Signals and systems
Project documentation

Denis Dzíbela
xdzibe00

January 7, 2022

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import io, fft, signal
from scipy.io import wavfile
```

0.1 Input preprocessing

0.1.1 File input

We use `scipy.io.wavfile.read()` to obtain the raw signal, and the frequency at which it was sampled, the signal has a high dynamic range and may have a DC bias, we take care of this in the following step.

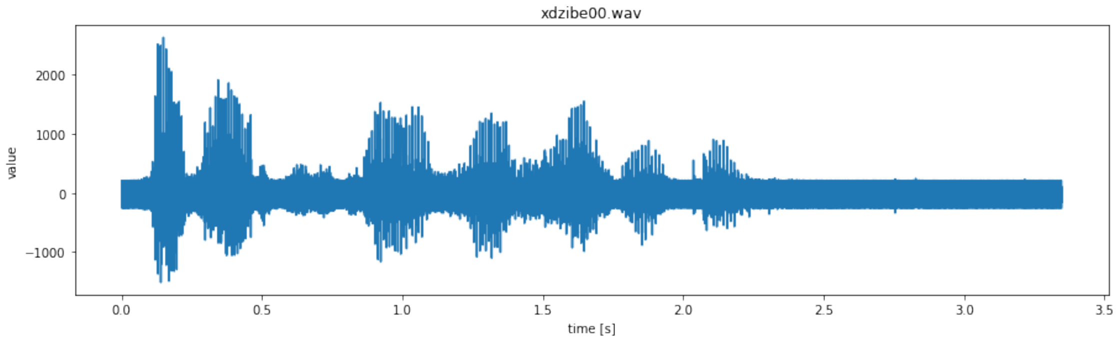
[2]:
```
samplingRate, rawData = io.wavfile.read('../audio/xdzibe00.wav')

#Get info about our signal
print('Sampling rate: ' + str(samplingRate) + 'Hz')
print('Audio length:')
print('               ' + str(rawData.size) + ' samples')
print('               ' + str(rawData.size / samplingRate) + ' seconds')
print('Max value:     ' + str(rawData.max()))
print('Min value:     ' + str(rawData.min()))

#Plotting our signal
plt.figure(1, figsize=(15,4))
plt.title('xdzibe00.wav')
plt.xlabel('time [s]')
plt.ylabel('value')
Time = np.linspace(0, len(rawData) / samplingRate, num=len(rawData))
plt.plot(Time, rawData)
plt.show()
```

```
Sampling rate: 16000Hz
Audio length:
               53556 samples
               3.34725 seconds
```

```
Max value:     2635
Min value:     -1516
```



0.1.2 Normalization and segmentation

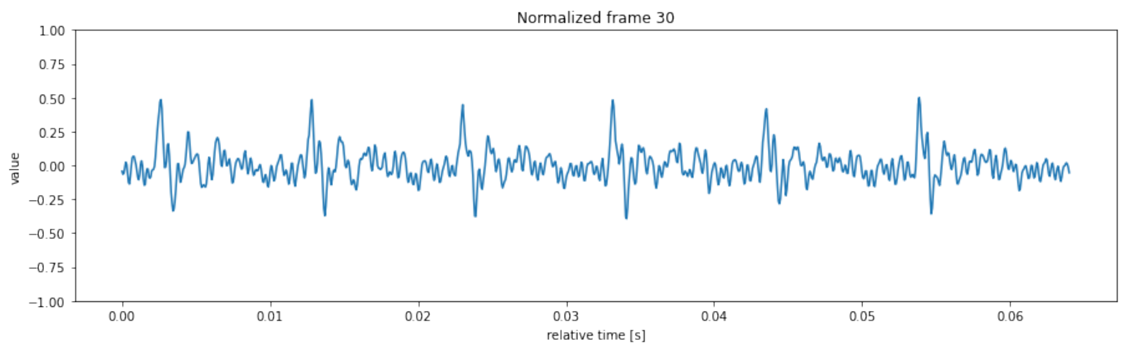
To be able to work with our signal more easily we will get rid of the DC bias and normalize the dynamic range from +1 to -1 .

For the purpose of calculating the discrete Fourier transform of our signal we split it into 104 frames, each 1024 samples long, with 512 samples overlap between frames, discarding the 105th frame as it is incomplete.

[3]:
```
#remove DC bias
rawData -= int(np.mean(rawData))
#normalize the data
normData = rawData / np.abs(rawData).max()

#slice into 1024 sample long frames, with 512 sample overlap
slicedData = np.zeros(shape=(int(normData.size / 512),1024))
for i in range(0, int(normData.size / 512) - 1): #the last frame would be␣
 ↪incomplete so we ignore it
    slicedData[i] = normData[i * 512:(i * 512) + 1024]

#Plot a selected frame
plt.figure(2, figsize=(15,4))
plt.title('Normalized frame 30')
plt.xlabel('relative time [s]')
plt.ylabel('value')
plt.ylim(-1, +1)
Time = np.linspace(0, slicedData[30].size / samplingRate, num=slicedData[30].
 ↪size)
plt.plot(Time, slicedData[30])
plt.show()
```



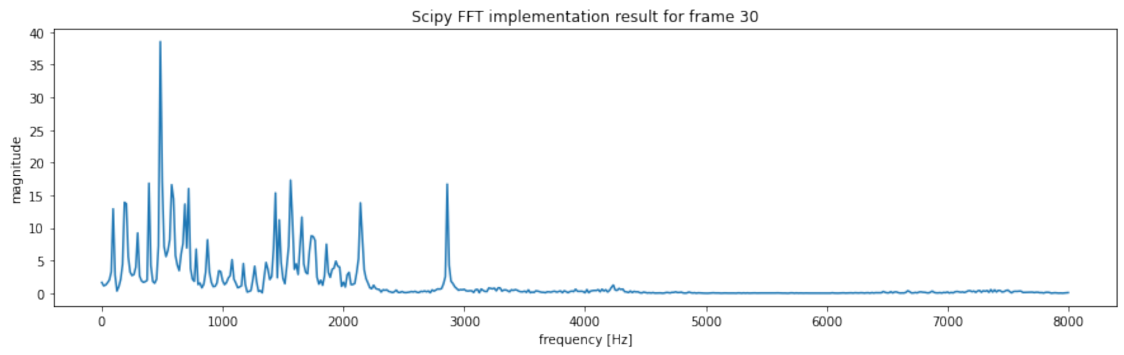
0.2 Fourier transform

0.2.1 Custom implementation

0.2.2 Comparison with a library implementation

[5]:
```python
dftLibrary = np.zeros(shape=(slicedData.shape))
for i in range(0, dftLibrary.shape[0]):
    dftLibrary[i] = np.abs(fft.fft(slicedData[i]))

#Plot the results
plt.figure(4, figsize=(15,4))
plt.title('Scipy FFT implementation result for frame 30')
plt.xlabel('frequency [Hz]')
plt.ylabel('magnitude')
#Convert DFT coeficientf to frequencies
f = np.arange(dftLibrary[30].size) / 1024 * samplingRate
#We only want to plot up to samplingRate/2
plt.plot(f[:f.size//2+1], dftLibrary[30][:dftLibrary[30].size//2+1])
plt.show()
```



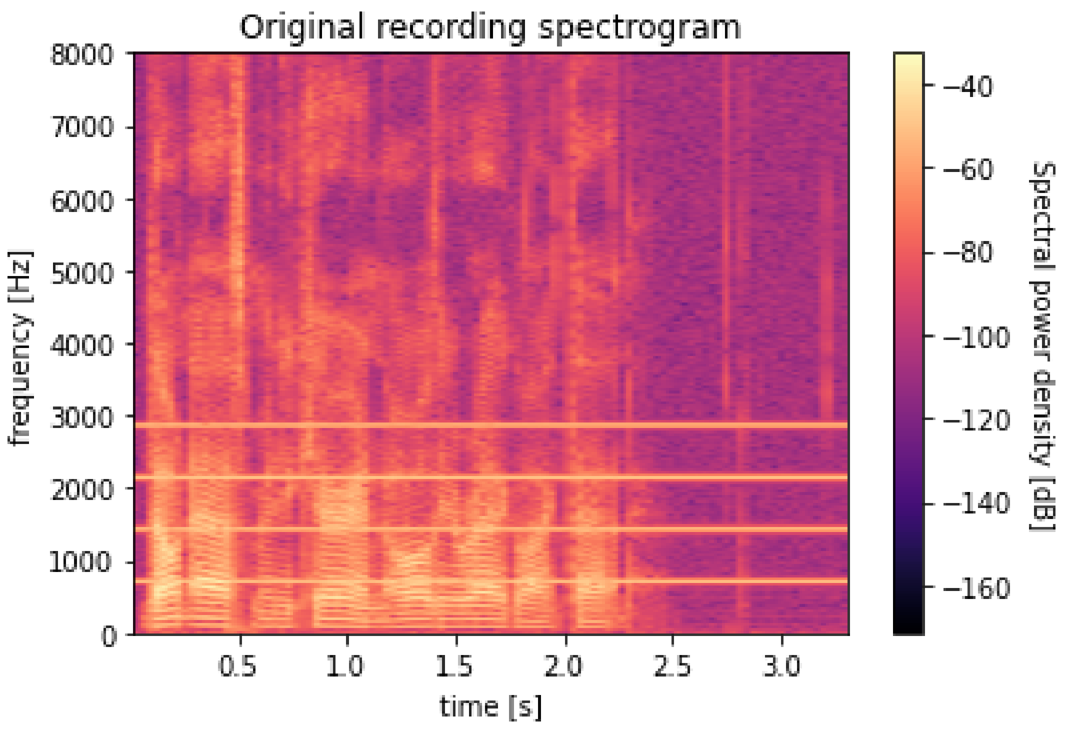
Looking at the spectrum of one frame we see isolated peaks at 2100Hz and 2800Hz, theese might be the frequencies we are looking for but we can not be sure from just one spectrum.

0.3 Spectrum analysis

0.3.1 Spectogram

We compute and show the signals power spectral density to more precisely identify the disrupting freqeucies.

[6]:
```
plt.figure(5)
plt.title('Original recording spectrogram')
plt.xlabel('time [s]')
plt.ylabel('frequency [Hz]')
plt.specgram(normData, 1024, noverlap=512, Fs=samplingRate, cmap='magma')
cbar = plt.colorbar()
cbar.set_label('Spectral power density [dB]', rotation=270, labelpad=15)
```



From the spectrogram we identify the 4 disrupting frequencies by the 4 bright lines at 700Hz, 1400Hz, 2100Hz and 2800Hz.

0.3.2 Rogue signal generation

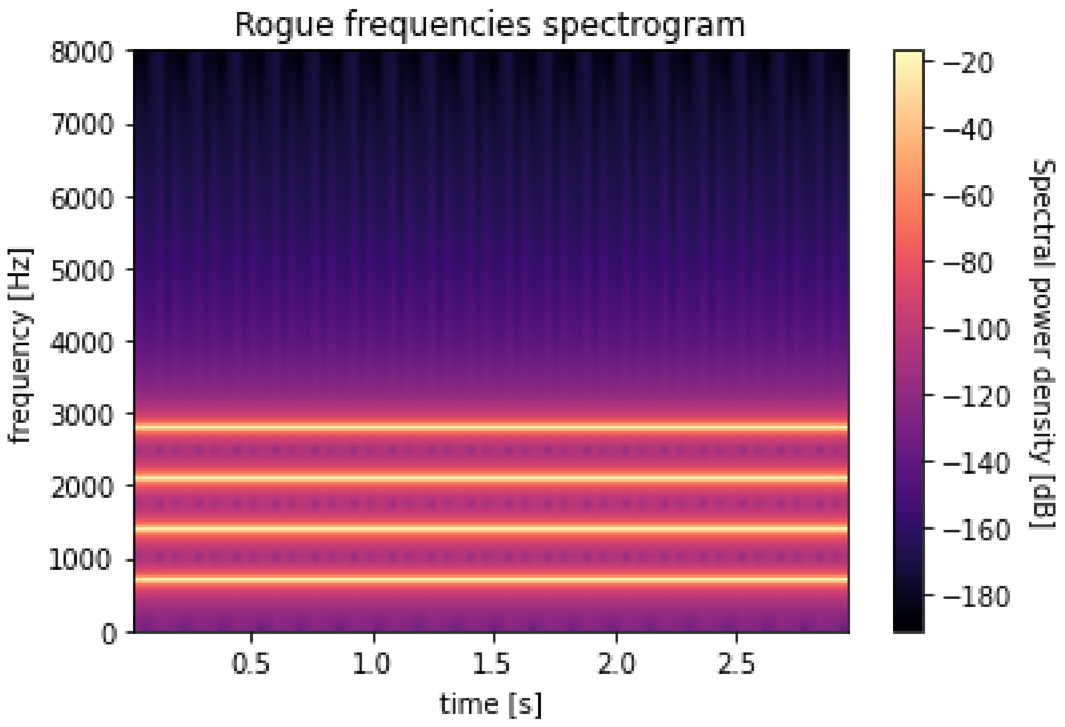
We shall confirm that we have detected the right frequencies by generating our own signal and comparing it with the original.

```
[7]: #Create a 0 valued signal 3 seconds long
rogueCosines = np.zeros(3 * samplingRate)

#Add the rogue frequencies
n = np.linspace(0, 3, 3 * samplingRate, False)
rogueCosines += np.cos(2 * np.pi * 700 * n)
rogueCosines += np.cos(2 * np.pi * 1400 * n)
rogueCosines += np.cos(2 * np.pi * 2100 * n)
rogueCosines += np.cos(2 * np.pi * 2800 * n)

#Write the result to file
io.wavfile.write('../audio/4cos.wav', samplingRate, rogueCosines)

#Plot the spectrum
plt.figure(6)
plt.title('Rogue frequencies spectrogram')
plt.xlabel('time [s]')
plt.ylabel('frequency [Hz]')
plt.specgram(rogueCosines, 1024, noverlap=512, Fs=samplingRate, cmap='magma')
cbar = plt.colorbar()
cbar.set_label('Spectral power density [dB]', rotation=270, labelpad=15)
```



By listening and visual comparison of the two signals' spectrograms we confirm that we have found the correct frequencies.

0.4 Filtration

0.4.1 Filter design

Filter design was chosen as 4 Butterworth bandstop filters centered on the 4 disrupting frequencies.

We call `scipy.signal.buttord()` to get the minimal order of our filters and then construct them using `scipy.signal.butter()`

[8]:
```
#Get Butterworth filter order
ord, wn = signal.buttord(wp=[0.1, 0.6], ws=[0.2, 0.5], gpass=3, gstop=40,␣
 ↪fs=samplingRate)

#Construct our four filters, some hand tweaking of the frequency bands was needed
sos = [0,0,0,0]
sos[0] = signal.butter(ord, [700 - 55, 700 + 55], 'bandstop', fs=samplingRate,␣
 ↪output='sos')
sos[1] = signal.butter(ord, [1400 -55, 1400 +55], 'bandstop', fs=samplingRate,␣
 ↪output='sos')
sos[2] = signal.butter(ord, [2100 -75, 2100 +70], 'bandstop', fs=samplingRate,␣
 ↪output='sos')
sos[3] = signal.butter(ord, [2800 -75, 2800 +90], 'bandstop', fs=samplingRate,␣
 ↪output='sos')
print('Filter coeficients:')
print('Filter 1')
print(signal.butter(ord, [700 - 55, 700 + 55], 'bandstop', fs=samplingRate,␣
 ↪output='ba'))
print('Filter 2')
print(signal.butter(ord, [1400 -55, 1400 +55], 'bandstop', fs=samplingRate,␣
 ↪output='ba'))
print('Filter 3')
print(signal.butter(ord, [2100 -75, 2100 +70], 'bandstop', fs=samplingRate,␣
 ↪output='ba'))
print('Filter 4')
print(signal.butter(ord, [2800 -75, 2800 +90], 'bandstop', fs=samplingRate,␣
 ↪output='ba'))
```

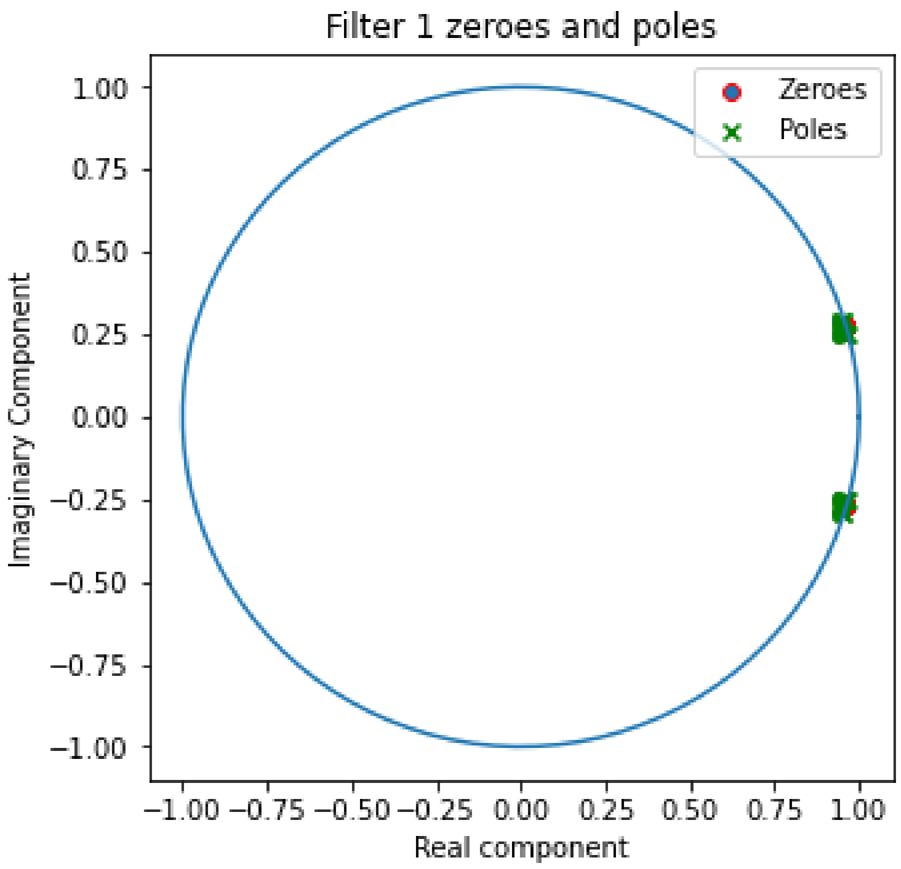
```
Filter coeficients:
Filter 1
(array([ 7.91277084e-01, -2.58993790e+01,  4.12376641e+02, -4.25475967e+03,
        3.19708105e+04, -1.86423303e+05,  8.77717822e+05, -3.42791743e+06,
        1.13219793e+07, -3.20826331e+07,  7.88536290e+07, -1.69528812e+08,
        3.20909216e+08, -5.37576214e+08,  8.00008051e+08, -1.06066733e+09,
        1.25528081e+09, -1.32762036e+09,  1.25528081e+09, -1.06066733e+09,
        8.00008051e+08, -5.37576214e+08,  3.20909216e+08, -1.69528812e+08,
        7.88536290e+07, -3.20826331e+07,  1.13219793e+07, -3.42791743e+06,
        8.77717822e+05, -1.86423303e+05,  3.19708105e+04, -4.25475967e+03,
        4.12376641e+02, -2.58993790e+01,  7.91277084e-01]), array([
1.00000000e+00, -3.22804191e+01,  5.06902595e+02, -5.15808411e+03,
        3.82254221e+04, -2.19829749e+05,  1.02077778e+06, -3.93186955e+06,
        1.28081388e+07, -3.57957181e+07,  8.67726130e+07, -1.83995396e+08,
        3.43518886e+08, -5.67565552e+08,  8.33068035e+08, -1.08937654e+09,
        1.27161319e+09, -1.32649599e+09,  1.23706760e+09, -1.03099093e+09,
        7.67000633e+08, -5.08358065e+08,  2.99324845e+08, -1.55968747e+08,
        7.15569487e+07, -2.87169796e+07,  9.99613905e+06, -2.98527260e+06,
        7.53971246e+05, -1.57960602e+05,  2.67210434e+04, -3.50774830e+03,
        3.35353992e+02, -2.07757727e+01,  6.26119423e-01]))
Filter 2
(array([ 7.91277084e-01, -2.29442886e+01,  3.26536286e+02, -3.03721623e+03,
        2.07440871e+04, -1.10825855e+05,  4.81786915e+05, -1.75049391e+06,
        5.41847824e+06, -1.44936617e+07,  3.38652612e+07, -6.96985633e+07,
        1.27172181e+08, -2.06741616e+08,  3.00594758e+08, -3.91981845e+08,
        4.59317807e+08, -4.84182554e+08,  4.59317807e+08, -3.91981845e+08,
        3.00594758e+08, -2.06741616e+08,  1.27172181e+08, -6.96985633e+07,
        3.38652612e+07, -1.44936617e+07,  5.41847824e+06, -1.75049391e+06,
        4.81786915e+05, -1.10825855e+05,  2.07440871e+04, -3.03721623e+03,
        3.26536286e+02, -2.29442886e+01,  7.91277084e-01]), array([
1.00000000e+00, -2.85972592e+01,  4.01384997e+02, -3.68202720e+03,
        2.48021388e+04, -1.30683731e+05,  5.60303483e+05, -2.00779418e+06,
        6.12955109e+06, -1.61705497e+07,  3.72648236e+07, -7.56430724e+07,
        1.36125948e+08, -2.18264437e+08,  3.13000909e+08, -4.02570684e+08,
        4.65269264e+08, -4.83746694e+08,  4.52629499e+08, -3.80994872e+08,
        2.88178170e+08, -1.95495588e+08,  1.18613304e+08, -6.41209819e+07,
        3.07304275e+07, -1.29727743e+07,  4.78382536e+06, -1.52441928e+06,
        4.13854040e+05, -9.39039978e+04,  1.73376578e+04, -2.50395832e+03,
        2.65546231e+02, -1.84052802e+01,  6.26119423e-01]))
Filter 3
(array([ 7.34460429e-01, -1.69756517e+01,  1.97125736e+02, -1.52678668e+03,
        8.84233915e+03, -4.07232471e+04,  1.54945833e+05, -4.99753040e+05,
        1.39163675e+06, -3.39131964e+06,  7.30690892e+06, -1.40294528e+07,
        2.41509954e+07, -3.74503936e+07,  5.24993694e+07, -6.67061130e+07,
        7.69608628e+07, -8.07089205e+07,  7.69608628e+07, -6.67061130e+07,
        5.24993694e+07, -3.74503936e+07,  2.41509954e+07, -1.40294528e+07,
        7.30690892e+06, -3.39131964e+06,  1.39163675e+06, -4.99753040e+05,
        1.54945833e+05, -4.07232471e+04,  8.84233915e+03, -1.52678668e+03,
        1.97125736e+02, -1.69756517e+01,  7.34460429e-01]), array([
1.00000000e+00, -2.26935722e+01,  2.58739997e+02, -1.96762989e+03,
        1.11886905e+04, -5.05946567e+04,  1.89014923e+05, -5.98590530e+05,
        1.63667763e+06, -3.91628051e+06,  8.28536728e+06, -1.56205812e+07,
        2.64043458e+07, -4.02055798e+07,  5.53451204e+07, -6.90545016e+07,
        7.82353891e+07, -8.05689469e+07,  7.54457504e+07, -6.42177461e+07,
        4.96333869e+07, -3.47706167e+07,  2.20207921e+07, -1.25627912e+07,
        6.42587264e+06, -2.92904136e+06,  1.18044620e+06, -4.16336244e+05,
        1.26777390e+05, -3.27251567e+04,  6.97891304e+03, -1.18354159e+03,
        1.50084378e+02, -1.26942471e+01,  5.39432122e-01]))
Filter 4
```

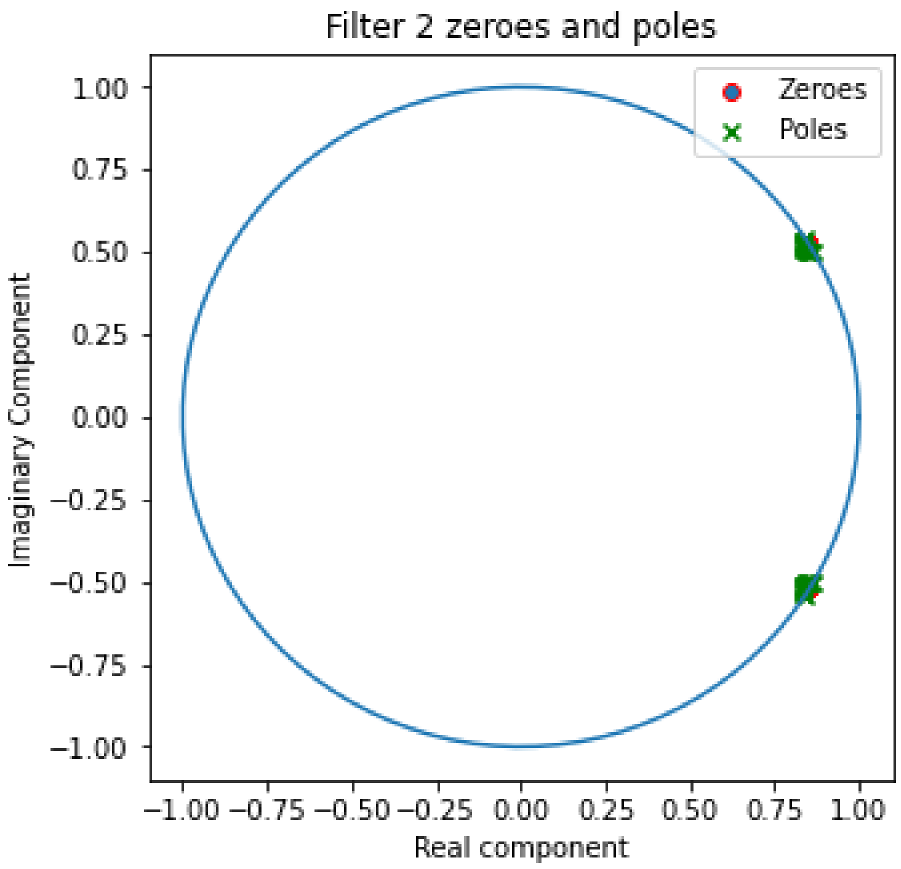
```
(array([ 7.03838838e-01, -1.08070538e+01,  9.00529305e+01, -5.25557783e+02,
        2.38182144e+03, -8.85175310e+03,  2.78989595e+04, -7.62690530e+04,
        1.83754364e+05, -3.94774573e+05,  7.63010474e+05, -1.33576686e+06,
        2.12925498e+06, -3.10295236e+06,  4.14671341e+06, -5.09319775e+06,
        5.75841543e+06, -5.99833958e+06,  5.75841543e+06, -5.09319775e+06,
        4.14671341e+06, -3.10295236e+06,  2.12925498e+06, -1.33576686e+06,
        7.63010474e+05, -3.94774573e+05,  1.83754364e+05, -7.62690530e+04,
        2.78989595e+04, -8.85175310e+03,  2.38182144e+03, -5.25557783e+02,
        9.00529305e+01, -1.08070538e+01,  7.03838838e-01]), array([
1.00000000e+00, -1.50373085e+01,  1.22710327e+02, -7.01331005e+02,
        3.11267149e+03, -1.13286849e+04,  3.49679520e+04, -9.36201453e+04,
        2.20904608e+05, -4.64804466e+05,  8.79860467e+05, -1.50863642e+06,
        2.35537917e+06, -3.36198880e+06,  4.40070270e+06, -5.29437049e+06,
        5.86329217e+06, -5.98264633e+06,  5.62597752e+06, -4.87446784e+06,
        3.88768676e+06, -2.84984838e+06,  1.91576596e+06, -1.17739499e+06,
        6.58881319e+05, -3.33979090e+05,  1.52303242e+05, -6.19339613e+04,
        2.21965113e+04, -6.89999585e+03,  1.81910311e+03, -3.93279914e+02,
        6.60259639e+01, -7.76352952e+00,  4.95389109e-01]))
```

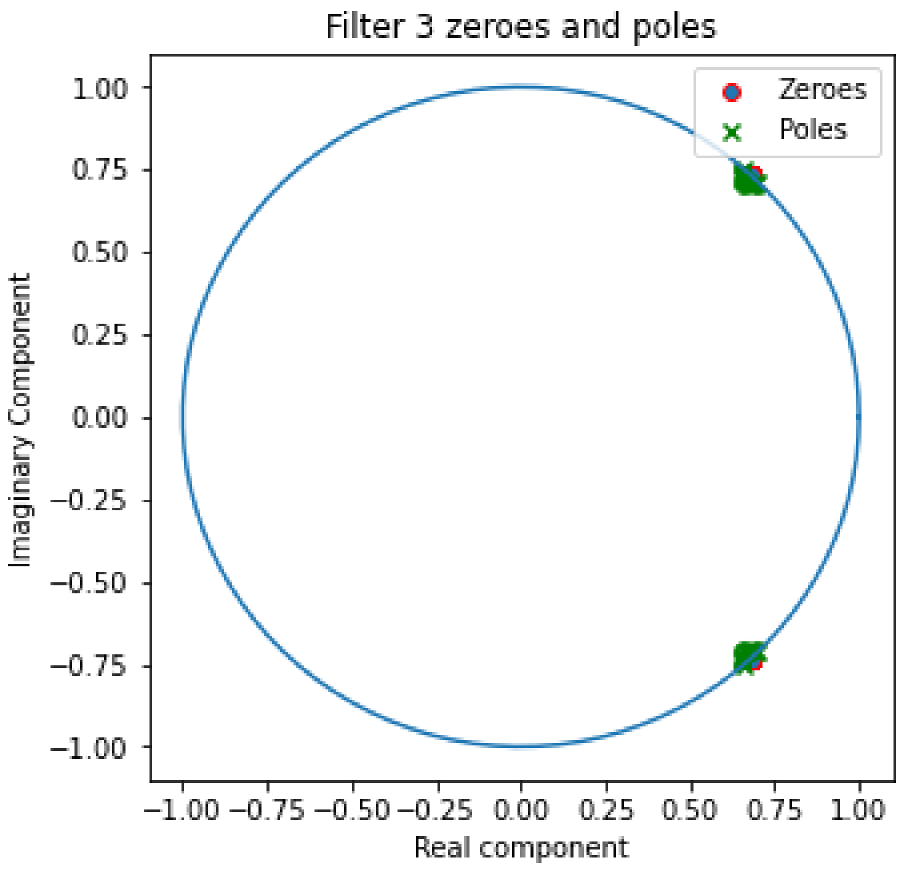
0.4.2 Zeroes and poles

We can use `scipy.signal.sos2zpk()` to calculate the zeroes and poles of our 4 filters

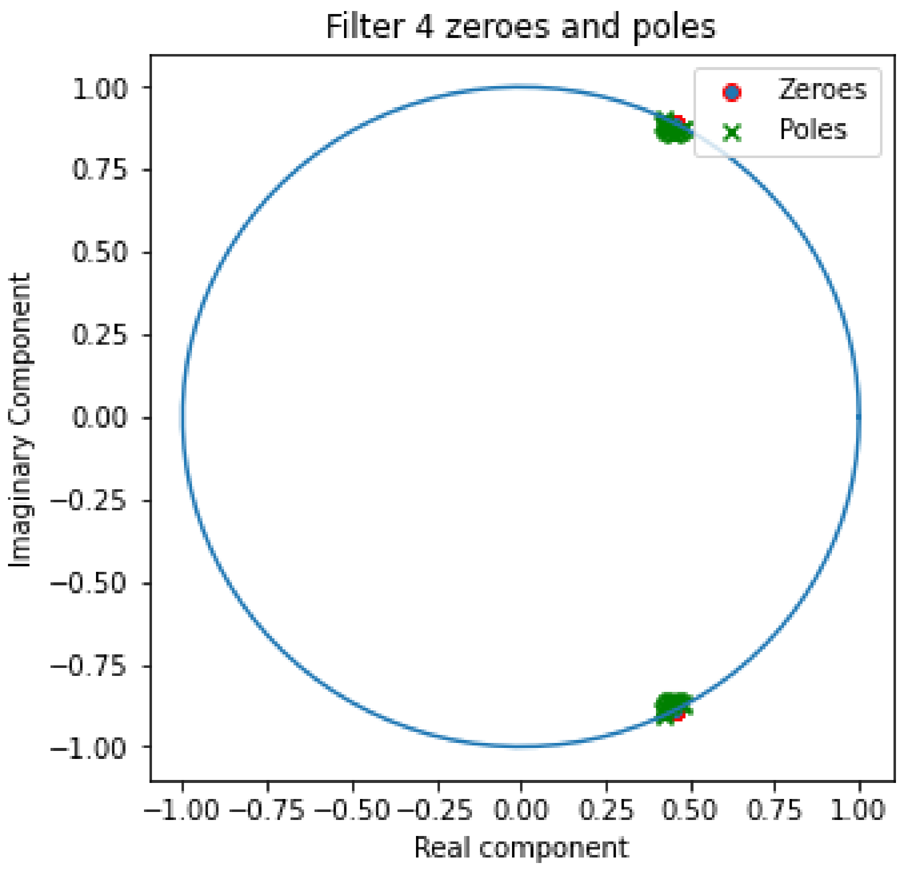
```
[9]: for i in range(0, 4):
         plt.figure(8 + i, figsize=(5,5))
         z, p, k = signal.sos2zpk(sos[i])
         plt.title('Filter ' + str(i + 1) + ' zeroes and poles')
         #Unit circle
         ang = np.linspace(0, 2*np.pi,100)
         plt.plot(np.cos(ang), np.sin(ang))
         #Zeroes
         plt.scatter(np.real(z), np.imag(z), marker='o', edgecolors='r',␣
      ↪label='Zeroes')
         #Poles
         plt.scatter(np.real(p), np.imag(p), marker='x', facecolor='g', label='Poles')
         #Axis labels
         plt.xlabel('Real component')
         plt.ylabel('Imaginary Component')
         plt.legend(loc='upper right')
         plt.show()
```


Filter 1 zeroes and poles
Zeroes
Poles
Imaginary Component
Real component
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00


Filter 2 zeroes and poles
Zeroes
Poles
Imaginary Component
Real component
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



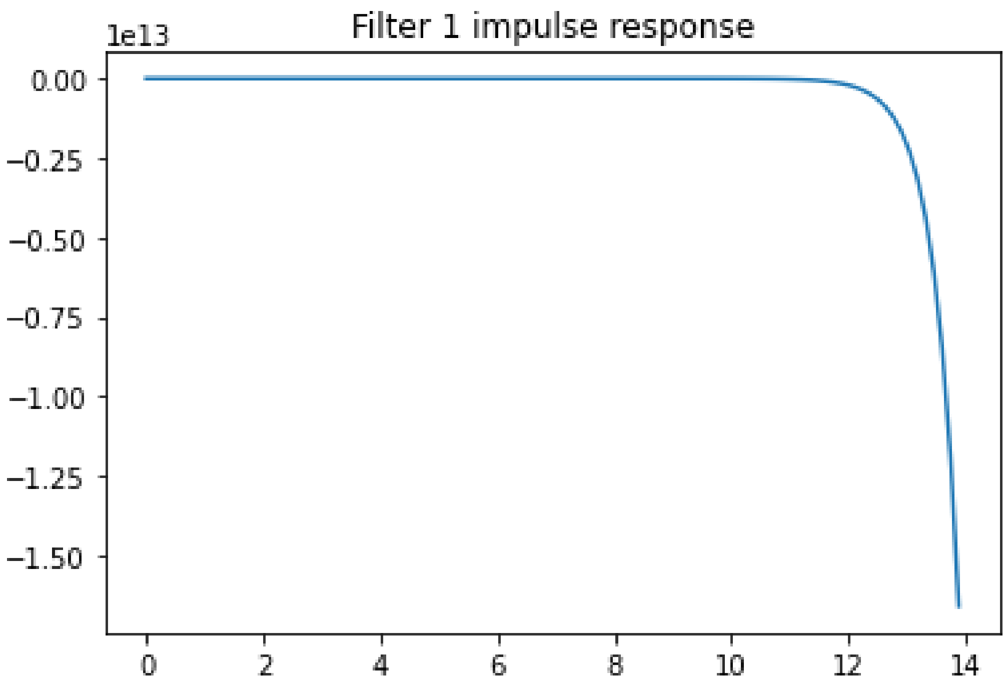
Filter 3 zeroes and poles
Zeroes
Poles
Imaginary Component
Real component
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

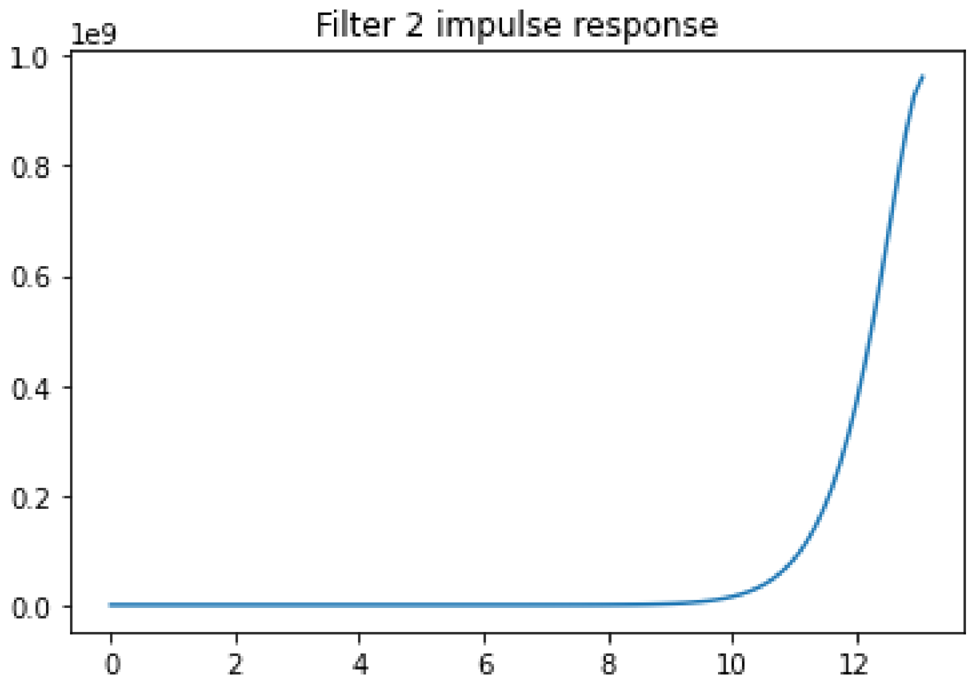


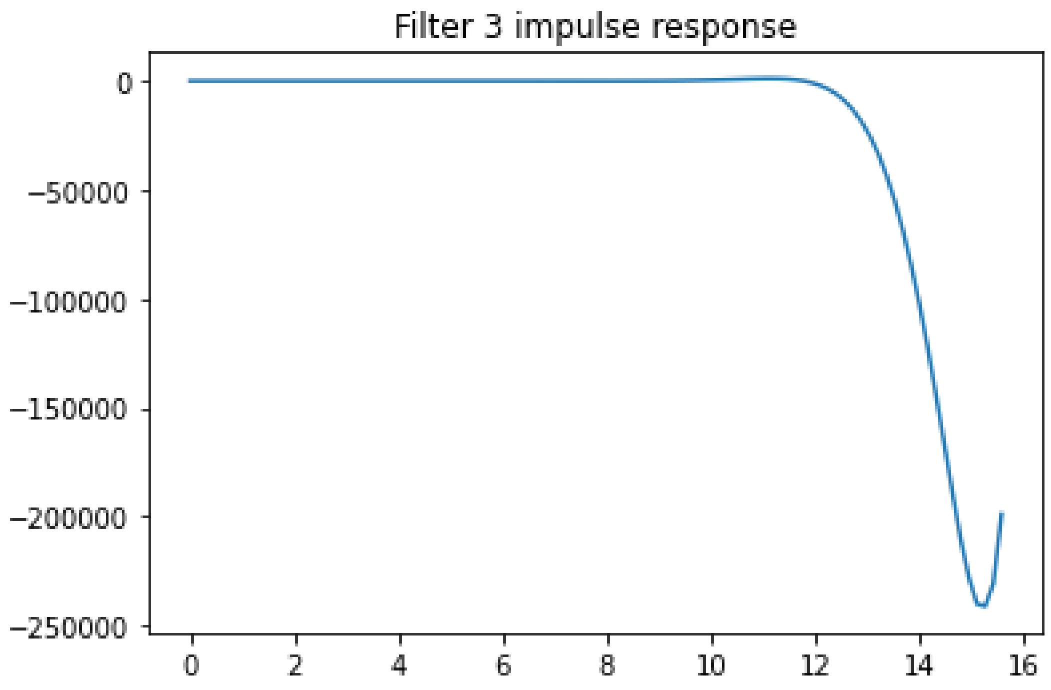
0.4.3 Impulse response

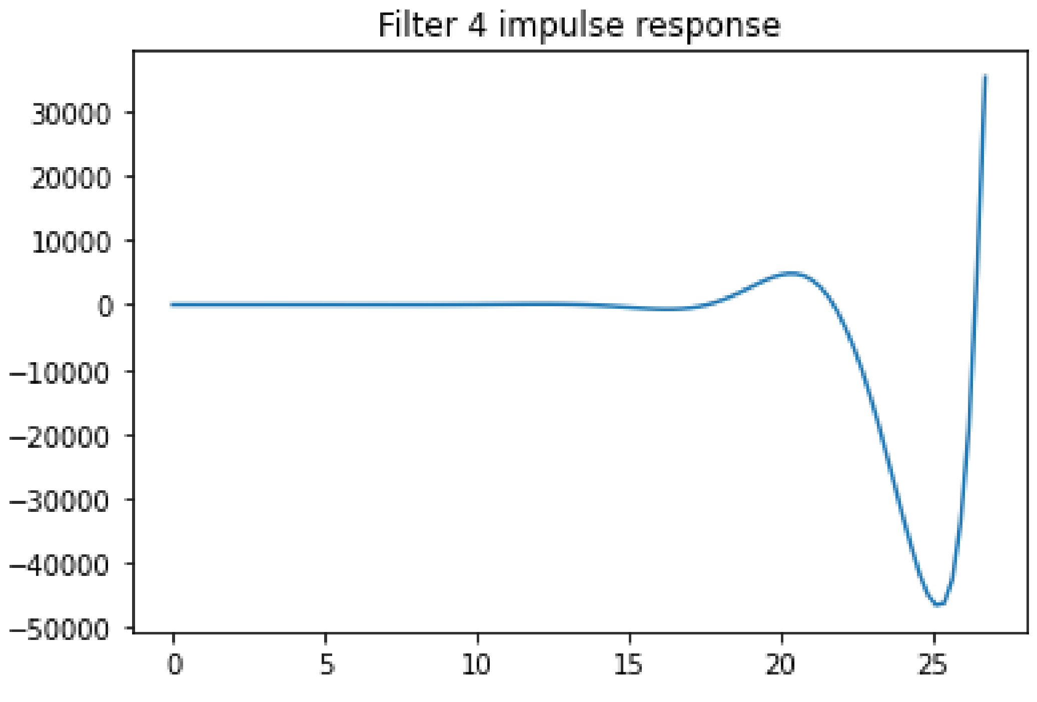
We can use the zeroes and poles with `scipy.signal.impulse()` to calculate the impulse response of the filters.

[10]:
```
for i in range(0,4):
    plt.figure(11 + i)
    plt.title('Filter ' + str(i + 1) + ' impulse response')
    z, p, k = signal.sos2zpk(sos[i])
    t, yout = signal.impulse((z,p,k))
    plt.plot(t, yout)
    plt.show()
```


Filter 1 impulse response
1e13
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
0
2
4
6
8
10
12
14


Filter 2 impulse response
1e9
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12


Filter 3 impulse response
0
−50000
−100000
−150000
−200000
−250000
0
2
4
6
8
10
12
14
16

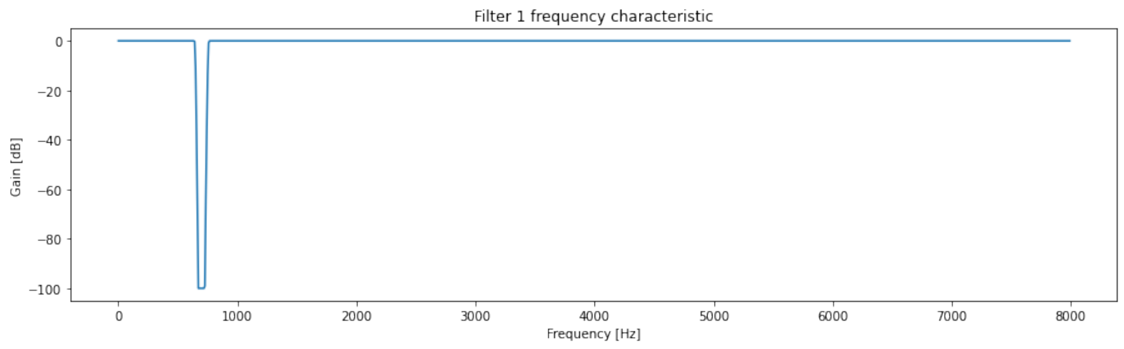

Filter 4 impulse response
30000
20000
10000
0
−10000
−20000
−30000
−40000
−50000
0
5
10
15
20
25

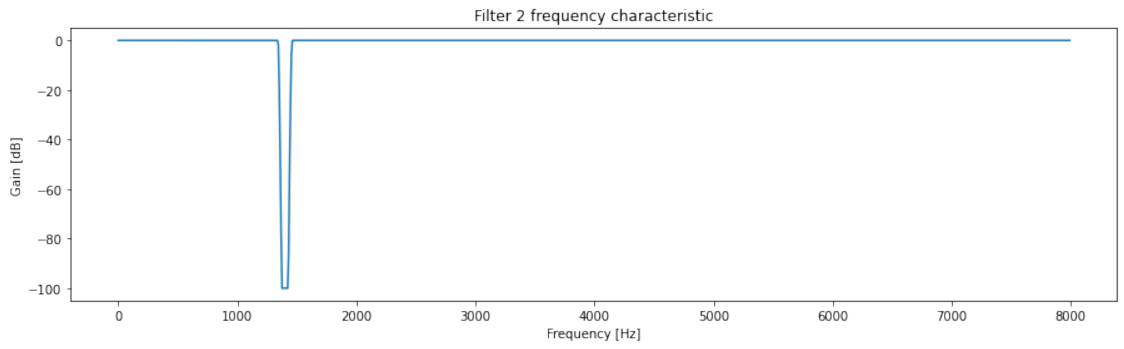
0.4.4 Frequency characteristic

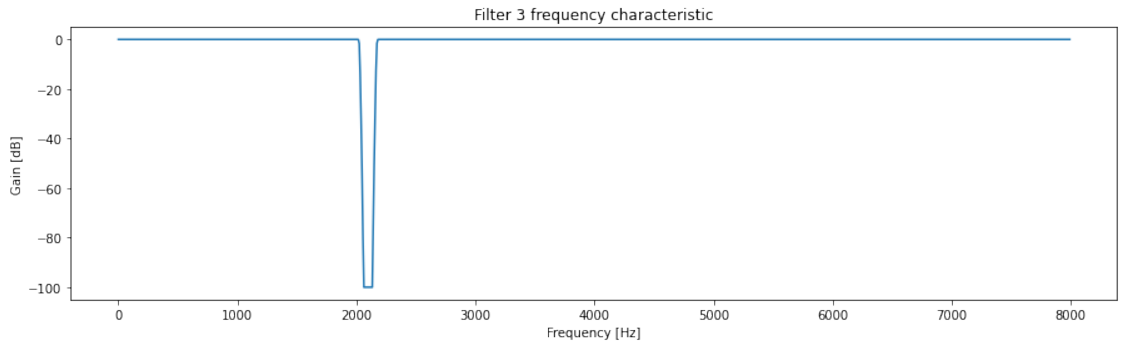
Likewise we call `scipy.signal.sosfreqz()` to get the frequency response of our filters

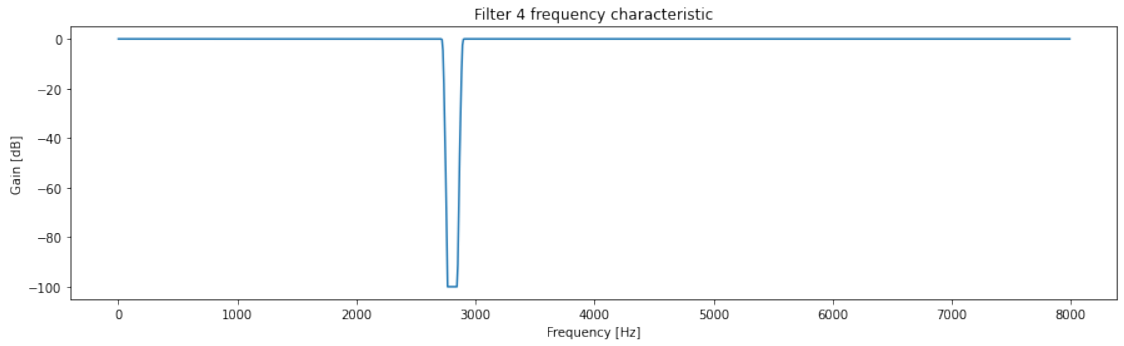
[11]:
```
for i in range(0, 4):
    plt.figure(14 + i, figsize=(15,4))
    w, h = signal.sosfreqz(sos[i], worN=1024, fs=samplingRate)
    plt.title('Filter ' + str(i + 1) + ' frequency characteristic')

    db = 20*np.log10(np.maximum(np.abs(h), 1e-5))
    plt.plot(w, db)
    #Axis labels
    plt.xlabel('Frequency [Hz]')
    plt.ylabel('Gain [dB]')
    plt.show()
```









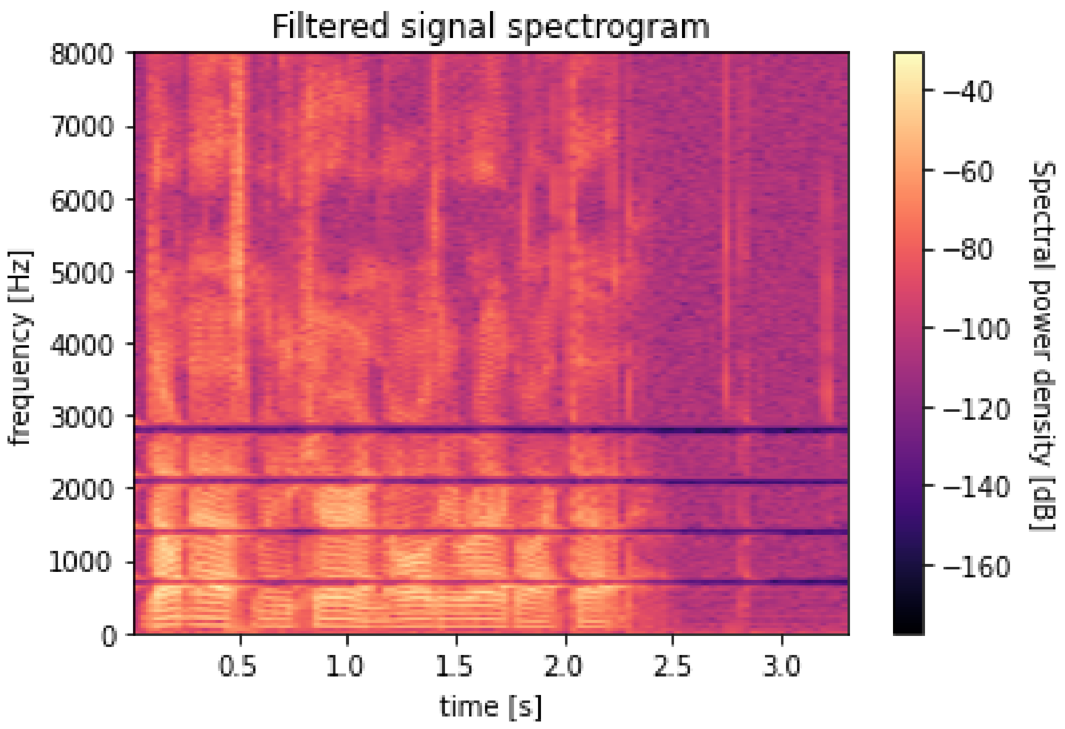
0.4.5 Filtration

Lastly we apply the four filters to the normalized signal, the spectrogram shows that the filters had blocked only little of the actual signal, leaving us with a clean recording.

[12]:
```
#Apply our four filters
filtered = signal.sosfilt(sos[0], normData)
for i in range(1,4):
    filtered = signal.sosfilt(sos[i], filtered)

#Save the result to file
io.wavfile.write('../audio/clean_bandstop.wav', samplingRate, filtered)

#Plot a spectrogram for good measure
plt.figure(666)
plt.title('Filtered signal spectrogram')
plt.xlabel('time [s]')
plt.ylabel('frequency [Hz]')
plt.specgram(filtered, 1024, noverlap=512, Fs=samplingRate, cmap='magma')
cbar = plt.colorbar()
cbar.set_label('Spectral power density [dB]', rotation=270, labelpad=15)
```


Filtered signal spectrogram
frequency [Hz]
8000
7000
6000
5000
4000
3000
2000
1000
0
0.5
1.0
1.5
2.0
2.5
3.0
time [s]
−40
−60
−80
−100
−120
−140
−160
Spectral power density [dB]