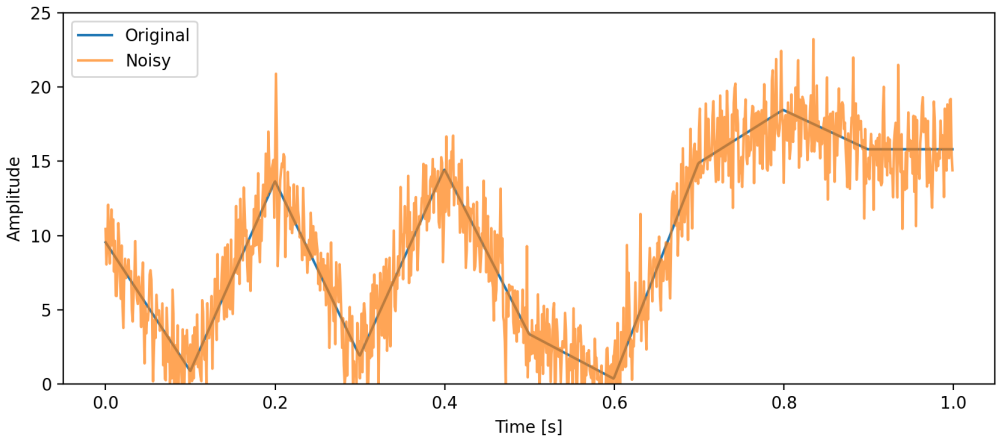
**A.** Using the code from lab 2, create a base time signal and a noisy version of it by adding random noise generated with the numpy.random.randn() function (the standard normal distribution, which is zero mean and unit variance). Plot the original and noisy signals together with the original overlaid on the noisy version, with the time axis labeled assuming a sampling rate of 1000 Hz. Constrain the y-axis to be [0,25] for all plots. Include a legend with the plot.

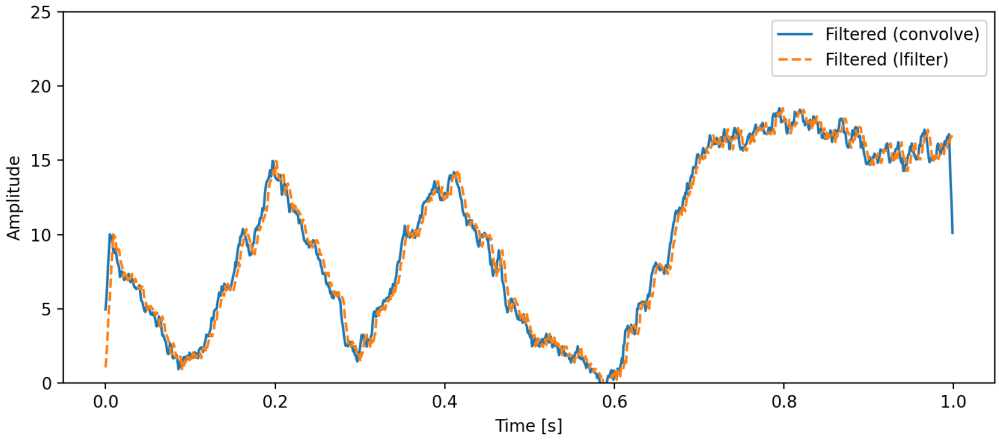
**B.** Create one smoothed version of the signal called filtsig1 by using the convolve function from lab 2 with the box impulse response and k=10. Create a second version called filtsig2 by using the signal.lfilter function. Recall that for the FIR filter, the impulse response is equal to the b coefficient vector. Plot the two filtered signals overlaid. Recall that the convolve function will change the length, so you will need to define a new time vector for that. You should find that the two methods give the same result except for edge effects.

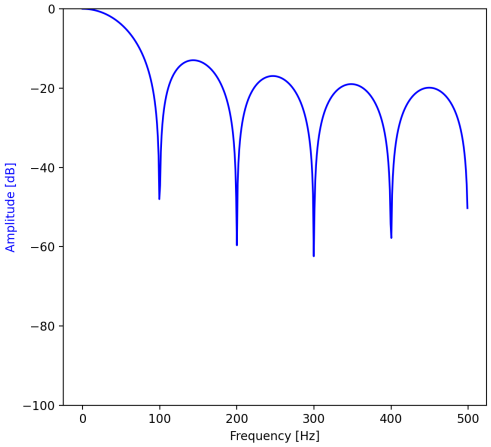
**C.** Use the function that you wrote in assignment 1 to plot the magnitude and phase for the frequency response of this filter. It should look like a low pass filter.

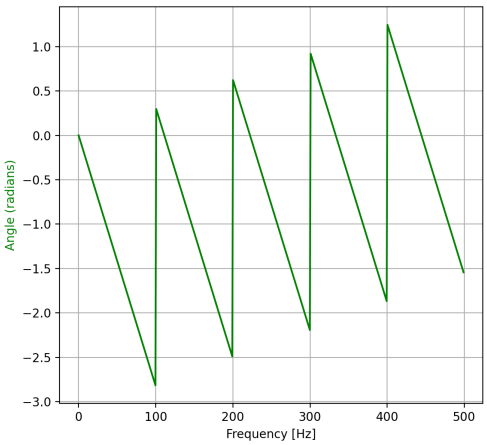
In [4]:
```
# Assignment 2 - Different Filter Implementations for Smoothing Signals

# Constants
n = 1000  # number of samples
fs = 1000  # sampling rate
k = 10  # filter order for moving average

# Part A
# Create a base time signal and a noisy version of it
t = np.arange(0, n) / fs  # time vector
p = 10  # points for piecewise linear signal
amp = 20  # amplitude range of base signal
base = np.interp(np.linspace(0, p, n), np.arange(0, p), np.random.rand(p) * amp

# Create some random noise to be added to the base signals
noiseamp = 2
noise = noiseamp * np.random.randn(n)
noisy_signal = base + noise

# Plot the original and noisy signals
plt.figure(figsize=(10, 4))
```

```
plt.plot(t, base, label='Original')
plt.plot(t, noisy_signal, label='Noisy', alpha=0.7)
plt.ylim([0, 25])
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.legend()
plt.show()

# Part B
# Create a box impulse response
h = np.ones(k) / k

# Filter the signal using convolution
filtsig1 = np.convolve(noisy_signal, h, mode='same')

# Filter the signal using lfilter
filtsig2 = signal.lfilter(h, [1.0], noisy_signal)

# Plot the filtered signals
plt.figure(figsize=(10, 4))
plt.plot(t, filtsig1, label='Filtered (convolve)')
plt.plot(t, filtsig2, label='Filtered (lfilter)', linestyle='dashed')
plt.ylim([0, 25])
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.legend()
plt.show()

# Part C
# Plot the frequency response of the filter
plot_mag_freq_response(h, [1], True, fs)
```






## Discussion

The moving window average (and its causal version) is an FIR filter, so the phase should be linear. How might the result change if you used a Butterworth filter?

## Answer

The moving average filter is a low pass FIR filter with a rectangular frequency response. While it indeed possesses a linear phase response, it lacks sharpness in the frequency cutoff. If we replaced it with a Butterworth filter, which is an IIR filter, we would see a few changes. The Butterworth filter provides a much sharper cutoff in the frequency response, reducing high-frequency noise more effectively. However, the phase response of a Butterworth filter is non-linear, which could result in phase distortions. This trade-off between sharper frequency cutoff and phase distortion is important to consider based on the specific requirements of the application.

# EE 242 Lab 4b – Digital Filtering - Applying Filters to Sound

Wheeler

This lab has 2 exercises to be completed as a team. Each should be given a separate code cell in your Notebook, followed by a markdown cell with report discussion. Your notebook should start with a markdown title and overview cell, which should be followed by an import cell that has the import statements for all assignments. For this assignment, you will need to import: numpy, the wavfile package from scipy.io, simpleaudio/librosa, and matplotlib.pyplot.

In [2]:
```
# We'll refer to this as the "import cell." Every module you import should be i
%matplotlib notebook
import numpy as np
import matplotlib
import scipy.signal as sig
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy import signal
import simpleaudio as sa
# import whatever other modules you use in this lab -- there are more that you
```

## Summary

In this lab, we will consider different types of digital filters (specifically discrete-time, linear, time-invariant filters) and look at their characterization in both time and frequency. This will give you some insight into how digital filters are implemented and into the properties of different digital filter design algorithms. You'll also learn about some of the signal processing functions available from the signal module in the scipy package which will be useful for designing and implement filters. You will work with examples that show you how filtering can be useful to remove noise and reshape the frequency content of a signal. Specifically, we'll revisit the lab 2 problem of removing noise from signals (or smoothing signals), then explore filter design methods, and finally implement a simple audio equalizer. This is a 2-week lab. It is recommended to work on the first 2 assignments in the first week and the remaining assignments in the second week.

## Lab 4b turn in checklist

• Lab 4b Jupyter notebook with code for the 2 exercises assignment in separate cells. Each assignment cell should contain markdown cells (same as lab overview cells) for the responses to lab report questions. Include your lab members' names at the top of the notebook.

**Please submit the report as PDF** (You may also use : https://www.vertopal.com/ suggested by a student)

In [3]:
```
# Helper functions
# Part A
def plot_mag_freq_response(b, a, plot_phase = False, fs = None):
    """
    This function plots the magnitude and phase of the frequency response
    of a filter given the filter coefficients a and b.
    """
    # Getting the frequency response
    w, h = signal.freqz(b, a)

    # Checking if a sampling frequency was provided
    if fs:
        # Convert from rad/sample to Hz
        w = w * fs / (2*np.pi)

    # Plotting the magnitude response
    plt.figure(figsize=(14, 6))
    plt.subplot(1, 2 if plot_phase else 1, 1)
    plt.plot(w, 20 * np.log10(abs(h)), 'b')
    plt.ylim([-100, 0])
    plt.ylabel('Amplitude [dB]', color='b')
    plt.xlabel('Frequency [{}Hz]'.format('' if fs else 'rad/sample'))

    if plot_phase:
        # Plotting the phase response on a second subplot
        plt.subplot(1, 2, 2)
        angles = np.unwrap(np.angle(h))
        plt.plot(w, angles, 'g')
        plt.ylabel('Angle (radians)', color='g')
        plt.grid()
        plt.xlabel('Frequency [{}Hz]'.format('' if fs else 'rad/sample'))

    plt.show()

# Part B
def plot_impulse_response(b, a, impulse_response_length):
    """
    This function computes and plots the impulse response of a filter given the
    filter coefficients a and b and a desired impulse response length.
    """
    # Creating the impulse signal
    impulse = np.zeros(impulse_response_length)
    impulse[0] = 1.0

    # Getting the impulse response
    x = signal.lfilter(b, a, impulse)

    # Plotting the impulse response
    plt.figure(figsize=(10, 4))
    plt.stem(x, use_line_collection=True)
    plt.ylabel('Amplitude')
    plt.xlabel('Samples')
    plt.show()

    return x
```

# Assignment 3 -- Filtering an Audio Signal

In this assignment, we'll explore different filtering problems using the horn signal that you worked with in Labs 2 and 3. You will need the audio packages that you used in previous labs. This assignment will have three parts, A-C.

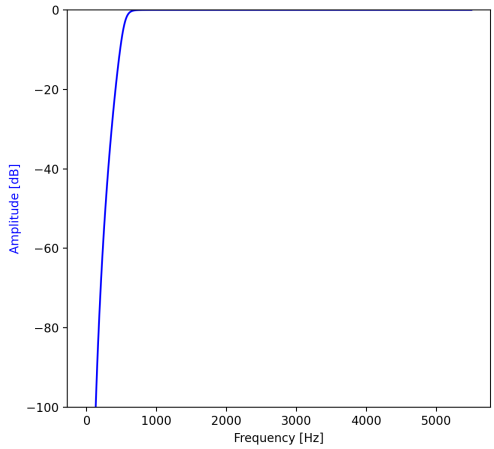
**A.** Read in the horn sound horn11short.wav. Design a highpass filter with a cutoff of 550Hz to remove the first harmonic. A Butterworth filter of order 8 should work. Plot the magnitude response and the impulse response of your filter. Using the same section of the signal as in lab 3, compute the FFT of the original and the filtered signal, and plot the log magnitude frequency content in a 2x1 plot. Play the original and the filtered waveforms.

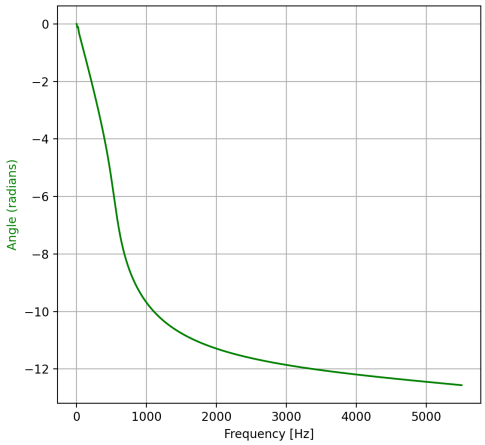
**B.** Create a noisy version of the horn sound by generating a sequence of random noise, as in lab 2, but with a scaling factor of 1000, and then adding that to the horn signal. Plot the log magnitude of the frequency content for the two signals, as in part A. Play the resulting signal and confirm that the noise is audible.

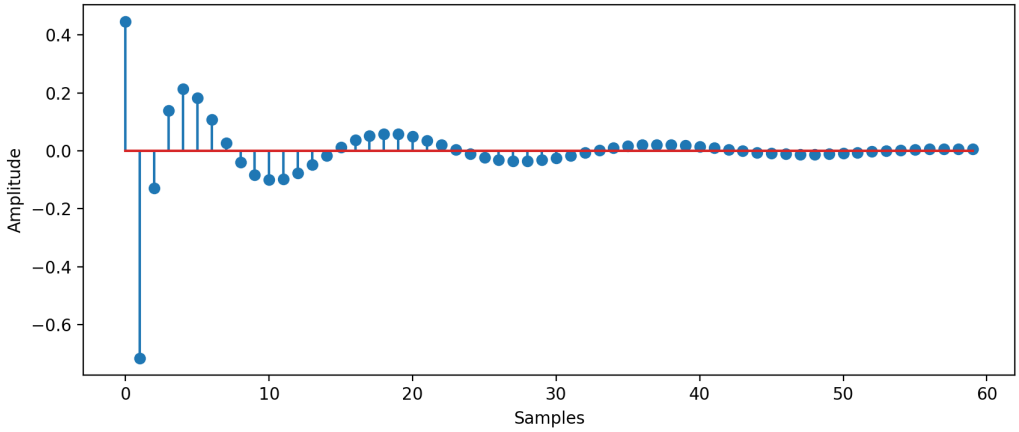
**C.** Design a lowpass filter with a cut-off corresponding to 1800Hz to remove the high frequency noise. (You can use a lower cut-off to remove more noise, but then you start removing audible harmonics.) Plot the log magnitude response and the impulse response of the filter. Plot the frequency content of the filtered signal.

In [4]:
```
# Assignment 3 - Filtering an Audio Signal


# Part A
# Read in the horn sound
fs, horn = wavfile.read('horn11short.wav')
horn = horn.astype(float)  # convert to float

# Design a highpass Butterworth filter
fc = 550  # Cut-off frequency of the filter
w_c = 2*fc/fs  # Normalize the frequency
b, a = signal.butter(8, w_c, 'high')  # Order 8

# Plot the magnitude response and the impulse response
plot_mag_freq_response(b, a, True, fs)
plot_impulse_response(b, a, 60)

# Filter the signal
horn_highpass = signal.lfilter(b, a, horn)

# Compute the FFT of the original and the filtered signal
horn_fft = np.fft.fft(horn)
horn_highpass_fft = np.fft.fft(horn_highpass)

# Plot the log magnitude frequency content
plt.figure(figsize=(10, 6), dpi = 80)
plt.subplot(2, 1, 1)
plt.plot(np.abs(horn_fft))
plt.title('Original')
plt.subplot(2, 1, 2)
```

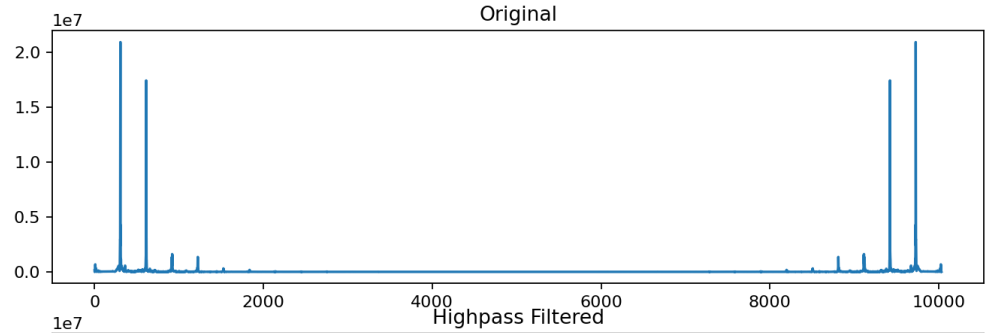
```
plt.plot(np.abs(horn_highpass_fft))
plt.title('Highpass Filtered')
plt.show()

# Play the original and the filtered waveforms
sa.play_buffer(horn.astype(np.int16), 1, 2, fs).wait_done()
sa.play_buffer(horn_highpass.astype(np.int16), 1, 2, fs).wait_done()

# Part B
# Create a noisy version of the horn sound
noise = 1000 * np.random.randn(len(horn))
horn_noisy = horn + noise

# Plot the frequency content
horn_noisy_fft = np.fft.fft(horn_noisy)
plt.figure(figsize=(10, 4), dpi = 80)
plt.plot(np.abs(horn_noisy_fft))
plt.title('Noisy Signal')
plt.show()

# Play the resulting signal
sa.play_buffer(horn_noisy.astype(np.int16), 1, 2, fs).wait_done()

# Part C
# Design a lowpass filter
fc = 1800  # Cut-off frequency
w_c = 2*fc/fs  # Normalize the frequency
b, a = signal.butter(8, w_c, 'low')  # Order 8

# Plot the magnitude response and the impulse response
plot_mag_freq_response(b, a, True, fs)
plot_impulse_response(b, a, 60)

# Filter the signal
horn_lowpass = signal.lfilter(b, a, horn_noisy)

# Plot the frequency content of the filtered signal
horn_lowpass_fft = np.fft.fft(horn_lowpass)
plt.figure(figsize=(10, 4), dpi = 80)
plt.plot(np.abs(horn_lowpass_fft))
plt.title('Lowpass Filtered')
plt.show()

# Play the resulting signal
sa.play_buffer(horn_lowpass.astype(np.int16), 1, 2, fs).wait_done()
```

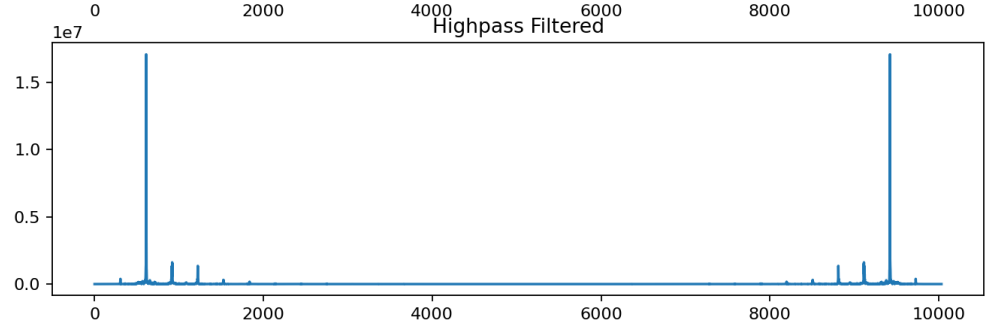


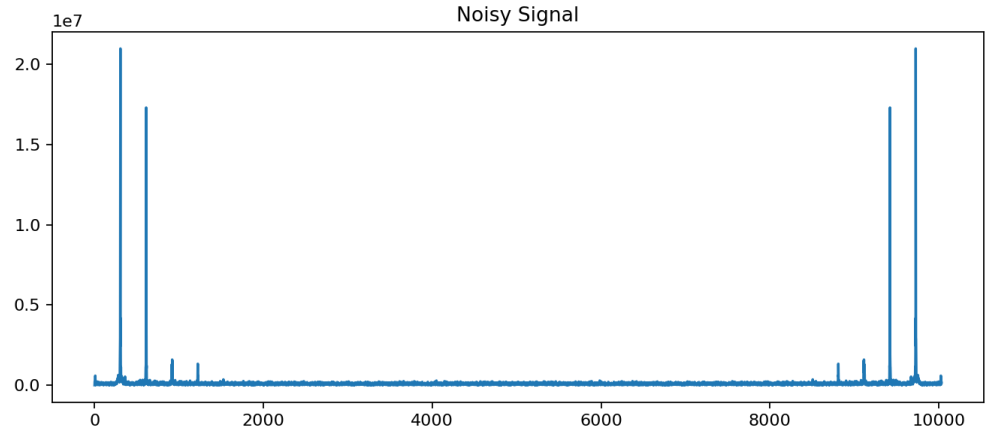



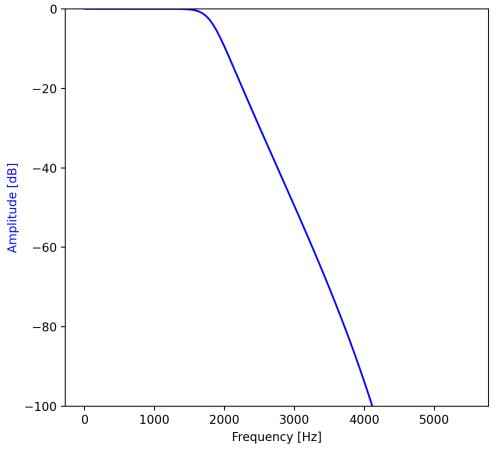



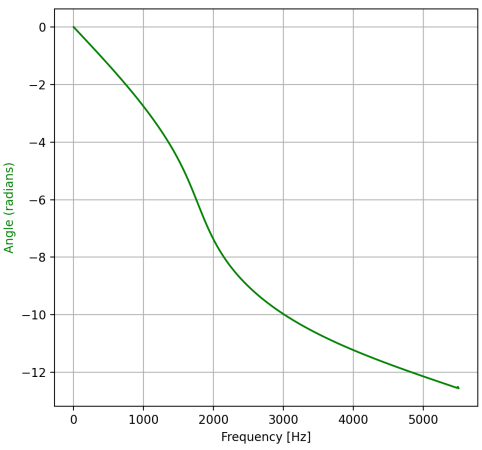

```
/var/folders/qz/cx605xfd6hg3dyw77ss7dhc80000gn/T/ipykernel_75549/3679311117.p
y:50: MatplotlibDeprecationWarning: The 'use_line_collection' parameter of ste
m() was deprecated in Matplotlib 3.6 and will be removed two minor releases la
ter. If any parameter follows 'use_line_collection', they should be passed as
keyword, not positionally.
  plt.stem(x, use_line_collection=True)
```

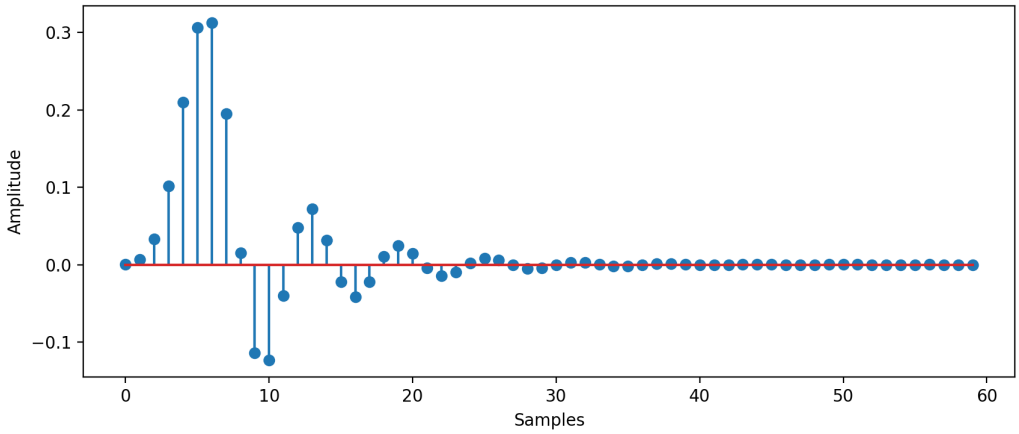

Original
1e7
2.0
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000

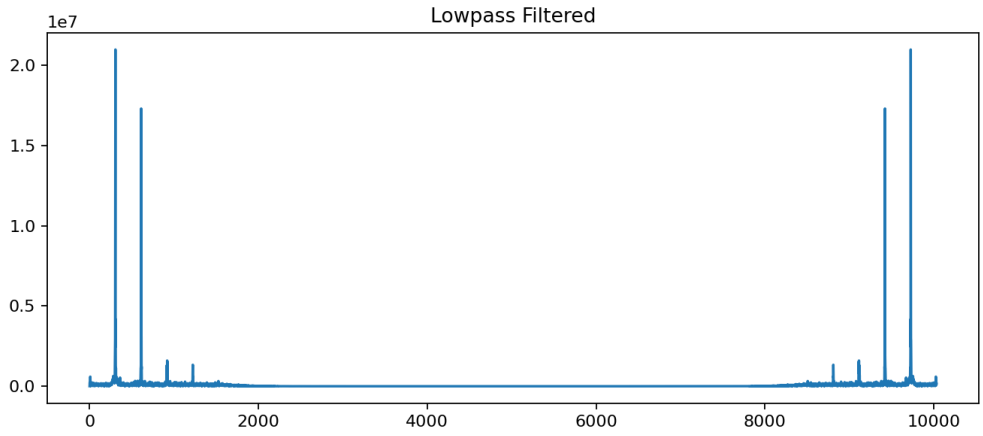

Highpass Filtered
1e7
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000


Noisy Signal
1e7
2.0
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000


0
−20
−40
−60
−80
−100
Amplitude [dB]
0
1000
2000
3000
4000
5000
Frequency [Hz]


0
−2
−4
−6
−8
−10
−12
Angle (radians)
0
1000
2000
3000
4000
5000
Frequency [Hz]




## Discussion

In part A, filtering out the first harmonic of the horn signal reduces the loudness, but it doesn't change the perceived note. Explain why this is. Discuss the differences in the impulse responses of the HPF and LPF.

## Answer

Filtering out the first harmonic of the horn signal only removes the frequency component that corresponds to the first harmonic. The perceived note is determined by the fundamental frequency of the signal, which is not affected by the highpass filter.

The impulse response of a highpass filter has a positive peak followed by a negative peak, while the impulse response of a lowpass filter has a positive peak followed by smaller positive peaks. This is because a highpass filter amplifies high frequency components and attenuates low frequency components, while a lowpass filter does the opposite.

# Assignment 4 -- Implementing a 3-Band Audio Equalizer

In this assignment, you will design a music equalizer, which breaks the sound file into multiple frequency bands by filtering, weighting, and summing to reconstruct the signal. To keep it simple, the equalizer will have only 3 bands, as illustrated in the background document: an LPF, a BFP and an HPF. This assignment will have four parts, A-D.

**A.** Design 3 5th-order Butterworth filters using the cut-off frequencies: $\pi/4$ and $\pi/2$. Plot the magnitude response of the 3 filters together in one plot

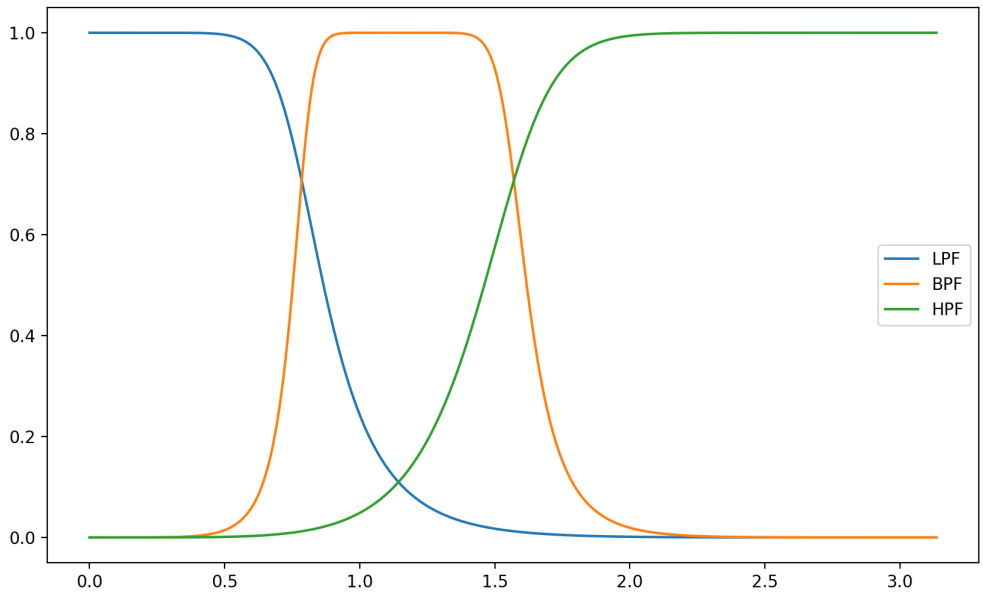
**B.** Write a function that takes as input a sound signal and three gains in dB, and outputs an equalized version of the signal. The function should use the filters that you designed to create 3 different components, then multiply each component by its respective gain (converted from dB to linear scale using your prelab result), and then sum up the re-weighted components to get the equalized result.

**C.** Read in music.wav file provided. Apply the equalizer to the music with $G1 = G2 = G3 = 0$ $dB$ and play the output. Verify that it sounds the same as the original input.

**D.** Experiment with different sets of gains (for example $G1 = G2 = 0$ $dB$ and $G3 = -40$ $dB$). Create two examples where the filtered version sounds different.

In [5]:
```
# Assignment 4 - Comparing frequency content of a signal

# Part A
# Design 3 5th-order Butterworth filters
b_lpf, a_lpf = signal.butter(5, 0.25, 'low')  # Low pass filter
b_bpf, a_bpf = signal.butter(5, [0.25, 0.5], 'band')  # Band pass filter
b_hpf, a_hpf = signal.butter(5, 0.5, 'high')  # High pass filter

# Plot the magnitude response of the 3 filters together in one plot
plt.figure(figsize=(10, 6))
for b, a, label in [(b_lpf, a_lpf, 'LPF'), (b_bpf, a_bpf, 'BPF'), (b_hpf, a_hpf
    w, h = signal.freqz(b, a)
    plt.plot(w, abs(h), label=label)
plt.legend()
plt.show()

# Part B
# Function that takes as input a sound signal and three gains in dB
def audio_equalizer(audio, G1, G2, G3):
    # Convert gains from dB to linear scale
    G1 = 10**(G1/20)
    G2 = 10**(G2/20)
    G3 = 10**(G3/20)
    # Filter the audio signal
    audio_lpf = signal.lfilter(b_lpf, a_lpf, audio)
    audio_bpf = signal.lfilter(b_bpf, a_bpf, audio)
    audio_hpf = signal.lfilter(b_hpf, a_hpf, audio)
    # Apply gains
    audio_lpf *= G1
    audio_bpf *= G2
    audio_hpf *= G3
    # Sum up the re-weighted components
    equalized_audio = audio_lpf + audio_bpf + audio_hpf
    return equalized_audio
```

```
# Part C
# Read in music.wav file
fs, music = wavfile.read('music.wav')
music = music.astype(float)  # convert to float
# Apply the equalizer with G1 = G2 = G3 = 0 dB
equalized_music = audio_equalizer(music, 0, 0, 0)
# Play the output
sa.play_buffer(equalized_music.astype(np.int16), 1, 2, fs).wait_done()

# Part D
# Experiment with different sets of gains
for G1, G2, G3 in [(0, 0, -40), (10, -10, 0)]:
    equalized_music = audio_equalizer(music, G1, G2, G3)
    sa.play_buffer(equalized_music.astype(np.int16), 1, 2, fs).wait_done()
```




## Discussion

Discuss what types of gains lead to an audible difference. Are there any constraints you need to put on the gains?

## Answer

Different sets of gains can lead to an audible difference in the filtered version of the audio signal. For example, increasing the gain of the highpass filter while decreasing the gain of the lowpass filter can result in a brighter sound with more emphasis on the higher frequencies. Similarly, increasing the gain of the lowpass filter while decreasing the gain of the highpass filter can result in a darker sound with more emphasis on the lower frequencies.

However, there are some constraints that need to be put on the gains to avoid distortion or clipping of the signal. For example, if the gain of any filter is set too high, it can cause the signal to exceed the maximum amplitude that can be represented by the bit depth of the audio file, resulting in clipping. Therefore, it is important to choose the gains carefully and ensure that they do not cause distortion or clipping of the signal.