exercise-2022

April 27, 2023

1 Lab Session 6: Signal Averaging and Confidence Intervals

1.1 Auditory Computation, Modelling and Devices (E092970A)

1.1.1 Dept of Information Technology (UGent) and Dept of Electronics and Informatics (VUB)

Sarah Verhulst

Students names and IDs: Cesar Zapata - 02213600 Academic Year : 2021-2022

[1]:
```
#run this first before you start
import os
import mat73
import numpy as np
import scipy as sci
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib import colors as colors
%matplotlib inline
from scipy import stats
import statsmodels.api as sm
import scipy.io
import scipy.signal as sig
import statsmodels
```

2 Part 1: Auditory Brainstem Response (ABR)

2.0.1 Auditory EEG for hearing diagnostics

Here, we will make use of the event-related potentials (EEG) to a sensory event. We use multiple repetitions to the same event to improve the signal-to-noise ratio of the data. We will use this information to perform bootstrapping-based hypothesis testing. In the first part, you will process your own recorded auditory brainstem responses (ABRs), which were recorded to 3000 repetitions of an acoustic click. The resulting signal has a very typical waveform with distinct peaks which occur within the first 10 ms after the click onset, and which resemble different processing stages along the auditory pathway. The peak at 1-2 ms (wave-I) reflects the ensemble of auditory-nerve fibers available, whereas the peak at 5-7 ms (wave-V) reflects the brainstem processing of sound (inferior colliculus). The ABR is hence some sort of impulse response of the ear, and is often

clinically used to detect peripheral brain lesions, or as a screening tool for neonates. The signal also has information about how the ear processes sound (i.e. for research purposes).

ABR stimuli were clicks of 80 and 100 dB SPL, which were repeated 3000 times. The clicks were presented in opposite polarity, which means that first a positive click was presented, then a negative one, and so forth. Four files were extracted from each of your recordings and are included in the EEG_data folder, e.g. for an ABR recorded with an 80 dB SPL click:

1. ABR_80_Cz: recorded signal from the electrode placed on the top of head (i.e. the B16 electrode as you saw in the lab session)

2. ABR_80_ref1: recorded signal from the left earlobe electrode (i.e. the external electrode that you placed on the left ear)
3. ABR_80_ref2: recorded signal from the right earlobe electrode (i.e. the external electrode that you placed on the right ear)
4. TrigsABR_80: trigger signal which indicates the onset of each epoch by "1". Odd epochs correspond to positive polarity stimuli and even epochs to negative polarity.

The same definition holds for the rest of the recorded conditions (ABR_100, SAM_EFR, david), recorded using the different stimuli. The files containing "group_stimulus" in their name correspond to the recordings made with each group's pre-designed stimuli.

Run the following code to extract your data from the recording.

[77]:
```
#Load in the data
ABRData80 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"ABR_80_Cz.mat"))
ABR80_ref1 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"ABR_80_ref1.mat"))
ABR80_ref2 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"ABR_80_ref2.mat"))
ABR80_trig = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"TrigsABR_80.mat"))

#here we do all the analysis for the 80dB click stimulus, you should do the␣
 ↪same for the 100 dB stimulus as well as
#for the ABR signals that you recorded using your programmed stimuli
#Sampling frequency
FS = 16384
Sig_80 = ABRData80['Ch'] #has the raw recording trace of EEG data
ref1_80 = ABR80_ref1['Ch']
ref2_80 = ABR80_ref2['Ch']
```

```
T_80 = ABR80_trig['Trigs'] #is the channel with equal duration of the␣
 ↪recording, but which has the trigger events
#(i.e. sample numbers at which a click was presented)
#find all the triggers
indx_80 = np.where(np.diff(T_80)==1)[1]+1 #index has the sample numbers at␣
 ↪which an event started
#We will just average both polarities together, but those interested could also␣
 ↪repeat the analysis
#for only the positive or negative polarity click stimuli by separating the odd␣
 ↪and even epochs (trigger indexes).

avg_ref1_ref2 = [(abs(ref1_80[0][i] - ref2_80[0][i]) / 2) for i in␣
 ↪range(len(ref1_80[0]))]
re_reference = [(Sig_80[0][i] - avg_ref1_ref2[i]) for i in␣
 ↪range(len(ref1_80[0]))]
```

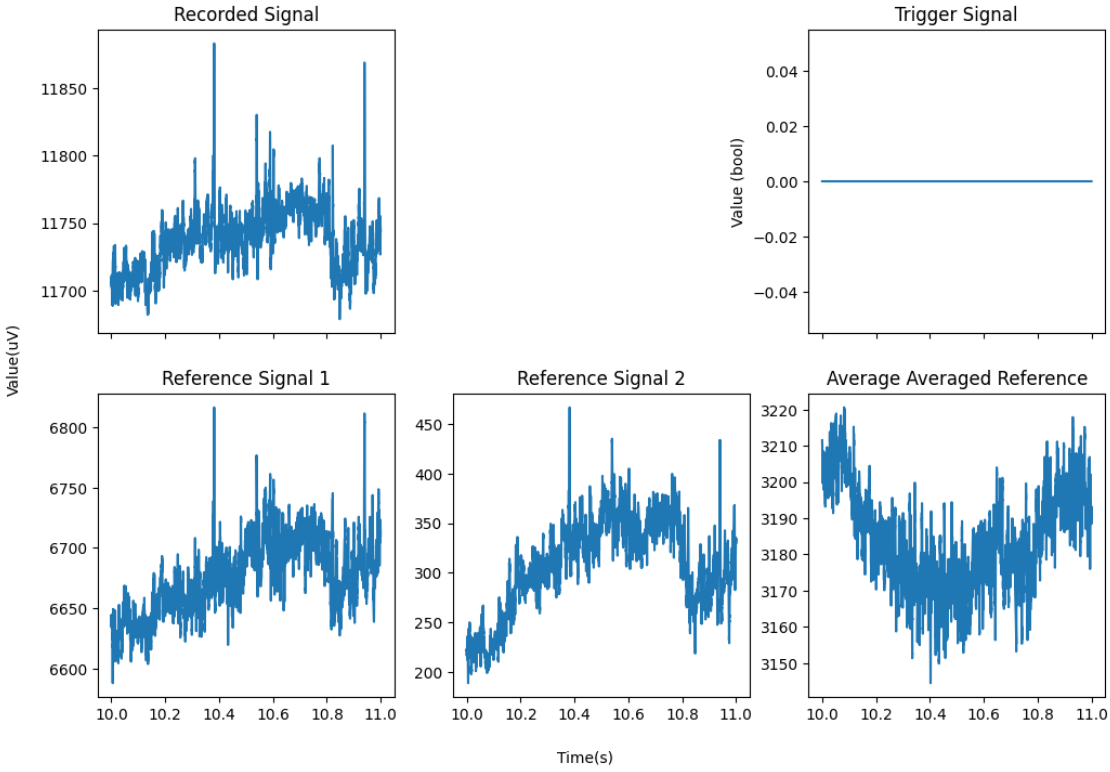
[78]:
```
print(ABRData80.keys())
```

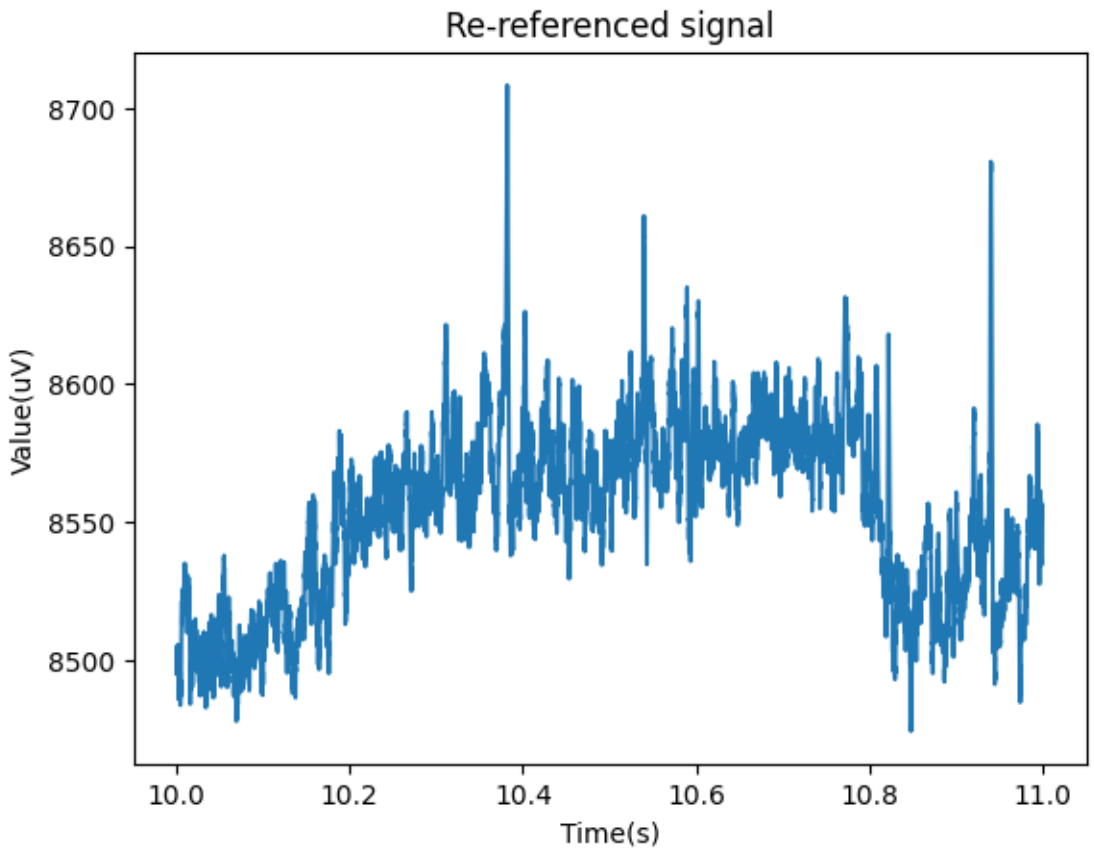
```
dict_keys(['__header__', '__version__', '__globals__', 'Ch'])
```

- Plot the 10-th to 11-th second (one second duration) of the Sig_80 (i.e. the main signal), reference channels (ref1_80 and ref2_80), average of the ref1_80 and ref2_80, and triggers (T_80) in separate figures using plt.subplot to get a feeling of your data. Make titles for each figure, i.e., 'Recorded signal', 'Reference Signal1', 'Reference Signal2', 'Averaged Reference' and 'Trigger signal'. First, define a time vector with a duration of one second and then plot the above-mentioned signals. Assign proper labels for the x and y axes.
- Re-reference the main signal (Sig_80) to the average of the reference channels (ref1_80 and ref2_80) and plot the re-referenced channel in a separate figure (the same time interval, i.e. the 10-th to 11-th second). To apply the re-referencing, subtract the averaged reference channel from the main signal (Sig_80).

[79]:
```
seconds = [10, 11] # trigger starts at 33s
time_window = np.arange(seconds[0], seconds[1], 1/FS) # time array for the␣
 ↪desired seconds

# plotting
figs, axs = plt.subplots(2, 3, figsize=(12, 8), sharex=True)

figs.text(0.5, 0.04, 'Time(s)', ha='center', va='center')
figs.text(0.06, 0.5, 'Value(uV)', ha='center', va='center', rotation='vertical')

axs[0][0].plot(time_window, Sig_80[0][seconds[0] * FS: seconds[1] * FS])
axs[0][0].set_title("Recorded Signal")

axs[0][2].plot(time_window, T_80[0][seconds[0] * FS: seconds[1] * FS])
axs[0][2].set_ylabel("Value (bool)")
axs[0][2].set_title("Trigger Signal")
```

```
axs[0][1].axis('off') # leave a blank space in the subplot

axs[1][0].plot(time_window, ref1_80[0][seconds[0] * FS: seconds[1] * FS])
axs[1][0].set_title("Reference Signal 1")

axs[1][1].plot(time_window, ref2_80[0][seconds[0] * FS: seconds[1] * FS])
axs[1][1].set_title("Reference Signal 2")

axs[1][2].plot(time_window, avg_ref1_ref2[seconds[0] * FS: seconds[1] * FS])
axs[1][2].set_title("Average Averaged Reference")

plt.show()

# plotting the re-referenced values
plt.figure(1)
plt.plot(time_window, re_reference[seconds[0] * FS: seconds[1] * FS])
plt.title("Re-referenced signal")
plt.xlabel("Time(s)")
plt.ylabel("Value(uV)")
plt.show()
```



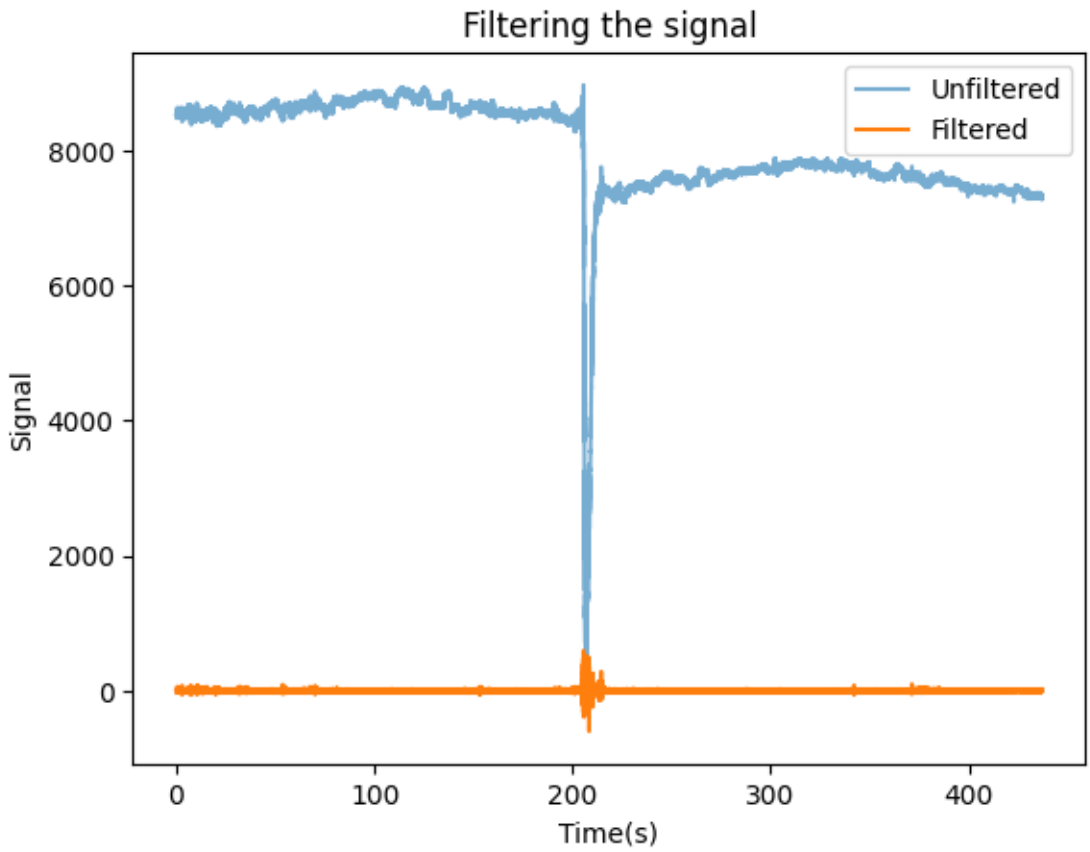


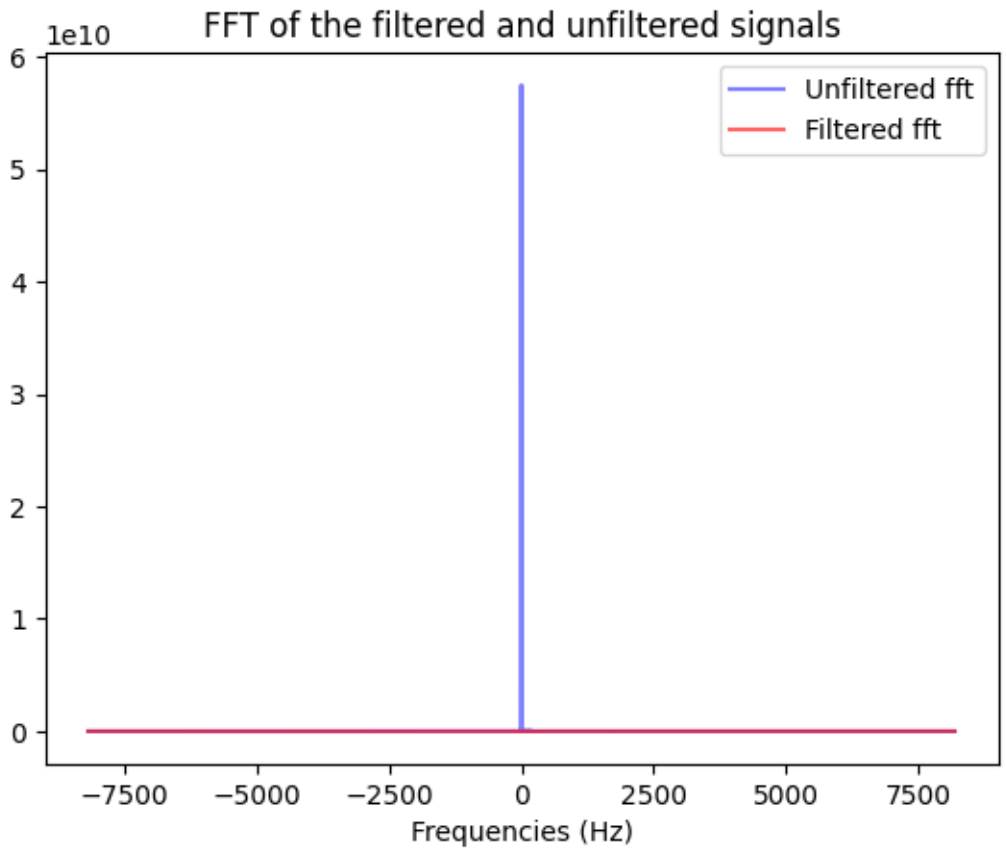
Apply a 4th order butterworth filter (sig.butter) to your data (re-referenced signal) with cut-off frequencies between 100 and 1500 Hz. Use the sig.filtfilt function to design your filters, apply first the HP filter and then afterwards the LP filter. Then, visualize the signal spectrum as well as the time-domain signal before and after filtering (use np.fft.fft).

[80]:
```
print(f"time: {len(re_reference) / FS}")
time = np.linspace(0, len(re_reference) / FS, len(re_reference))

# function for the butter filter, highpass and lowpass
def filtering(signal, order, FS, lowcut, highcut):

    F_highpass = highcut / (FS/2)
    F_lowpass = lowcut / (FS/2)

    HP_a, HP_b = sig.butter(order, F_highpass, btype='high')
    LP_a, LP_b = sig.butter(order, F_lowpass, btype='low')

    Filtered_sig = sig.filtfilt(HP_a, HP_b, signal)
    Filtered_sig = sig.filtfilt(LP_a, LP_b, Filtered_sig)
```

```
    return Filtered_sig

# function for the fft of the signal
def fft_sig(signal, filtered_signal, FS):

    freq_sig = np.fft.fftfreq(len(signal), 1/FS)
    freq_sig = np.fft.fftshift(freq_sig)

    fft_sig = np.abs(np.fft.fft(filtered_signal))
    fft_sig = np.fft.fftshift(fft_sig)

    return freq_sig, fft_sig


Filtered_sig_80 = filtering(re_reference, 4, FS, 1500, 100)

freq_org, fft_org = fft_sig(re_reference, re_reference, FS)
freq_filt, fft_filt = fft_sig(re_reference, Filtered_sig_80, FS)


# plotting original signal and re_referenced
plt.plot(time, re_reference, alpha=0.6, label="Unfiltered")
plt.plot(time, Filtered_sig_80, label="Filtered")
plt.title("Filtering the signal")
plt.xlabel("Time(s)")
plt.ylabel("Signal")
plt.legend()
plt.show()

# plotting the fft before and after filtering
plt.plot(freq_org, fft_org, color='b', alpha=0.5, label='Unfiltered fft')
plt.plot(freq_filt, fft_filt, color='r', alpha=0.6, label='Filtered fft')
plt.title("FFT of the filtered and unfiltered signals")
plt.xlabel("Frequencies (Hz)")
plt.legend()
plt.show()
```

```
time: 437.0
```


Filtering the signal
Unfiltered
Filtered
8000
6000
4000
2000
0
Signal
0
100
200
300
400
Time(s)



Now it is time to epoch your data into the events, here it is important to cut your signal 5ms before the trigger onset (indx events) and analyse up to 20ms after the trigger event. Count the number of events you will have (len(indx)), and then make a matrix which has dimensions (len(indx) x samples), where samples correspond to a time axis which starts 5 ms before the click (event occurs at 0) and runs 20 ms until after the click. Once you made your epochs, make a figure which plots time vs all your epochs at once (you might have to transpose your matrix to plot using the ".T" function). Afterwards, subtract the mean of each epoch from the same epoch to apply the baseline correction. This visualisation is helpful when determining a threshold above which you will remove "bad" or "noisy" epochs. Here, you will likely only remove 5 or 10 % of the noisiest epochs (the ones with the largest absolute amplitude). Write a code to automatically remove equal number of epochs of each polarity with amplitudes above a reasonable "self-determined" threshold value. Discard the noisy epochs and now compare your epochs before and after noise-rejecting. Did your algorithm work?

Once you have rejected the noisy epochs, concatenate the epochs of each polarity and then average the epochs and have a look at the mean ABR waveform (plotted over time). Do you notice the wave-V peak near 6-7 ms? Perform the same preprocessing steps for the 100 dB ABR data. Compare the wave-1 and 5 peaks and latencies in two ABR recordings (ABR80 and ABR100) and provide your explanation in this regard at the bottom of your code.

NOTE: Regarding the ABR peaks latencies, consider an approximate 1.2 ms offset caused by sound

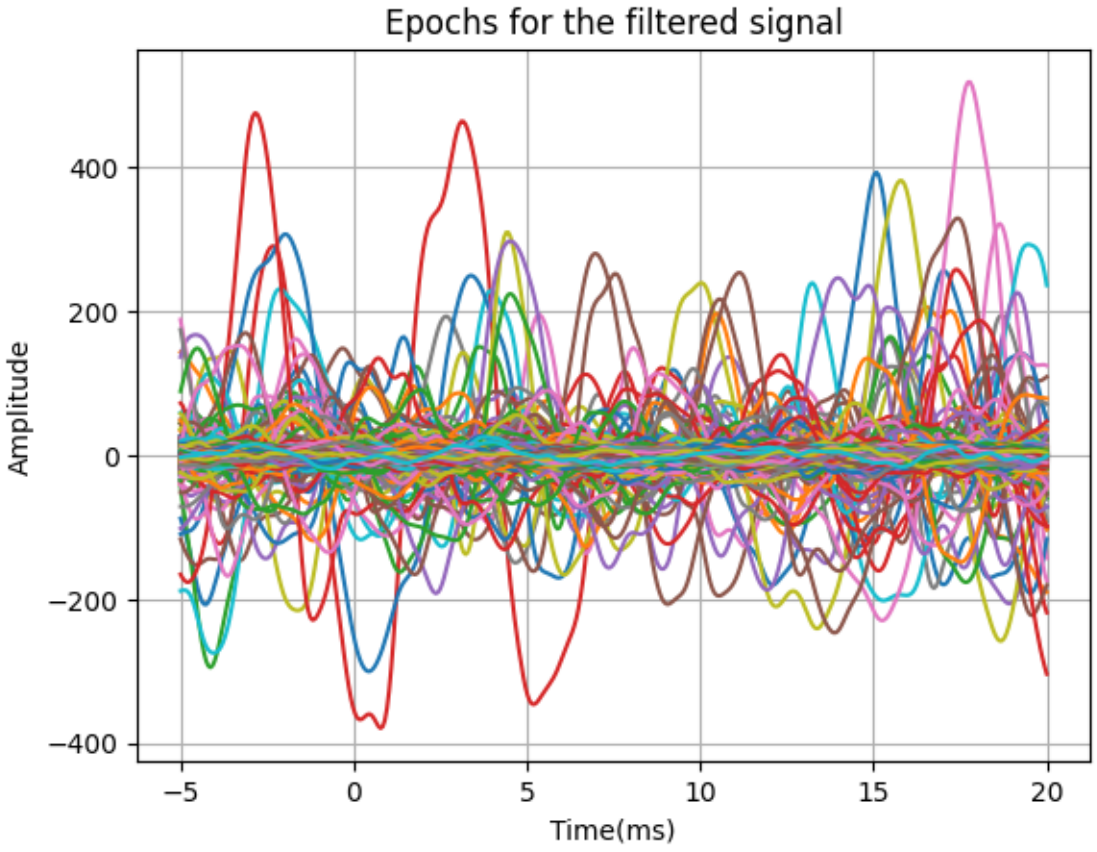
delivery system.

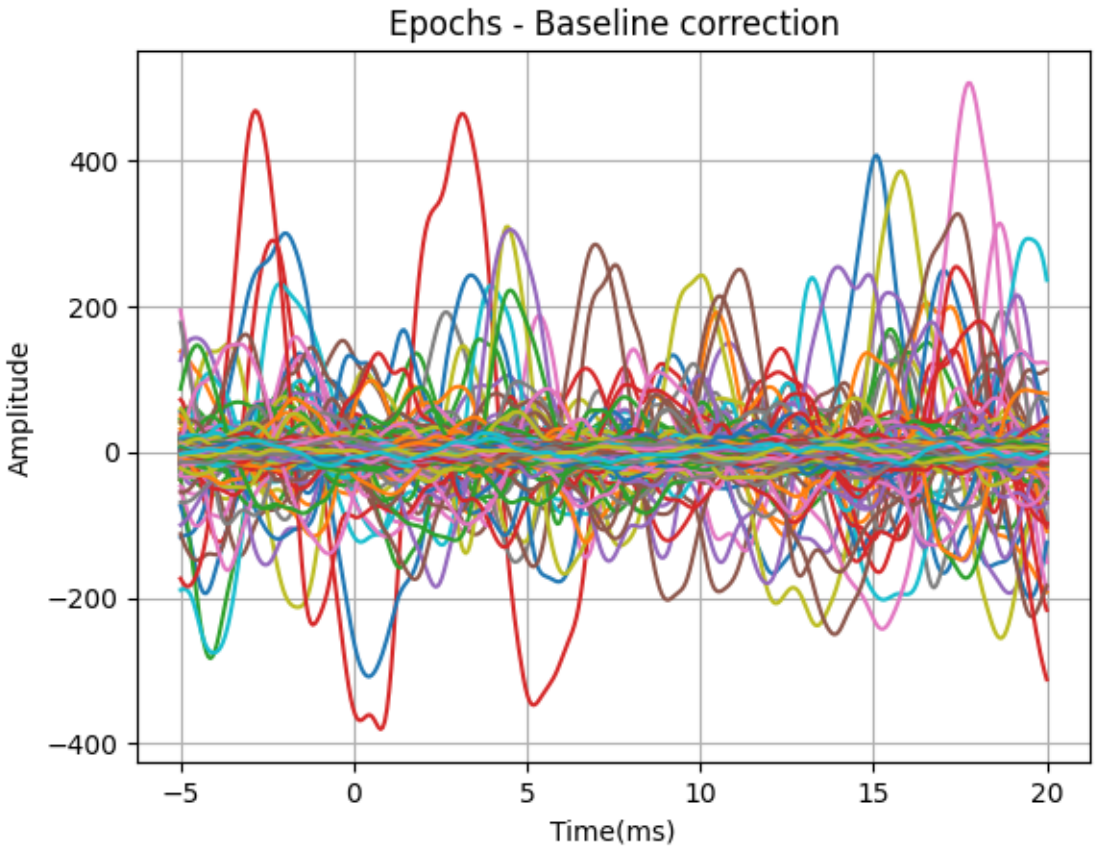
[81]:
```
print(f"fs: {FS}")
FSm = FS / 1000
nsamples_epoch = (25*FSm)-1 # int is 410
print(f"# of samples in an epoch: {nsamples_epoch}")

epochs = np.zeros([len(indx_80), int(nsamples_epoch)]) # rows x columns ->␣
 ↪len(indx) x samples
time_epoch = np.linspace(-5, 20, 408)

print(f"shape epochs: {epochs.shape}") # 3000 triggers, 410 samples in 25ms

# creating epochs matrix
for i in range(len(indx_80)):
    click = indx_80[i]
    epochs[i] = Filtered_sig_80[click - int(5*FSm): click + int(20*FSm)] # from␣
 ↪(click - 5ms) to (click + 20ms)
    plt.plot(time_epoch, epochs[i])
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.title("Epochs for the filtered signal")
plt.grid()
plt.show()


# baseline correction for each epoch
epochs_baseline = [(epoch - np.mean(epoch)) for epoch in epochs]

# plotting every epoch (baseline)
for i in range(len(epochs)):
    plt.plot(time_epoch, epochs_baseline[i])

plt.title("Epochs - Baseline correction")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()


# Denoising algorithm
def denoise(epochs, denoise_size):
    n_epochs = np.array(epochs).shape[0] # 3000 epochs

    sum_epochs = np.zeros(n_epochs)
    # summation of every epoch to categorize into the 10% epochs with highest␣
 ↪abs amplitudes
```

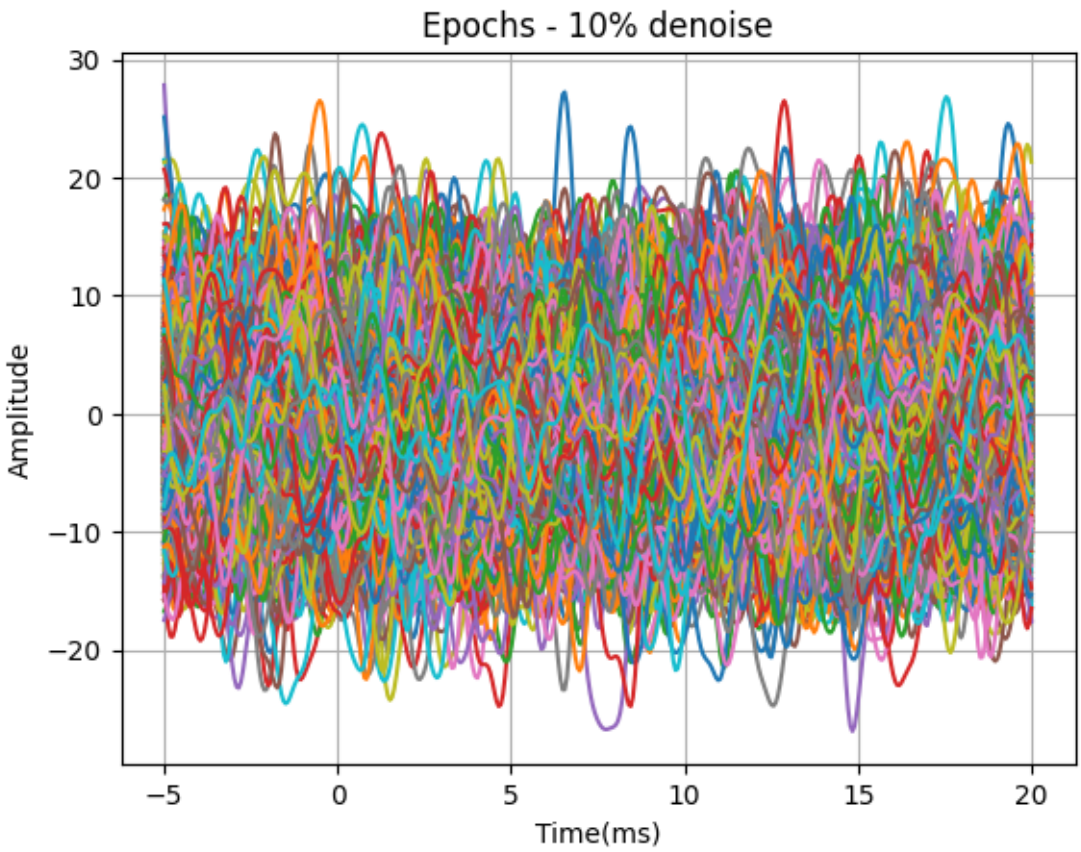
```
    for i in range(n_epochs):
        sum_epochs[i] = np.sum(np.abs(epochs[i][:]))

    sorted_sum_epochs = np.array(np.argsort(sum_epochs))[::-1] # sort in␣
 ↪descending order -> want to see indices for the noisiest epochs
    to_delete = sorted_sum_epochs[ : int(n_epochs * denoise_size)] # indices of␣
 ↪10% noisiest epochs

    # removing noisy epochs
    denoised_epochs_80 = []
    for i in range(n_epochs):
        if i in to_delete: # if the index is in the list of indices to delete,␣
 ↪just go to the next epoch
            continue
        denoised_epochs_80.append(epochs[i]) # if the epoch is not on the list␣
 ↪"to delete" then it's a good value, append it

    return denoised_epochs_80


denoised_epochs_80 = denoise(epochs_baseline, 0.1)

# plotting every epoch (baseline)
for i in range(len(denoised_epochs_80)):
    plt.plot(time_epoch, denoised_epochs_80[i])

plt.title("Epochs - 10% denoise")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()

print(f"epochs after denoising - 10%: {np.array(denoised_epochs_80).shape}") #␣
 ↪2700 remaining epochs

ABR_80 = np.mean(denoised_epochs_80, axis=0)

plt.plot(time_epoch, ABR_80)
x_ticks = np.append(plt.xticks()[0], 7)
plt.axvline(x=7, color='r', label="7ms peak", alpha=0.6, ls='dotted')
plt.xticks(x_ticks)
plt.xlim([-6, 21])
plt.title('ABR for the filtered signal - 80dB (10% denoise)')
plt.xlabel('Time(ms)')
plt.ylabel('Voltage [mV]')
plt.legend()
plt.grid()
```

```
plt.show()


'''
The denoising with 10% excluded values worked notably well for this dataset.␣
 ↪The response at 7ms is well visible and the signal does not seem to have
significant artifacts that can obscure the interpretation of the epochs.
'''
```

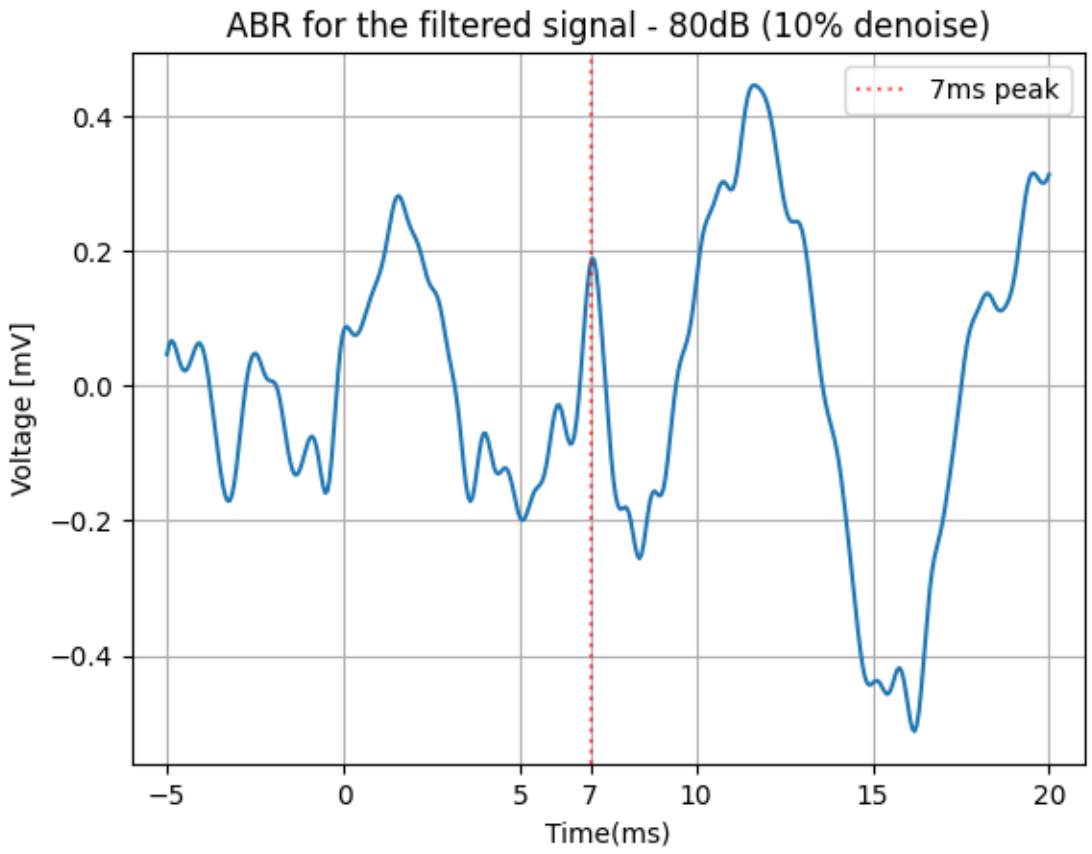
```
fs: 16384
# of samples in an epoch: 408.6
shape epochs: (3000, 408)
```




Epochs - Baseline correction
Amplitude
400
200
0
−200
−400
−5
0
5
10
15
20
Time(ms)




```
epochs after denoising - 10%: (2700, 408)
```



[81]:
```
'\nThe denoising with 10% excluded values worked notably well for this dataset.
The response at 7ms is well visible and the signal does not seem to have
\nsignificant artifacts that can obscure the interpretation of the epochs.\n'
```

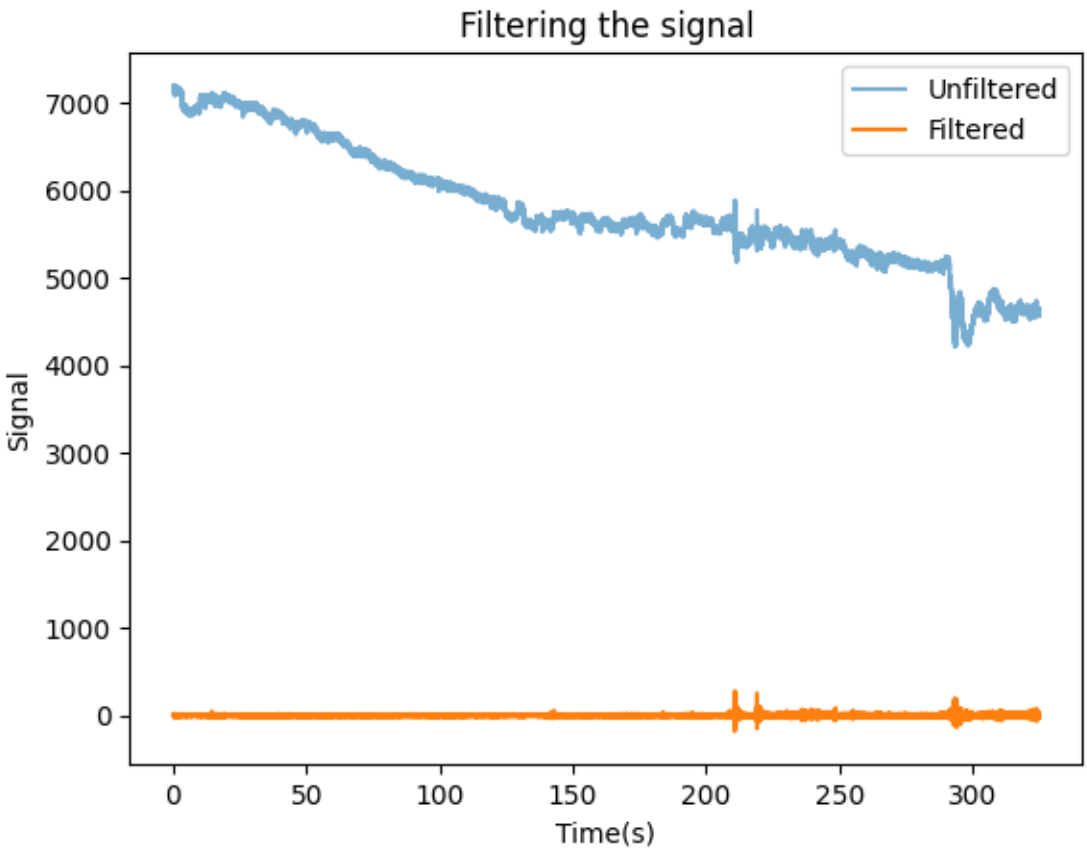
[82]:
```
# loading the data
ABRData70 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"ABR_70_Cz.mat"))
ABR70_ref1 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"ABR_70_ref1.mat"))
ABR70_ref2 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"ABR_70_ref2.mat"))
ABR70_trig = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"TrigsABR_70.mat"))

print(ABRData70.keys())
```

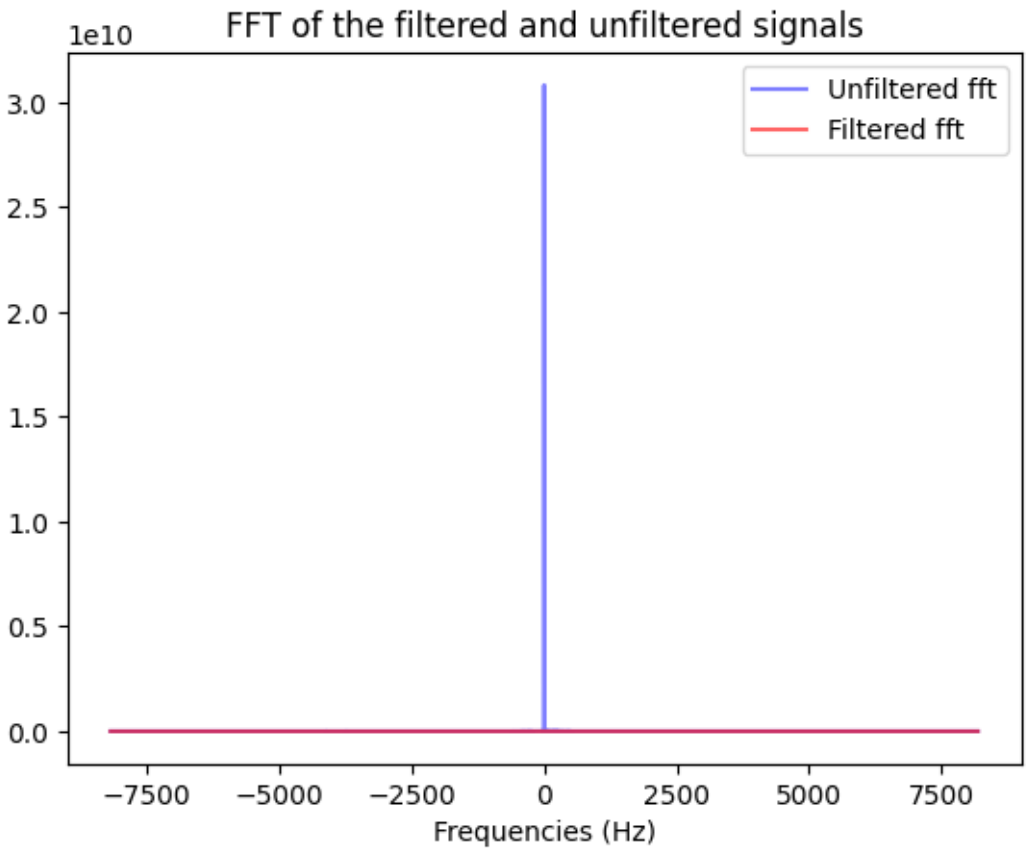
```
FS = 16384 # Sampling frequency
Sig_70 = ABRData70['Ch'] #has the raw recording trace of EEG data
ref1_70 = ABR70_ref1['Ch']
ref2_70 = ABR70_ref2['Ch']

T_70 = ABR70_trig['Trigs'] # channel with equal duration of the recording, but␣
 ↪has the trigger events

#find all the triggers
indx_70 = np.where(np.diff(T_70)==1)[1]+1 #index has the sample numbers at␣
 ↪which an event started

avg_ref1_ref2 = [(abs(ref1_70[0][i] - ref2_70[0][i]) / 2) for i in␣
 ↪range(len(ref1_70[0]))]
re_reference_70 = [(Sig_70[0][i] - avg_ref1_ref2[i]) for i in␣
 ↪range(len(ref1_70[0]))]


time_70 = np.linspace(0, len(re_reference_70) / FS, len(re_reference_70)) #␣
 ↪time vector for the signal


#                        FILTERING AND FFT
Filtered_sig_70 = filtering(re_reference_70, 4, FS, 1500, 100) # filtering␣
 ↪signal with lowpass and highpass as with sig_80

freq_org_70, fft_org_70 = fft_sig(re_reference_70, re_reference_70, FS) #
freq_filt_70, fft_filt_70 = fft_sig(re_reference_70, Filtered_sig_70, FS)


# plotting original signal and re_reference_70d
plt.plot(time_70, re_reference_70, alpha=0.6, label="Unfiltered")
plt.plot(time_70, Filtered_sig_70, label="Filtered")
plt.title("Filtering the signal")
plt.xlabel("Time(s)")
plt.ylabel("Signal")
plt.legend()
plt.show()

# plotting the fft before and after filtering
plt.plot(freq_org_70, fft_org_70, color='b', alpha=0.5, label='Unfiltered fft')
plt.plot(freq_filt_70, fft_filt_70, color='r', alpha=0.6, label='Filtered fft')
plt.title("FFT of the filtered and unfiltered signals")
plt.xlabel("Frequencies (Hz)")
plt.legend()
```

```
plt.show()


#                         EPOCHS AND DENOISING
FSm = FS / 1000
nsamples_epoch = (25*FSm)-1 # int is 410
print(f"# of samples in an epoch: {nsamples_epoch}")

epochs_70 = np.zeros([len(indx_70), int(nsamples_epoch)]) # rows x columns ->␣
 ↪len(indx) x samples
time_epoch_70 = np.linspace(-5, 20, 408)

print(f"shape epochs_70: {epochs_70.shape}") # ?

# creating epochs matrix
for i in range(len(indx_70)):
    click = indx_70[i]
    epochs_70[i] = Filtered_sig_70[click - int(5*FSm): click + int(20*FSm)] #␣
 ↪from (click - 5ms) to (click + 20ms)
    plt.plot(time_epoch_70, epochs_70[i])
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.title("Epochs_70 for the filtered signal")
plt.grid()
plt.show()


# baseline correction for each epoch
epochs_70_baseline = [(epoch - np.mean(epoch)) for epoch in epochs_70]

# plotting every epoch (baseline)
for i in range(len(epochs_70)):
    plt.plot(time_epoch_70, epochs_70_baseline[i])

plt.title("Epochs_70 - Baseline correction")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()


denoised_epochs_70 = denoise(epochs_70_baseline, 0.1)

# plotting every epoch (baseline)
for i in range(len(denoised_epochs_70)):
    plt.plot(time_epoch_70, denoised_epochs_70[i])
```

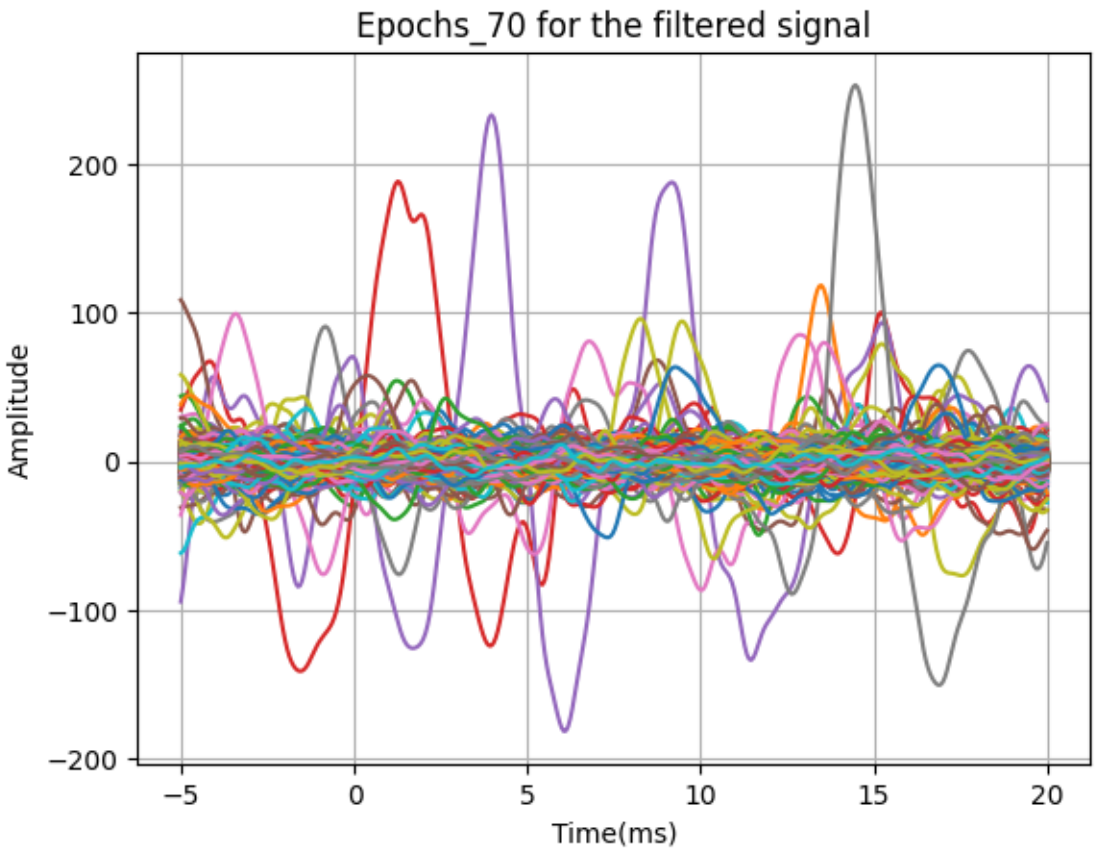
```
plt.title("Epochs_70 - 10% denoise")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()

print(f"epochs_70 after denoising - 10%: {np.array(denoised_epochs_70).shape}")␣
 ↪# 2700 remaining epochs_70

ABR_70 = np.mean(denoised_epochs_70, axis=0)

plt.plot(time_epoch, ABR_80, label="ABR 80dB")
plt.plot(time_epoch_70, ABR_70, label="ABR 70dB")
plt.legend()
plt.title('ABR comparison - 80dB | 70dB')
plt.xlabel('Time(ms)')
plt.ylabel('Voltage [mV]')
plt.grid()
plt.show()


'''
The 70dB signal seems less clean than the 80dB one and it is more difficult to␣
 ↪observe the response to the impulses. This is noticeable in the ABR
as well as the denoised epochs that show some peaks that are a bit far from the␣
 ↪center.
'''
```

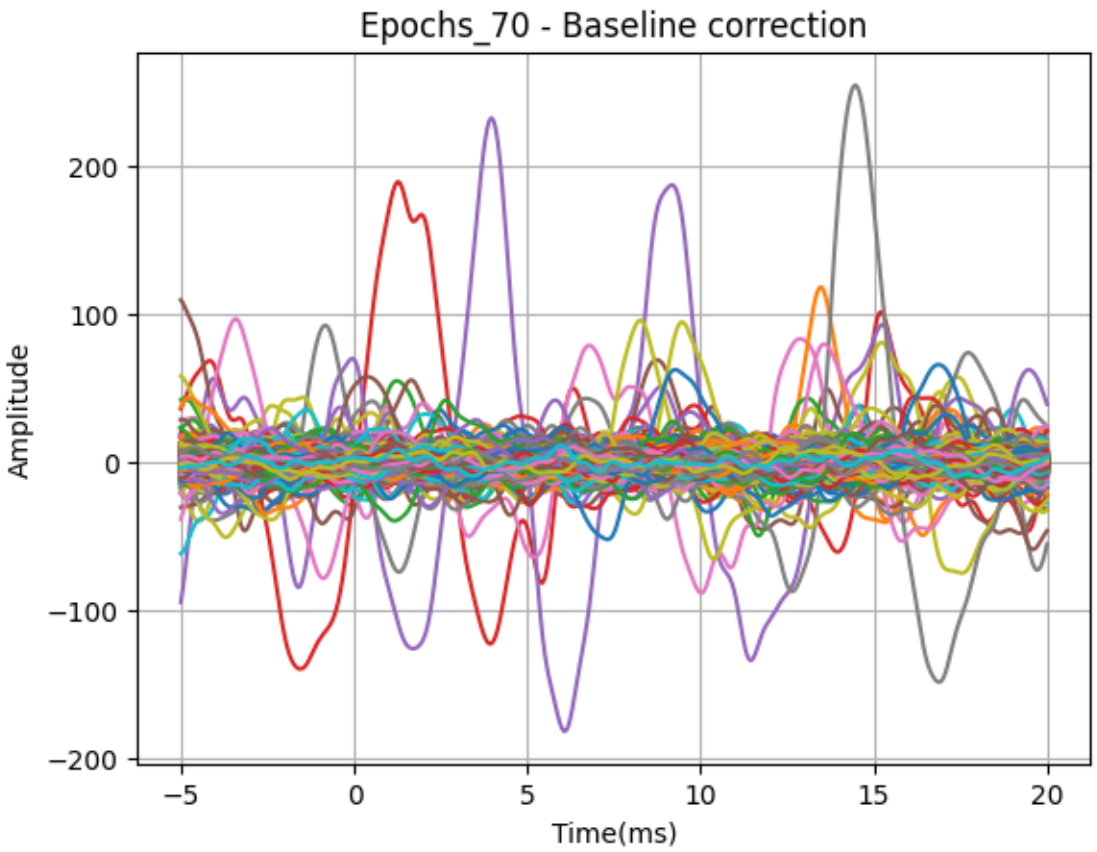
```
dict_keys(['__header__', '__version__', '__globals__', 'Ch'])
```

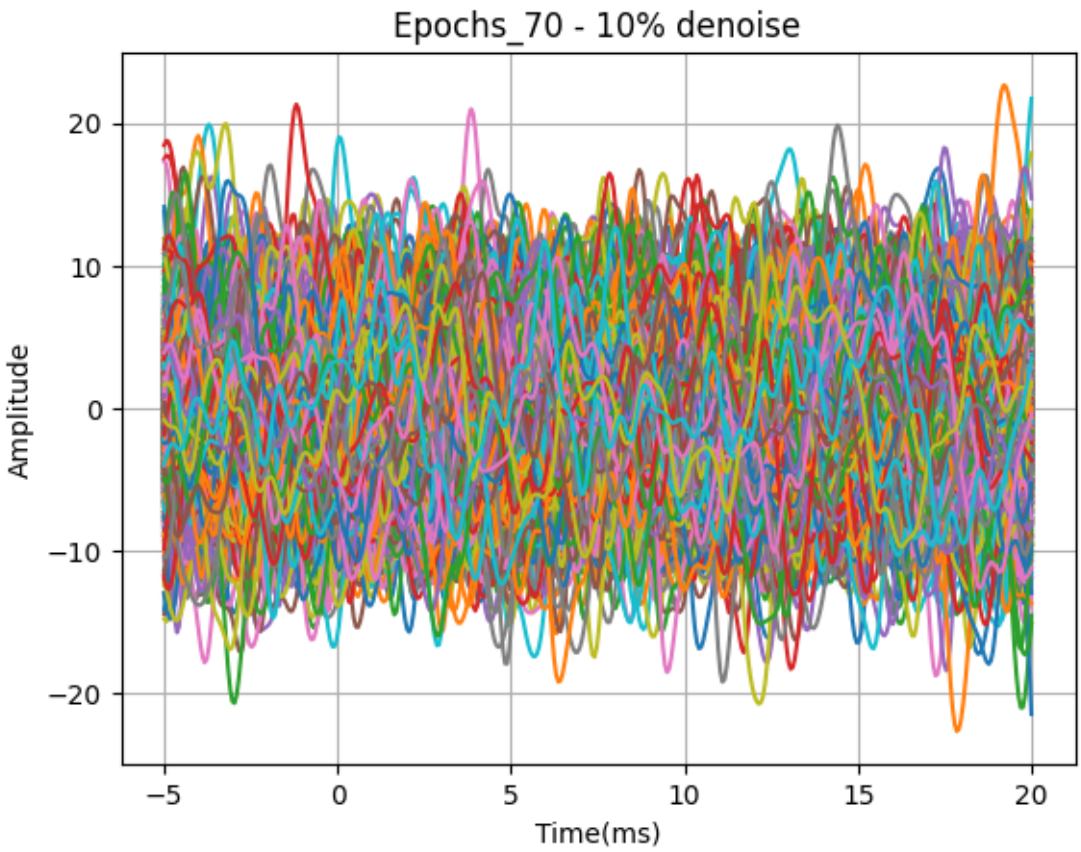

Filtering the signal
Unfiltered
Filtered
Signal
7000
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
300
Time(s)



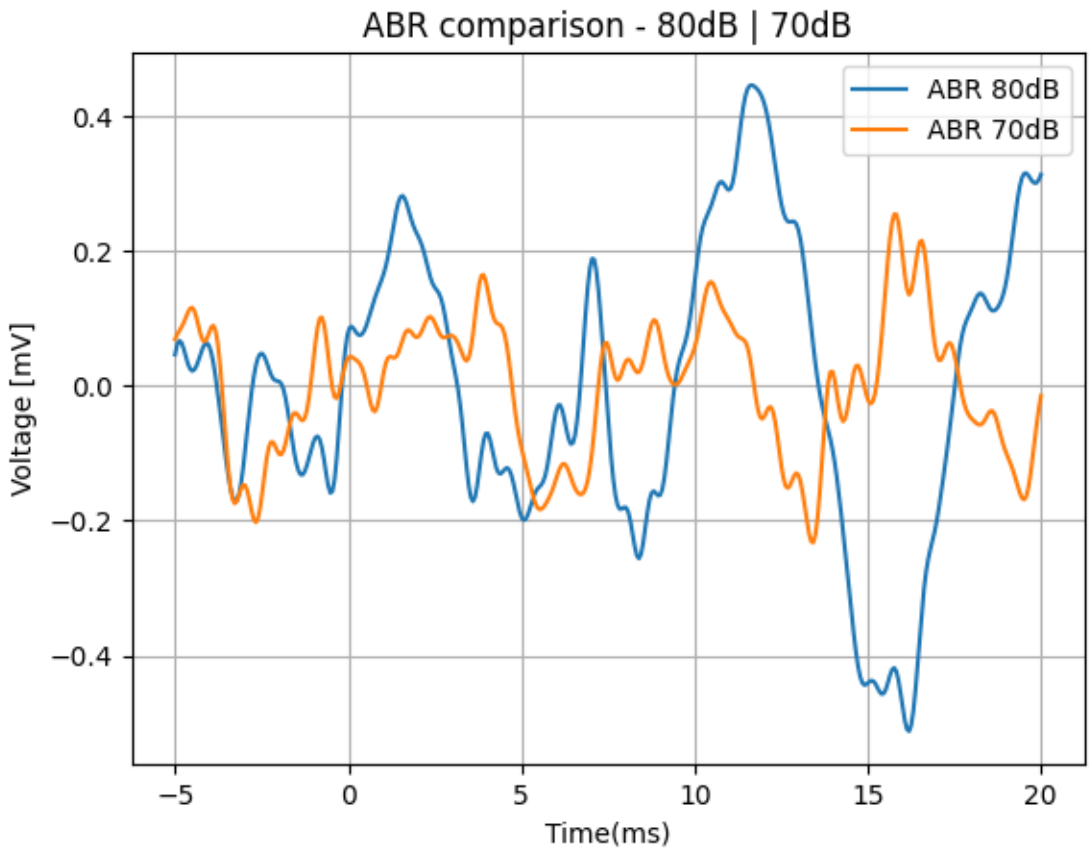

```
# of samples in an epoch: 408.6
shape epochs_70: (3000, 408)
```


Epochs_70 for the filtered signal
Amplitude
Time(ms)
200
100
0
−100
−200
−5
0
5
10
15
20


Epochs_70 - Baseline correction
Amplitude
200
100
0
−100
−200
−5
0
5
10
15
20
Time(ms)




```
epochs_70 after denoising - 10%: (2700, 408)
```

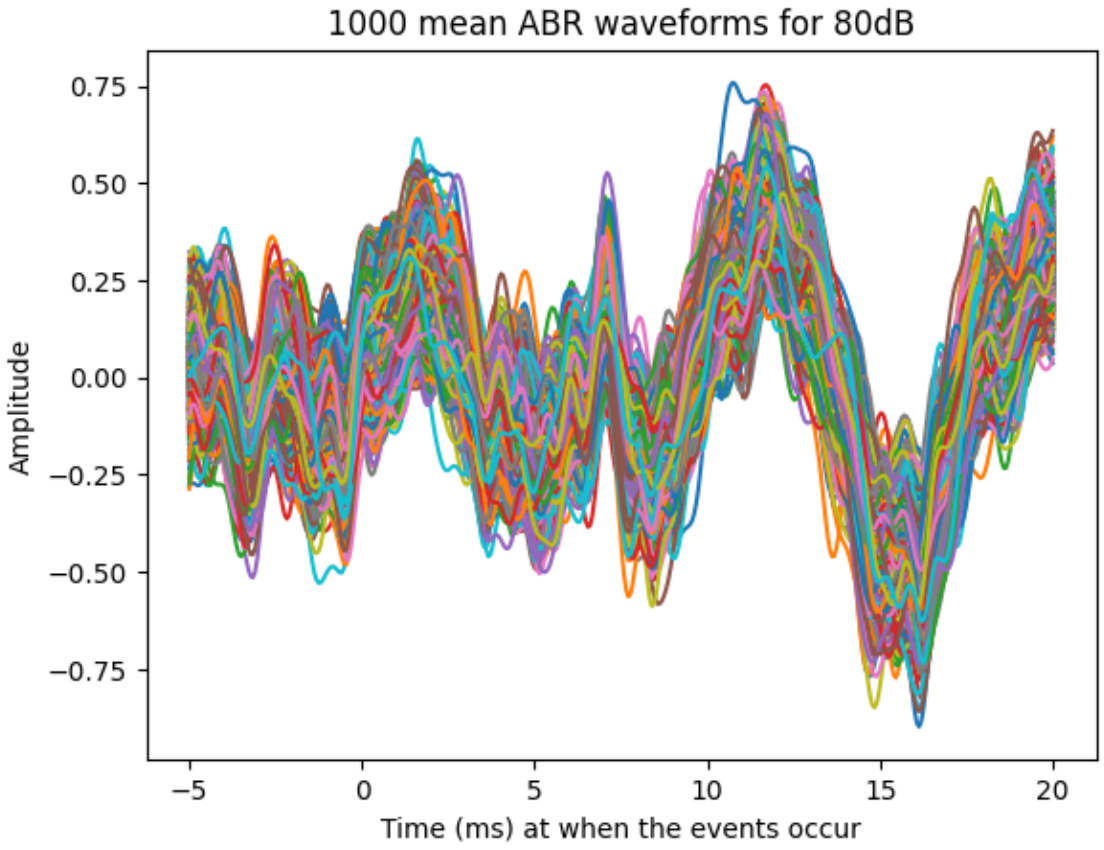


[82]: '\nThe 70dB signal seems less clean than the 80dB one and it is more difficult to observe the response to the impulses. This is noticeable in the ABR\nas well as the denoised epochs that show some peaks that are a bit far from the center.\n'

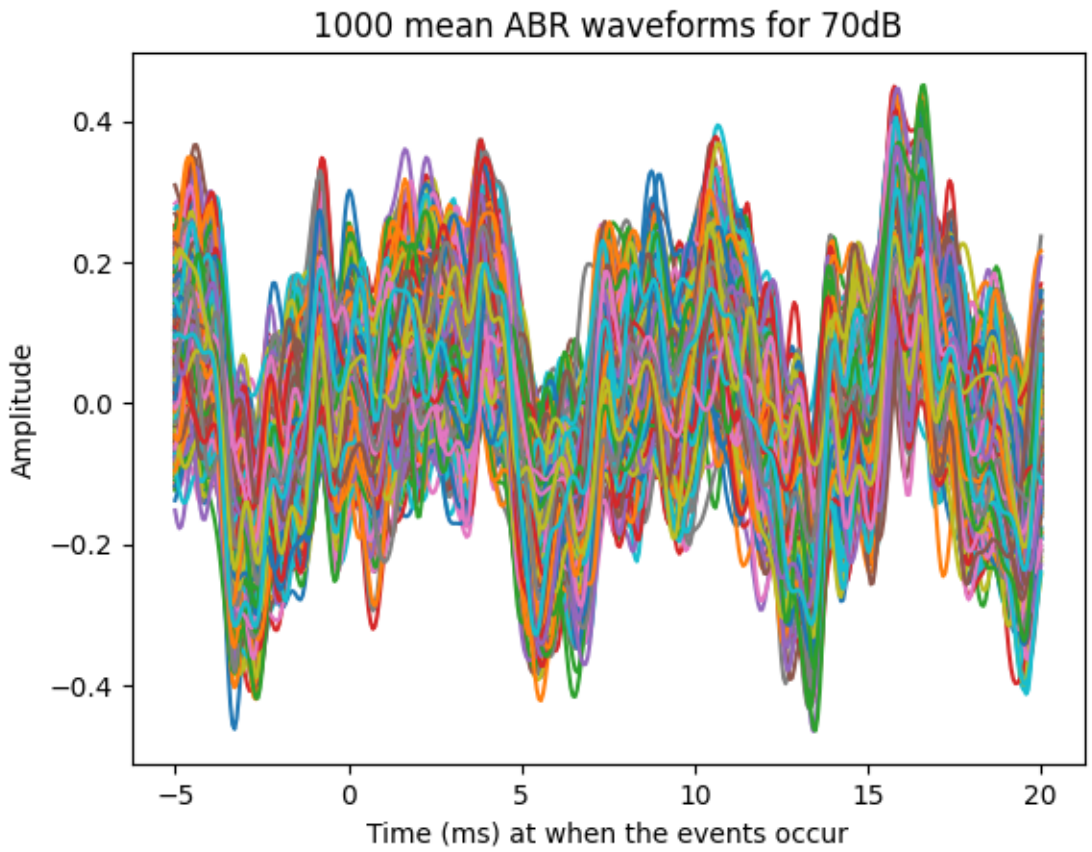
To perform statistics on the signal peaks of the 80 and 100 dB ABR waveforms, it is necessary to quantify the variability of each of the datapoints in the waveform. You can estimate this variability by using a resampling method which calculates a waveform corresponding to the 5 and 95% datapoint on each time sample. In this way, you generate a 5% and 95% Confidence Interval (CI) on the data-points of the mean waveform (i.e. approximately corresponding to the standard error of the mean) which you can use to evaluate the significance of waveform feature differences in the two recordings. - Use the matrix with epochs (after noise-rejection) to generate a 5th and 95th percentile waveform of the 80 or 100 dB . The specific steps include, (i) calculate and store 1000 resampled mean waveforms from the epoch matrix, by randomly sampling epochs (with replacement) from the matrix before you calculate the mean waveform (use np.random.choice to get the random indices, and pick out the corresponding epochs before you calculate the mean signal for each of the 1000 resampling loops). - After this procedure, you have 1000 mean waveforms over time stored. You can plot your matrix with 1000 mean ABR waveforms (use yourmatrixname.T in case there is a dimension mismatch with your time vector), to see that most mean ABR waveforms preserve the ABR waveform maximum near 5 ms.

[83]:
```
time_epoch = np.linspace(-5, 20, 408)
waveforms_80dB = np.zeros([1000, 408])
waveforms_70dB = np.zeros([1000, 408])

def random_sample_1000(epochs):

  for i in range(1000):
    sample_random_epochs = np.random.choice(len(epochs), size=(len(epochs)),␣
 ↪replace=True) # sampling random 1000 epochs
    samples = [epochs[idx] for idx in sample_random_epochs] # array of random␣
 ↪samples

    mean_waveform = np.mean(samples,axis=0)
    # waveforms_80dB[i,:] = mean_waveform
    plt.plot(time_epoch, mean_waveform_80dB)
  plt.title('1000 mean ABR waveforms for 80dB')
  plt.xlabel('Time (ms) at when the events occur')
  plt.ylabel('Amplitude')
  plt.show()

  return samples, mean_waveform

# samples_80dB, mean_waveform_80dB = random_sample_1000(denoised_epochs_80)

for i in range(1000):
  sample_random_epochs_80dB = np.random.choice(len(denoised_epochs_80),␣
 ↪size=(len(denoised_epochs_80)), replace=True) # sampling random 1000 epochs
  samples_80dB = [denoised_epochs_80[idx] for idx in sample_random_epochs_80dB]␣
 ↪# array of random samples

  mean_waveform_80dB = np.mean(samples_80dB,axis=0)
  waveforms_80dB[i,:] = mean_waveform_80dB
  plt.plot(time_epoch, mean_waveform_80dB)
plt.title('1000 mean ABR waveforms for 80dB')
plt.xlabel('Time (ms) at when the events occur')
plt.ylabel('Amplitude')
plt.show()

print(np.array(mean_waveform_80dB).shape)
print(np.array(samples_80dB).shape)


for i in range(1000):
  sample_random_epochs_70dB = np.random.choice(len(denoised_epochs_70),␣
 ↪size=(len(denoised_epochs_70)), replace=True) # sampling random 1000 epochs
  samples_70dB = [denoised_epochs_70[idx] for idx in sample_random_epochs_70dB]␣
 ↪# array of random samples
```

```
  mean_waveform_70dB = np.mean(samples_70dB,axis=0)
  waveforms_70dB[i,:] = mean_waveform_70dB
  plt.plot(time_epoch, mean_waveform_70dB)
plt.title('1000 mean ABR waveforms for 70dB')
plt.xlabel('Time (ms) at when the events occur')
plt.ylabel('Amplitude')
plt.show()

print(np.array(mean_waveform_70dB).shape)
print(np.array(samples_70dB).shape)
```




```
(408,)
(2700, 408)
```



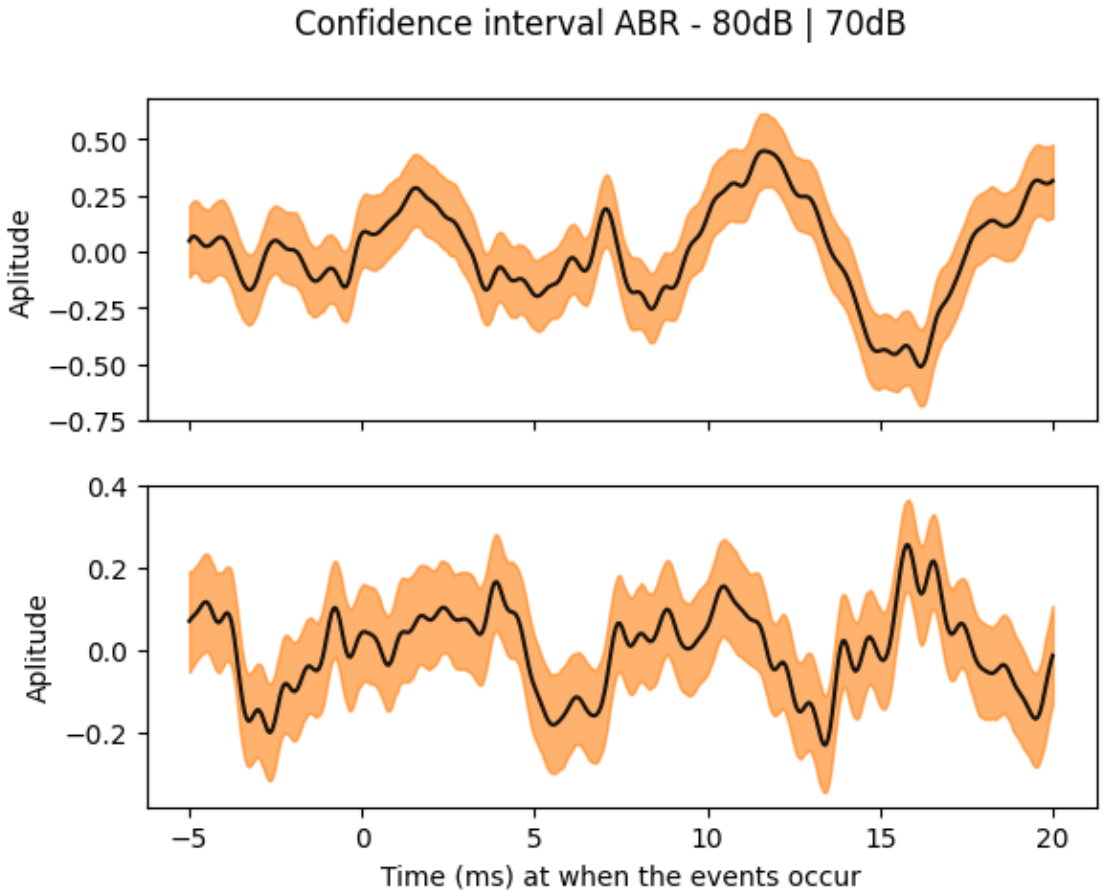

```
(408,)
(2700, 408)
```

Next, you can calculate the CI05 and CI95 waveforms out of the (1000 x time sample) matrix with resampled mean signals. To do this, you need to write a for loop that has an equal length as your time vector ("for n in range(0,len(time)):"), and that for each sample, ranks the values from low to high. You can store the indices of ranked values using the "np.argsort" function. Once you did this, you have the indices corresponding to ascending order values for each time sample. Take index numbers [49] and [949] out for each time-sample to generate the CI05 and CI95 waveforms respectively. - For the Ch 80 and 100 dB waveforms, plot the mean waveform (before bootstrapping) along with the CI05 and CI95 waveforms on the same plot (with different colors for the 80 and 100 dB condition). You can also modify the following script to plot the confidence interval as a filled area around the mean waveform (but then it is graphically better to plot the results on two figures instead). - plt.show() - fig, ax = plt.subplots(1,1,sharex=True) - ax.plot(t,xmean,color='k') - ax.fill_between(t, CI05[:,0], CI95[:,0],color='C1') - plt.show()

[85]:
```
#                              80dB CI
CI05_80 = np.zeros(len(time_epoch))
CI95_80 = np.zeros(len(time_epoch))
print(CI05_80.shape)
```

```
for n in range(len(time_epoch)):
    ampl_timestamp = waveforms_80dB[:, n]
    indx_sorted = np.argsort(ampl_timestamp)

    # compute 5th and 95th percentile
    CI05_80_index = indx_sorted[int(len(indx_sorted)*0.05)]
    CI95_80_index = indx_sorted[int(len(indx_sorted)*0.95)]

    # save the computed percentiles to the CI05_80, CI95_80 arrays
    CI05_80[n] = ampl_timestamp[CI05_80_index]
    CI95_80[n] = ampl_timestamp[CI95_80_index]



#                                                   70dB CI
CI05_70 = np.zeros(len(time_epoch))
CI95_70 = np.zeros(len(time_epoch))
print(CI05_70.shape)

for n in range(len(time_epoch)):
    ampl_timestamp = waveforms_70dB[:, n]
    indx_sorted = np.argsort(ampl_timestamp)

    # compute 5th and 95th percentile
    CI05_70_index = indx_sorted[int(len(indx_sorted)*0.05)]
    CI95_70_index = indx_sorted[int(len(indx_sorted)*0.95)]

    # save the computed percentiles to the CI05_70, CI95_70 arrays
    CI05_70[n] = ampl_timestamp[CI05_70_index]
    CI95_70[n] = ampl_timestamp[CI95_70_index]


# plotting
fig, ax = plt.subplots(2,1,sharex=True)

ax[0].plot(time_epoch, ABR_80, color='k', alpha=0.9)
ax[0].fill_between(time_epoch, CI05_80, CI95_80,color='C1', alpha=0.6)
ax[0].set_ylabel("Aplitude")

ax[1].plot(time_epoch, ABR_70, color='k', alpha=0.9)
ax[1].fill_between(time_epoch, CI05_70, CI95_70,color='C1', alpha=0.6)
ax[1].set_ylabel("Aplitude")

fig.suptitle('Confidence interval ABR - 80dB | 70dB ')
plt.xlabel('Time (ms) at when the events occur')
plt.show()
```

```
(408,)
(408,)
```



3 Part 2: Envelope Following Response (EFR)

The next recording is one from an envelope-following response (EFR). Here the stimulus was a 500-ms long, 70 dB SPL, 4-kHz pure tone which was modulated with a 120 Hz envelope. When the auditory-nerve fibers are intact, the EEG response will follow the stimulus envelope frequency, and hence we need to analyse the strength of the response in the frequency domain. Again, the triggers may be of positive and negative polarity, and you can follow the same steps as of ABR to load in the data and identify the samples at which the onsets occur. Afterwards, plot the Cz channel, reference channels, average of the two reference channels and triggers in different panels of a subplot. Then, similar to ABRs, re-reference the Cz channel recording to the average of the reference channels and plot the result in a separate figure. The labeling of the files is similar to that of the ABRs:

1. SAM_EFR_Cz: recorded signal from the electrode placed on the top of head (i.e. the B16 electrode)
2. SAM_EFR_ref1: recorded signal from the left earlobe electrode

3. SAM_EFR_ref2: recorded signal from the right earlobe electrode
4. TrigsSAM_EFR: trigger signal which indicates the onset of each epoch by "1". Odd epochs were presented with positive polarity and even epochs with negative polarity.

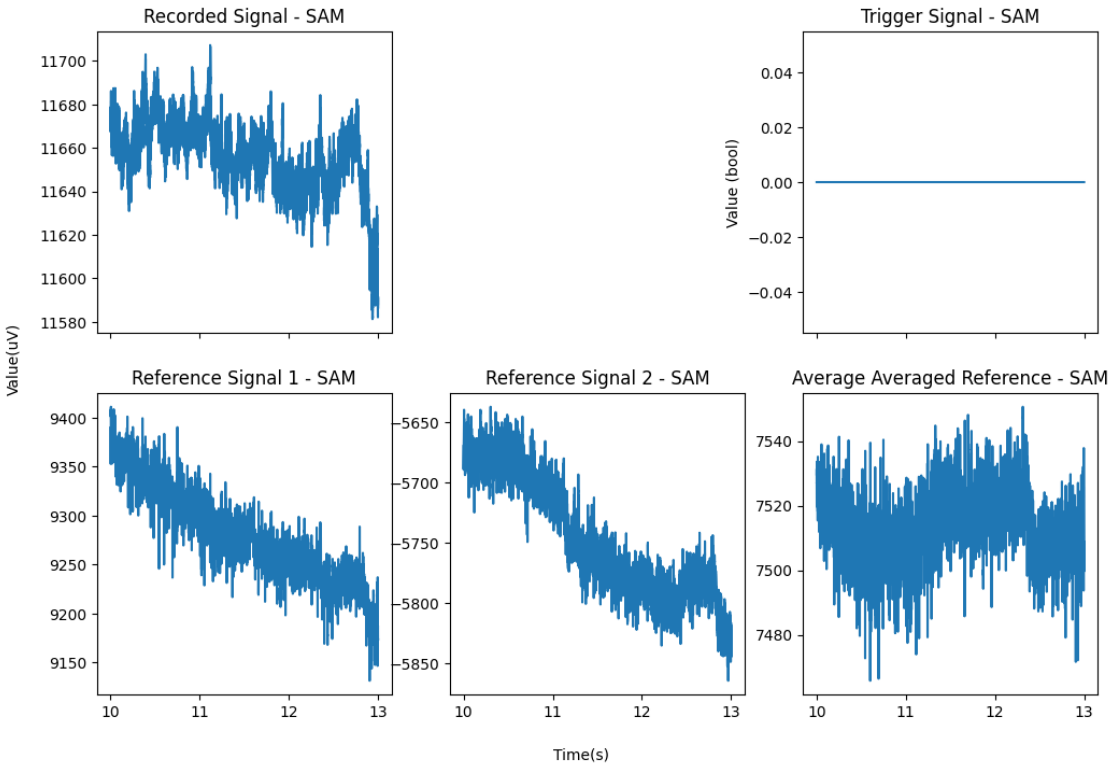
Note: - The sampling frequency (FS) is 16384 Hz. - The stimulus was repeated 1000 times (500 positive and 500 negative epochs) - Limit the plots' x-axis to the range of 3 seconds (e.g. 10 to 13 seconds).

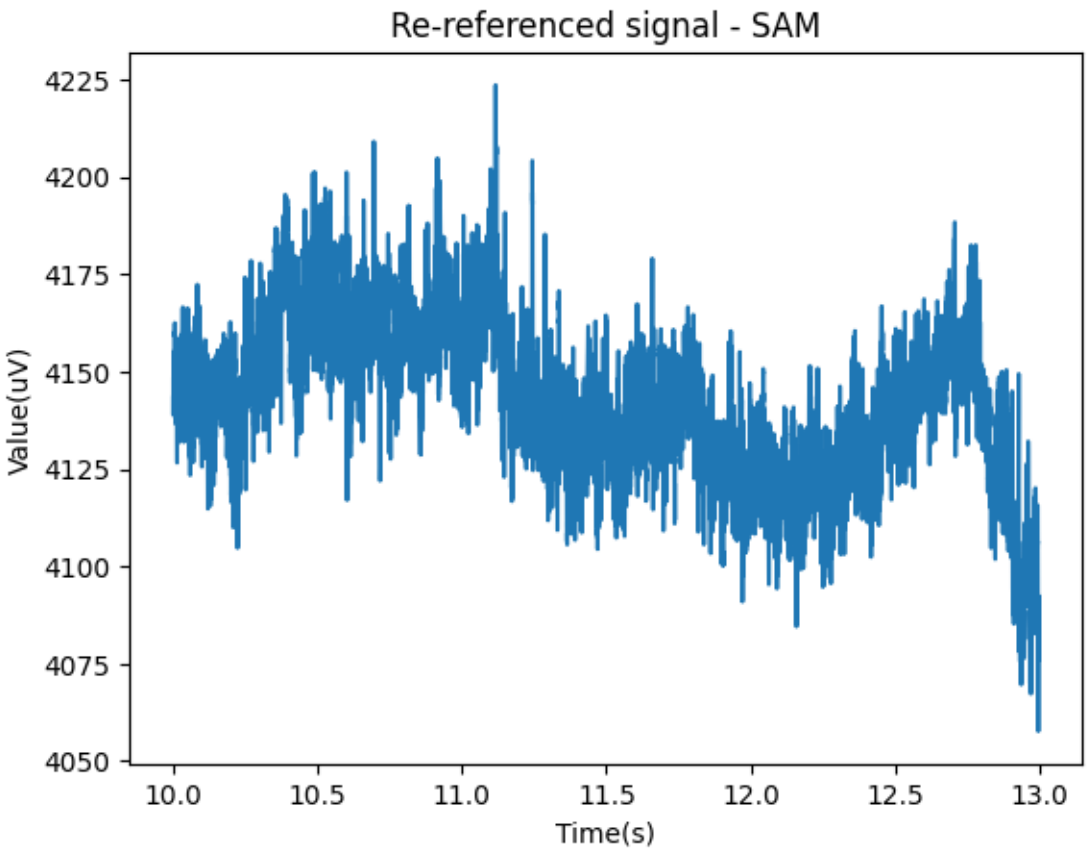
[86]:
```
#Load in the data
SAM_data = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"SAM_EFR_Cz.mat"))
SAM_EFR_ref1 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"SAM_EFR_ref1.mat"))
SAM_EFR_ref2 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"SAM_EFR_ref2.mat"))
SAM_EFR_trig = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"TrigsSAM_EFR.mat"))

FS_EFR = 16384 # Sampling frequency EFR
Sig_SAM = SAM_data['Ch'] #has the raw recording trace of EEG data
ref1_SAM = SAM_EFR_ref1['Ch']
ref2_SAM = SAM_EFR_ref2['Ch']


T_SAM = SAM_EFR_trig['Trigs']

#find all the triggers
indx_SAM = np.where(np.diff(T_SAM)==1)[1]+1 #index has the sample numbers at␣
 ↪which an event started

avg_ref1_ref2_SAM = [(abs(ref1_SAM[0][i] - ref2_SAM[0][i]) / 2) for i in␣
 ↪range(len(ref1_SAM[0]))]
re_reference_SAM = [(Sig_SAM[0][i] - avg_ref1_ref2_SAM[i]) for i in␣
 ↪range(len(ref1_SAM[0]))]

print(SAM_data.keys())
```

```
dict_keys(['__header__', '__version__', '__globals__', 'Ch'])
```

[87]:
```
# Plotting Cz channel, reference channels, avg of the two reference channels,␣
 ↪and triggres in different panels of a subplot
seconds_SAM = [10, 13]
```

```
time_window_SAM = np.arange(seconds_SAM[0], seconds_SAM[1], 1/FS) # time array␣
 ↪for the desired seconds


figs, axs = plt.subplots(2, 3, figsize=(12, 8), sharex=True)

figs.text(0.5, 0.04, 'Time(s)', ha='center', va='center')
figs.text(0.06, 0.5, 'Value(uV)', ha='center', va='center', rotation='vertical')

axs[0][0].plot(time_window_SAM, Sig_SAM[0][seconds_SAM[0] * FS: seconds_SAM[1]␣
 ↪* FS])
axs[0][0].set_title("Recorded Signal - SAM")

axs[0][2].plot(time_window_SAM, T_SAM[0][seconds_SAM[0] * FS: seconds_SAM[1] *␣
 ↪FS])
axs[0][2].set_ylabel("Value (bool)")
axs[0][2].set_title("Trigger Signal - SAM")

axs[0][1].axis('off') # leave a blank space in the subplot

axs[1][0].plot(time_window_SAM, ref1_SAM[0][seconds_SAM[0] * FS: seconds_SAM[1]␣
 ↪* FS])
axs[1][0].set_title("Reference Signal 1 - SAM")

axs[1][1].plot(time_window_SAM, ref2_SAM[0][seconds_SAM[0] * FS: seconds_SAM[1]␣
 ↪* FS])
axs[1][1].set_title("Reference Signal 2 - SAM")

axs[1][2].plot(time_window_SAM, avg_ref1_ref2_SAM[seconds_SAM[0] * FS:␣
 ↪seconds_SAM[1] * FS])
axs[1][2].set_title("Average Averaged Reference - SAM")

plt.show()

# plotting the re-referenced values
plt.figure(1)
plt.plot(time_window_SAM, re_reference_SAM[seconds_SAM[0] * FS: seconds_SAM[1]␣
 ↪* FS])
plt.title("Re-referenced signal - SAM")
plt.xlabel("Time(s)")
plt.ylabel("Value(uV)")
plt.show()
```


Recorded Signal - SAM
Trigger Signal - SAM
Reference Signal 1 - SAM
Reference Signal 2 - SAM
Average Averaged Reference - SAM
Value(uV)
Value (bool)
Time(s)



Different than for the ABRs, you should apply a filter with cut-off frequencies between 60 and 1000 Hz, and epoch your recordings between 100 ms and 500 ms after the trigger onset to maintain a steady-state response which is not influenced by the offset. You may have a look at the waveform in earlier starting windows too in case you want to observe the onset response.

(1) Plot the signal before and after filtering in separate figures with proper titles. You can use plt.subplot. (2) After epoching, subtract the mean of each epoch from the same epoch to apply the baseline correction.

(2) Plot the extracted epochs. To do it, first, define a time vector according to the duration of your epochs, i.e. 400 ms, convert it to miliseconds and then label the x-axis, properly. Title your plot as "400-ms epochs".

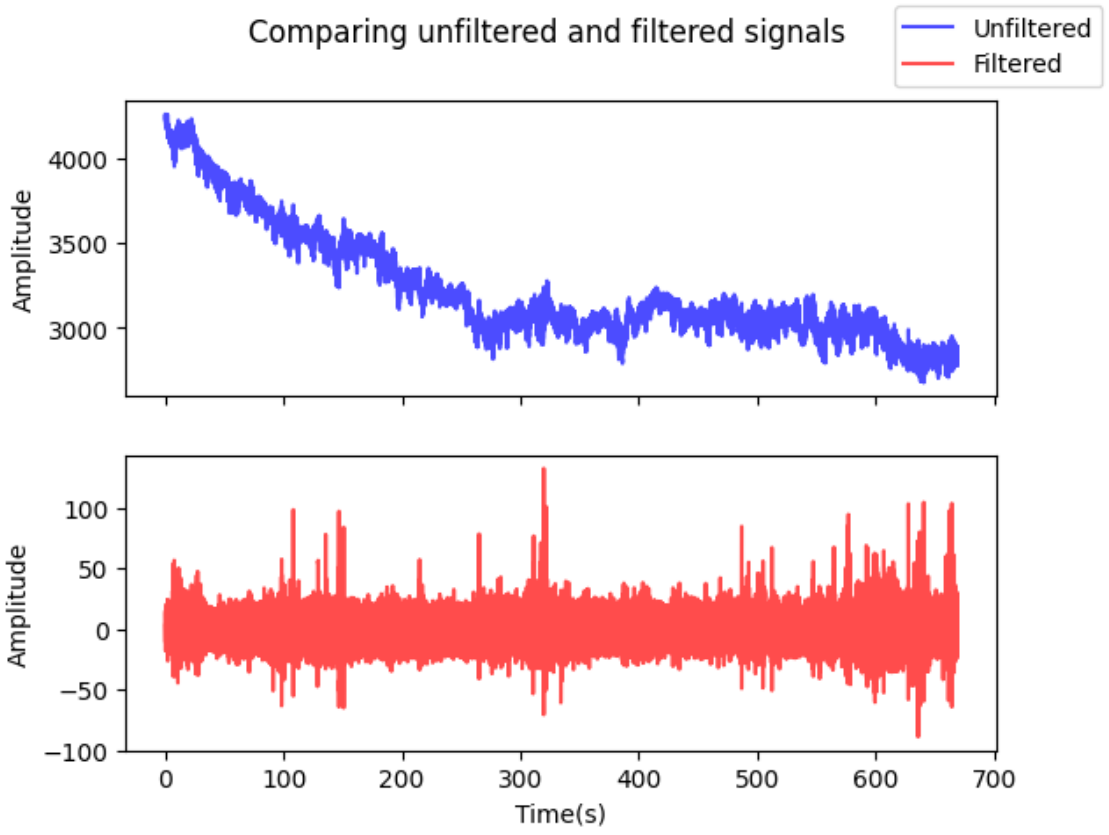
(3) Repeat the same procedure as the step (2-3), but with epochs of 500 ms, i.e. define your epochs between 0 to 500 ms after the trigger to see the onset response.

Note that you may need to transpose your variables (depending on how you have defined them) before plotting.

[88]:
```
#                                                    FILTERING THE SIGNAL
print(f"time_SAM: {len(re_reference_SAM) / FS_EFR}")
time_SAM = np.linspace(0, len(re_reference_SAM) / FS_EFR, len(re_reference_SAM))
```

```
# function for the cutoff filter
def filtering(signal, order, FS, lowcut, highcut, btype_filt):

    F1 = lowcut / (FS/2)
    F2 = highcut / (FS/2)

    b, a = sig.butter(order, [F1, F2], btype=btype_filt)
    Filtered_sig = sig.filtfilt(b, a, signal)

    return Filtered_sig


Filtered_sig_SAM = filtering(re_reference_SAM, 4, FS_EFR, 60, 1000, 'band')


# plotting original signal and re_reference_SAMd
fig, ax = plt.subplots(2,1,sharex=True)

ax[0].plot(time_SAM, re_reference_SAM, alpha=0.7, color='b', label="Unfiltered")
ax[0].set_ylabel("Amplitude")

ax[1].plot(time_SAM, Filtered_sig_SAM, color='r', label="Filtered", alpha=0.7)
ax[1].set_ylabel("Amplitude")

fig.suptitle("Comparing unfiltered and filtered signals")
fig.legend()

plt.xlabel("Time(s)")
plt.show()
```

```
time_SAM: 669.0
```

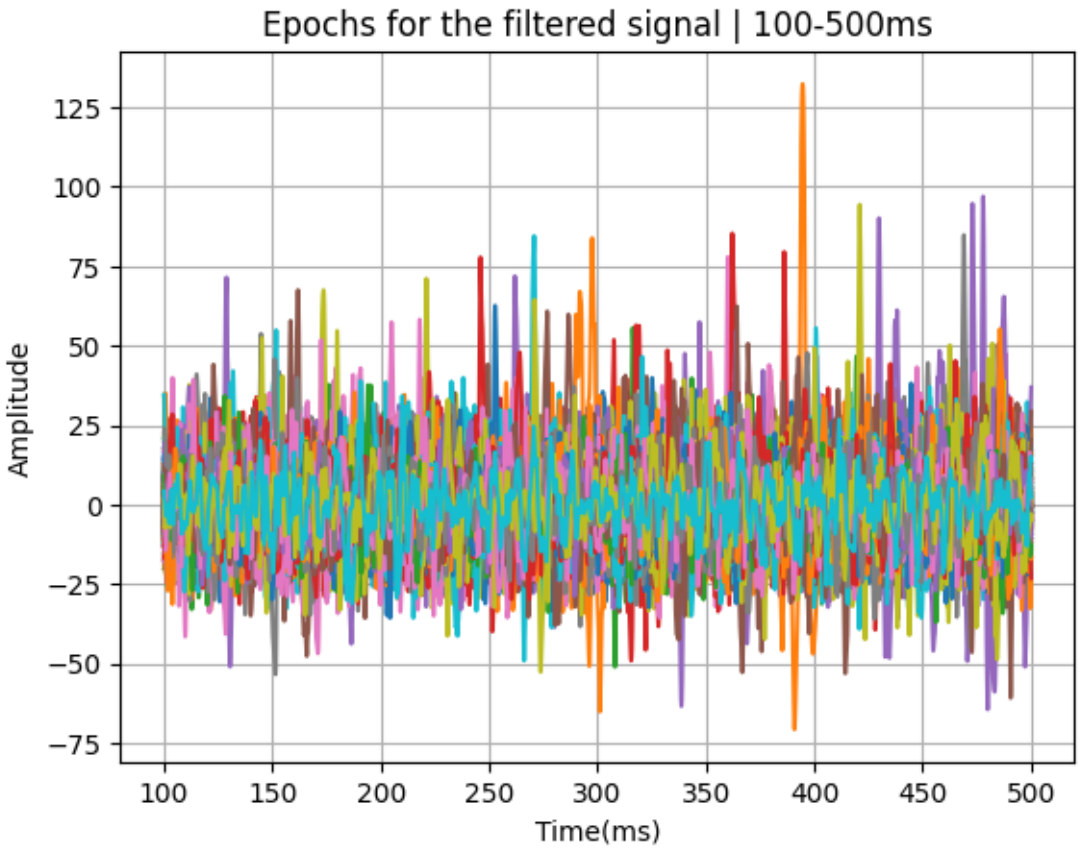


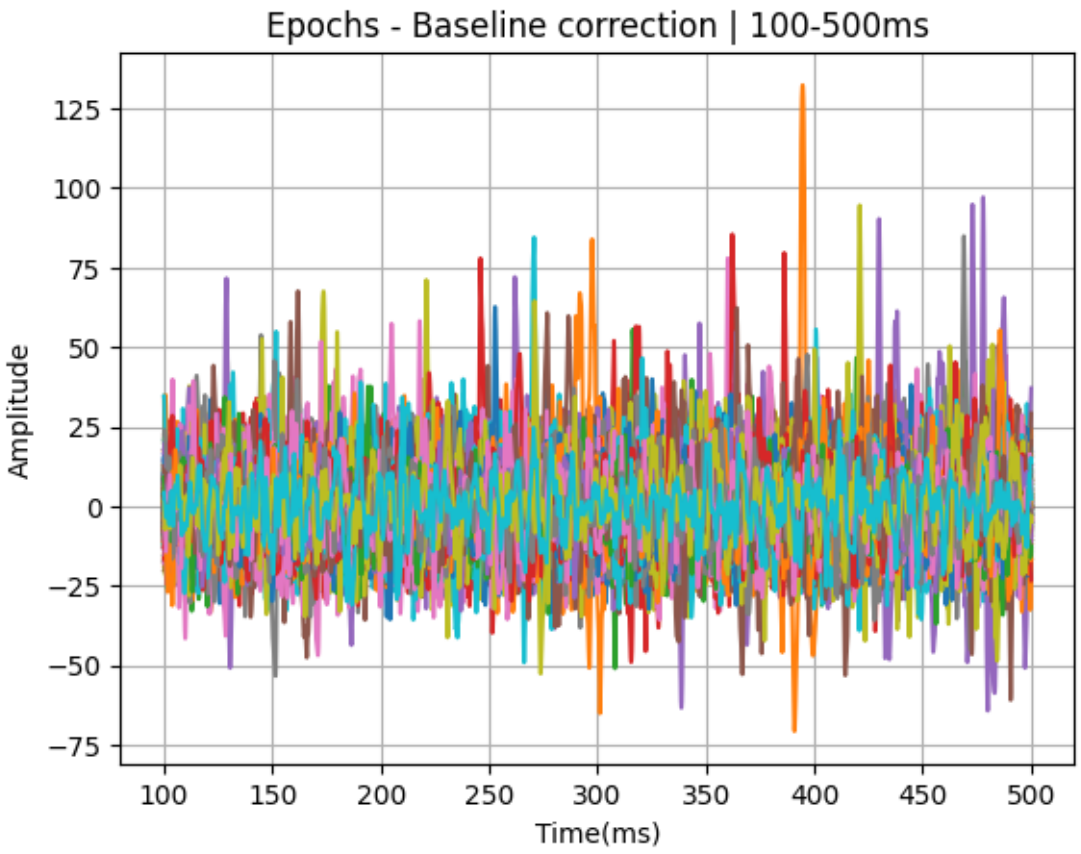
[89]:
```
#                                                          EPOCHS FOR SAM:␣
 ↪100-500ms
print(f"fs: {FS_EFR}")
FS_EFRm = FS_EFR / 1000
nsamples_epoch_100_500 = int(400*FS_EFRm)+1 # int is 6554
print(f"# of samples in an epoch: {nsamples_epoch_100_500}")

epochs_100_500 = np.zeros([len(indx_SAM), int(nsamples_epoch_100_500)]) # rows␣
 ↪x columns -> len(indx) x samples
time_epoch_100_500 = np.linspace(100, 500, nsamples_epoch_100_500)

print(f"shape epochs_100_500: {epochs_100_500.shape}") # 1000 triggers, 6554␣
 ↪samples in 400ms

# creating epochs_100_500 matrix
for i in range(len(indx_SAM)):
    click = indx_SAM[i]
    epochs_100_500[i] = Filtered_sig_SAM[click + int(100*FS_EFRm): click +␣
 ↪int(500*FS_EFRm)] # from (click + 100ms) to (click + 500ms)
    plt.plot(time_epoch_100_500, epochs_100_500[i])
```

```
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.title("Epochs for the filtered signal | 100-500ms")
plt.grid()
plt.show()


# baseline correction for each epoch
epochs_baseline_100_500 = [(epoch - np.mean(epoch)) for epoch in epochs_100_500]

# plotting every epoch (baseline)
for i in range(len(epochs_100_500)):
    plt.plot(time_epoch_100_500, epochs_baseline_100_500[i])

plt.title("Epochs - Baseline correction | 100-500ms")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()



#                                                       EPOCHS FOR SAM:␣
 ↪0-500ms
print(f"fs: {FS_EFR}")
FS_EFRm = FS_EFR / 1000
nsamples_epoch_0_500 = int(500*FS_EFRm) # int is 8192
print(f"# of samples in an epoch: {nsamples_epoch_0_500}")

epochs_0_500 = np.zeros([len(indx_SAM), int(nsamples_epoch_0_500)]) # rows x␣
 ↪columns -> len(indx) x samples
time_epoch_0_500 = np.linspace(0, 500, nsamples_epoch_0_500)

print(f"shape epochs_0_500: {epochs_0_500.shape}") # 1000 triggers, 8192␣
 ↪samples in 25ms

# creating epochs_0_500 matrix
for i in range(len(indx_SAM)):
    click = indx_SAM[i]
    epochs_0_500[i] = Filtered_sig_SAM[click: click + int(500*FS_EFRm)] # from␣
 ↪click to (click + 500ms)
    plt.plot(time_epoch_0_500, epochs_0_500[i])
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.title("Epochs for the filtered signal | 0-500ms")
plt.grid()
plt.show()
```

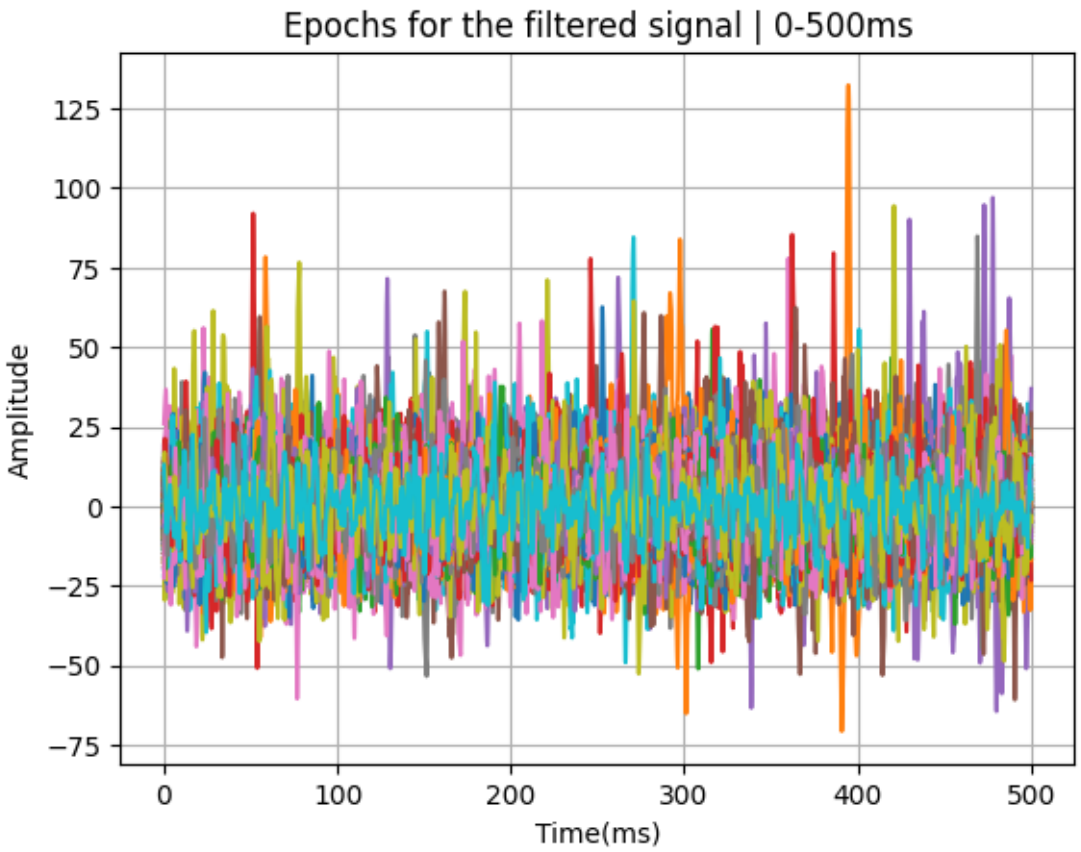
```
# baseline correction for each epoch
epochs_baseline_0_500 = [(epoch - np.mean(epoch)) for epoch in epochs_0_500]

# plotting every epoch (baseline)
for i in range(len(epochs_0_500)):
    plt.plot(time_epoch_0_500, epochs_baseline_0_500[i])

plt.title("Epochs - Baseline correction | 0-500ms")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()
```

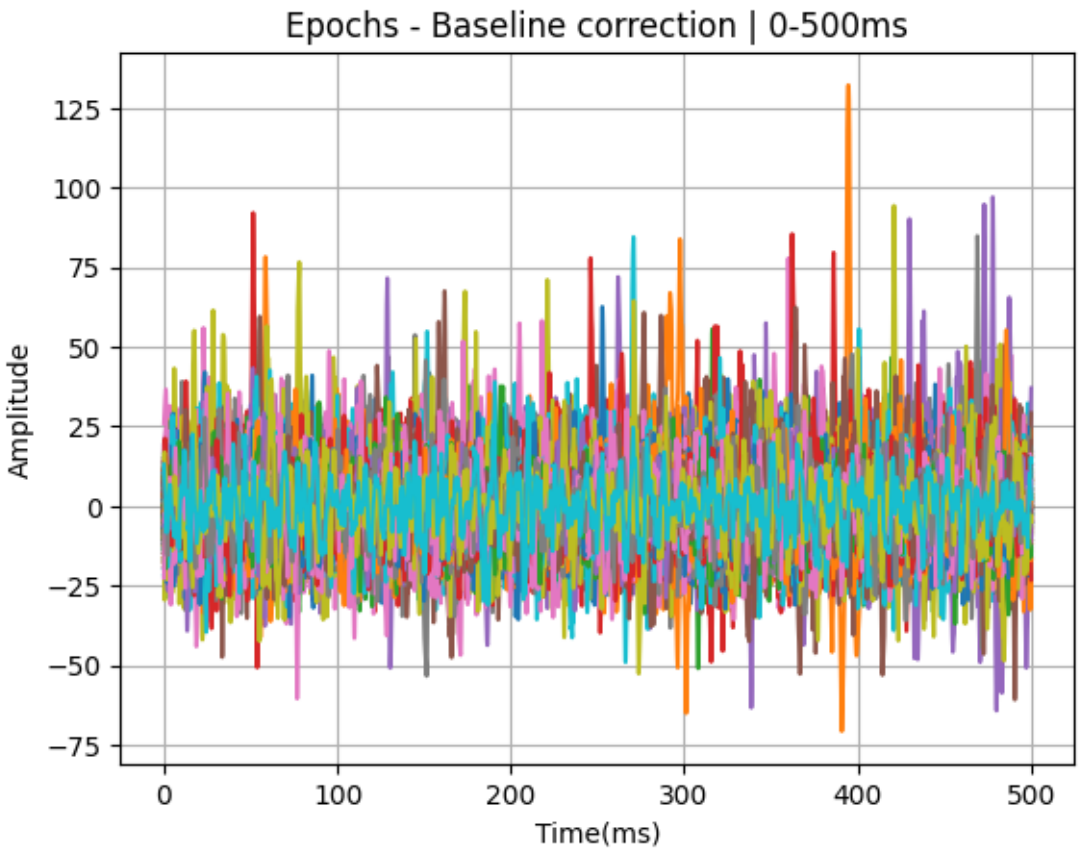
```
fs: 16384
# of samples in an epoch: 6554
shape epochs_100_500: (1000, 6554)
```






```
fs: 16384
# of samples in an epoch: 8192
shape epochs_0_500: (1000, 8192)
```


Epochs for the filtered signal | 0-500ms
Amplitude
125
100
75
50
25
0
−25
−50
−75
0
100
200
300
400
500
Time(ms)



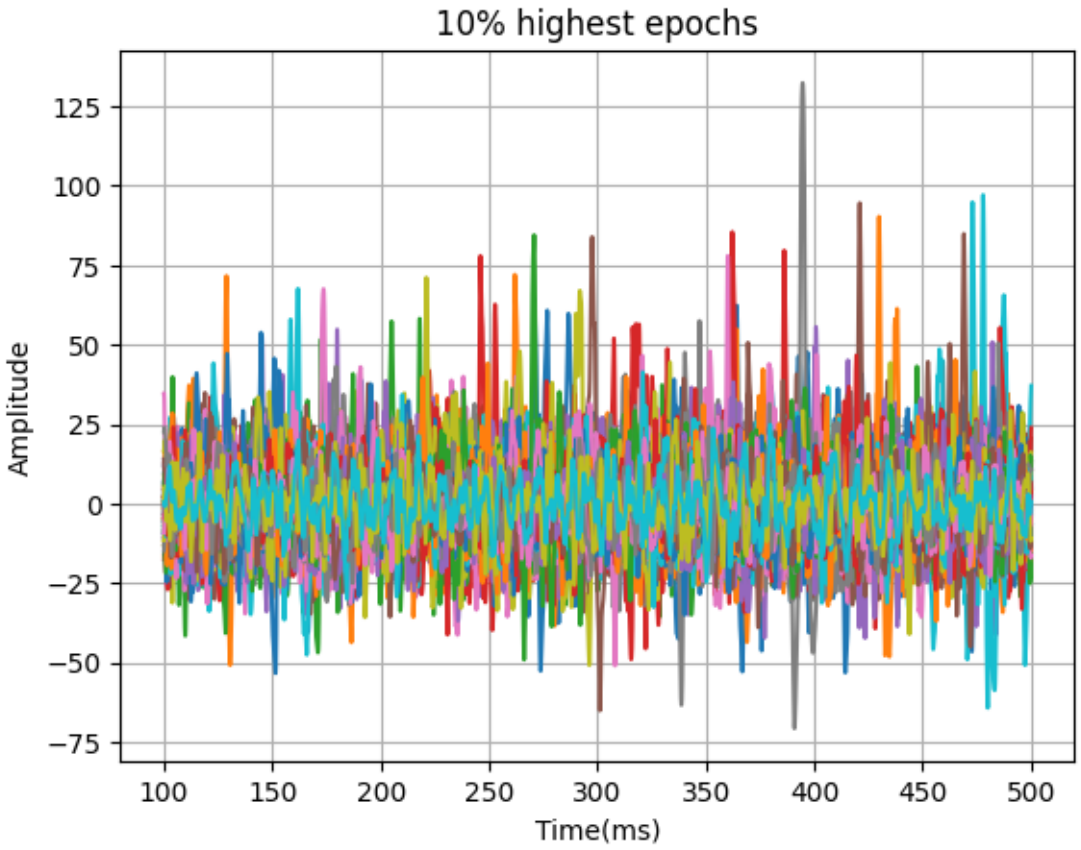
Have a look at the quality of your epochs, and make a piece of code which removes those epochs which have the 10% highest overall amplitude from both the positive and negative epochs, to make sure you remove an equal number of positive and negative epochs to remove epochs with artifacts. To implement it, use the 400-ms epochs in the previous section and drop the 10% of epochs of each polarity with the highest maximum values.

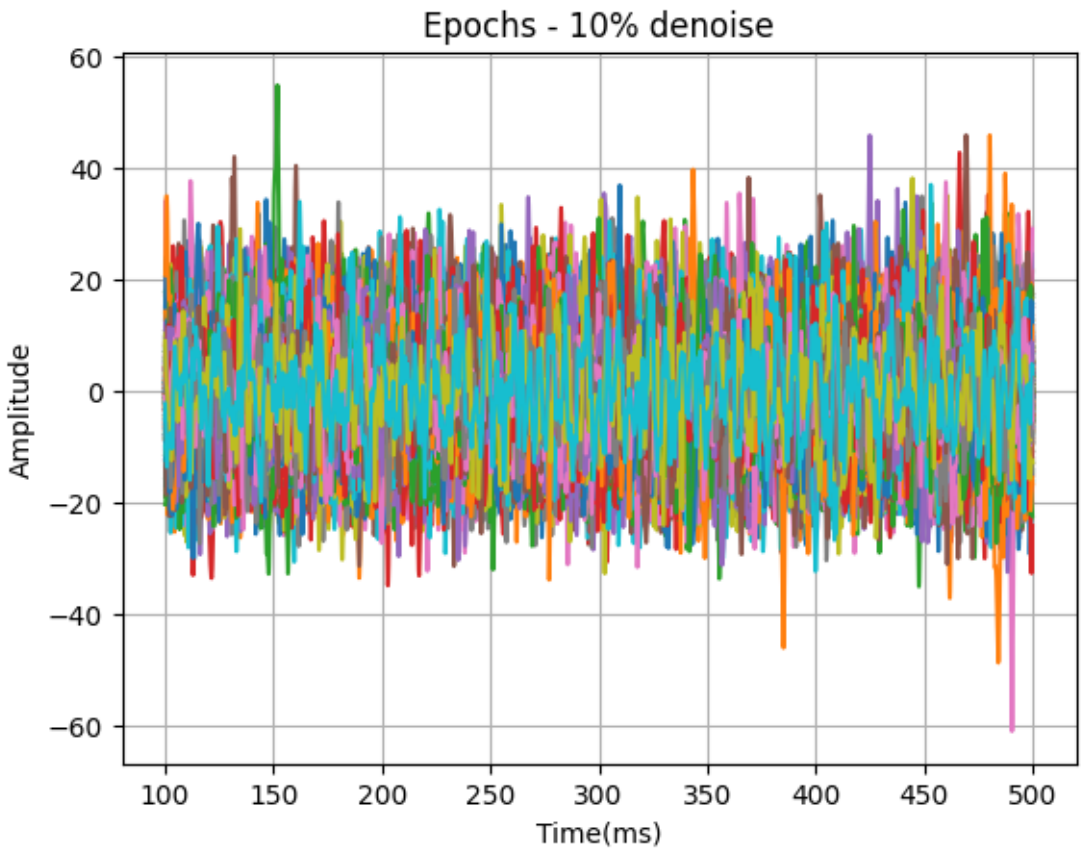
- Plot the sorted maximum values of epochs of each polarity in separate figures.
- Then remove the 10% of epochs with the highest amplitudes and plot the remained epochs maximum values on top of the figures of the previous step (Do it for both polarities).
- After removing the artifact-contaminated epochs, plot the artifact-free positive and negative epochs in separate figures with proper titles and x-axis labels. How would you interpret the figures? Do you think removing 10% of the noisy epochs was enough or still your epochs contain artifact-contaminated epochs?

[91]:
```
# Denoising algorithm
def denoise_SAM(epochs, denoise_size):
    n_epochs = np.array(epochs).shape[0] # 1000 epochs

    sum_epochs = np.zeros(n_epochs)
    # summation of every epoch to categorize into the 10% epochs with highest␣
 ↪abs amplitudes
```

```
    for i in range(n_epochs):
        sum_epochs[i] = np.sum(np.abs(epochs[i][:]))

    sorted_sum_epochs = np.array(np.argsort(sum_epochs))[::-1] # sort in␣
 ↪descending order -> want to see indices for the noisiest epochs
    to_delete = sorted_sum_epochs[ : int(n_epochs * denoise_size)] # indices of␣
 ↪10% noisiest epochs

    # removing noisy epochs
    denoised_epochs_SAM = []
    for i in range(n_epochs):
        if i in to_delete: # if the index is in the list of indices to delete,␣
 ↪just go to the next epoch
            continue
        denoised_epochs_SAM.append(epochs[i]) # if the epoch is not on the list␣
 ↪"to delete" then it's a good value, append it

    return to_delete, denoised_epochs_SAM


sorted_max_values_SAM, denoised_epochs_SAM =␣
 ↪denoise_SAM(epochs_baseline_100_500, 0.1)

# plotting the max 10% values
for indx_sort in sorted_max_values_SAM:
    plt.plot(time_epoch_100_500, epochs_baseline_100_500[indx_sort])
plt.title("10% highest epochs")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()


# plotting every denoised epoch
for i in range(len(denoised_epochs_SAM)):
    plt.plot(time_epoch_100_500, denoised_epochs_SAM[i])
plt.title("Epochs - 10% denoise")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()

print(f"epochs after denoising - 10%: {np.array(denoised_epochs_SAM).shape}") #␣
 ↪900 remaining epochs

"""
```

```
In the case of this signal, 10% is not enough for denoising it, as there are␣
 ↪some peaks that peek outside of the general tendency of the amplitude. This␣
 ↪will result in
a noisier signal and the epochs' data may be more difficult to interpret␣
 ↪accurately.
"""
```



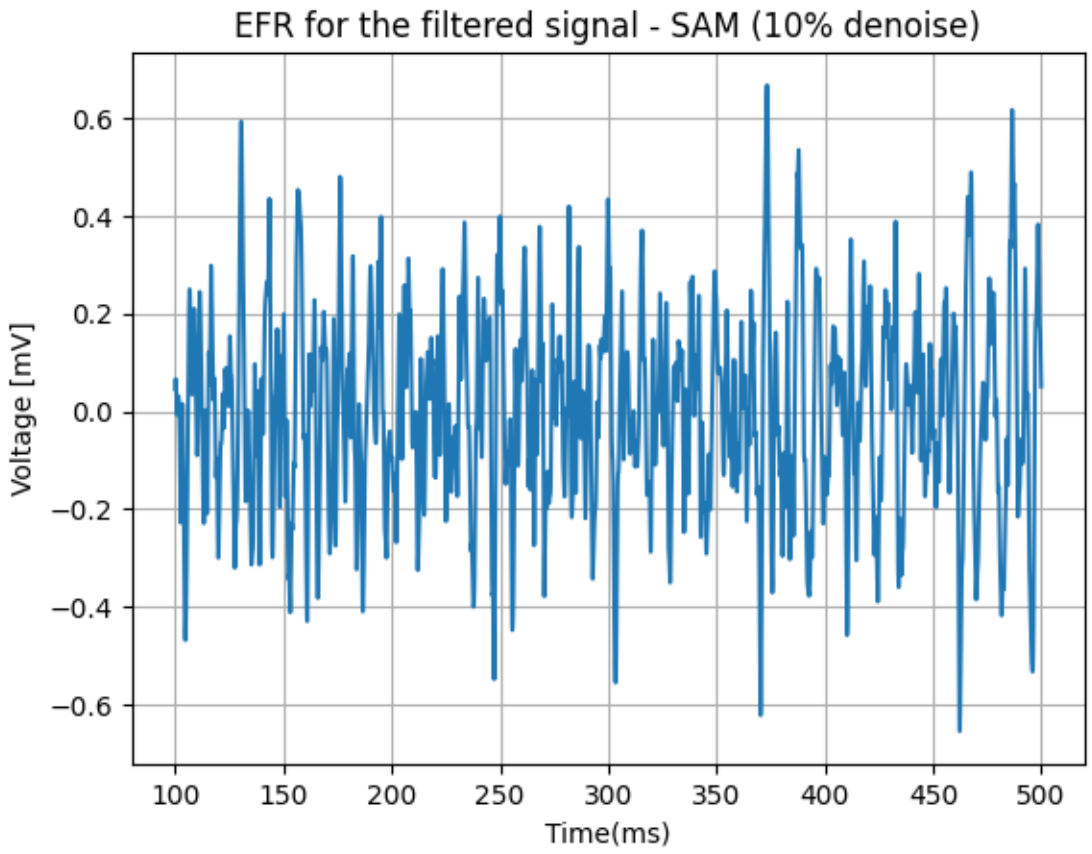



```
epochs after denoising - 10%: (900, 6554)
```

[91]:
```
"\nIn the case of this signal, 10% is not enough for denoising it, as there are
some peaks that peek outside of the general tendency of the amplitude. This will
result in\na noisier signal and the epochs' data may be more difficult to
interpret accurately.\n"
```

Next, concatenate the positive and the negative epochs, and average the response across all epochs. This is the EFR time-domain waveform, make a time-axis and plot the result (t=0 should correspond to the trigger time).

[92]:
```
EFR_SAM = np.mean(denoised_epochs_SAM, axis=0)

plt.plot(time_epoch_100_500, EFR_SAM)
plt.title('EFR for the filtered signal - SAM (10% denoise)')
plt.xlabel('Time(ms)')
plt.ylabel('Voltage [mV]')
plt.grid()
plt.show()
```

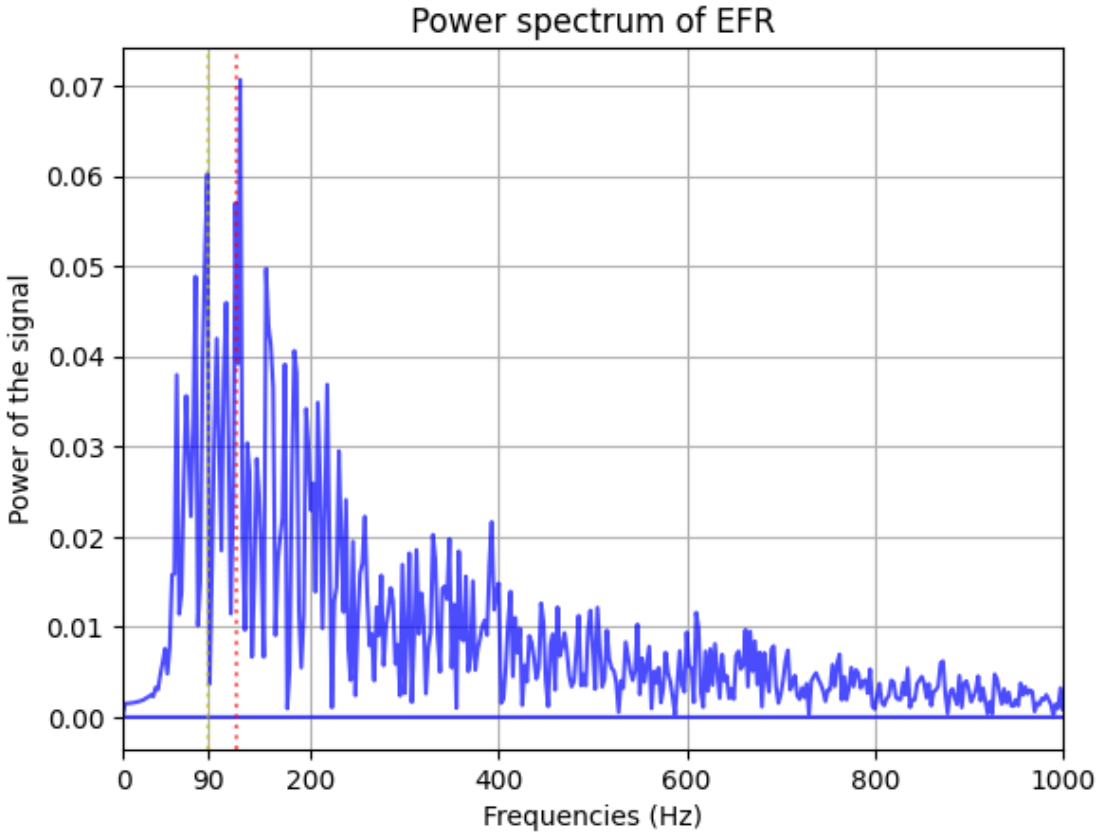


Now plot the power spectrum of the EFR (2*abs(fft(X))/length(X)), make a frequency axis and if necessary plot the 20log10 response. After plotting the spectrum, limit your frequency axis to the range of [0-1000] Hz.

Do you notice a peak at 120 Hz? Are there other spectral peaks observed? The size of the spectral peak can be identified as the EFR strength can be related to the fidelity with which the auditory brainstem can track stimulus envelopes. Ageing listeners and listeners with hearing damage may have weaker responses because of synaptopathy (a reduction in the number of auditory-nerve fibers).

[93]:
```
pow_EFR_SAM = 2 * np.abs(np.fft.fft(EFR_SAM)) / len(EFR_SAM)

window_length_EFR_SAM = EFR_SAM.size
sample_rate = 1/FS
freq_EFR_SAM = np.fft.fftfreq(window_length_EFR_SAM, sample_rate)

# plotting power spectrum of EFR
fig = plt.figure()
plt.plot(freq_EFR_SAM, pow_EFR_SAM, color='b', alpha=0.7)
plt.xlim([0, 1000])
x_ticks = np.append(plt.xticks()[0], 120)
x_ticks = np.append(plt.xticks()[0], 90)
```

```
plt.axvline(x=120, color='r', label="120Hz", alpha=0.6, ls='dotted')
plt.axvline(x=90, color='y', label="90Hz", alpha=0.6, ls='dotted')
plt.xticks(x_ticks)
plt.title('Power spectrum of EFR')
plt.xlabel("Frequencies (Hz)")
plt.ylabel('Power of the signal')
plt.grid()
plt.show()

"""
The two highest peaks are at 120Hz and 90Hz
"""
```



[93]: '\nThe two highest peaks are at 120Hz and 90Hz \n'

The last recording corresponds to the envelope-following response (EFR) of a word extracted from the Flemish Speech Matrix, to show how EFRs can be extracted for more complex stimuli. The word “David” was used and was monotonized so that its fundamental frequency (F0) is constant over time at ~220 Hz. The stimulus was then high-pass filtered with a cutoff frequency of 1650 Hz and calibrated at 70 dB SPL. For the recording, the stimulus was cut to 500 ms and was

repeated 1200 times (600 positive and 600 negative epochs). The EEG response to such a stimulus is expected to follow the fundamental frequency of the original speech.

The same procedure (as for the modulated tone) can be followed to extract the Cz channel, reference channels and trigger signals, and re-reference the Cz channel recording to the average of the reference channels. For your convenience you can plot everything in the same way as before. The same labelling is followed:

1. david_EFR_Cz: recorded signal from the electrode placed on the top of head
2. david_EFR_ref1: recorded signal from the left earlobe electrode
3. david_EFR_ref2: recorded signal from the right earlobe electrode
4. Trigsdavid_EFR: trigger signal which indicates the onset of each epoch by "1". Odd epochs correspond to positive polarity stimuli and even epochs to negative polarity.

[94]:
```
#Load in the data
david_data = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"david_EFR_Cz.mat"))
david_EFR_ref1 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"david_EFR_ref1.mat"))
david_EFR_ref2 = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"david_EFR_ref2.mat"))
david_EFR_trig = scipy.io.loadmat(os.path.join("C:
 ↪\\Users\\cesar\\Documents\\UGent\\2022_2023\\Second_Term\\Auditory_computation\\EEG_data\\2
 ↪"Trigsdavid_EFR.mat"))

FS_EFR = 16384 # davidpling frequency EFR
Sig_david = david_data['Ch'] #has the raw recording trace of EEG data
ref1_david = david_EFR_ref1['Ch']
ref2_david = david_EFR_ref2['Ch']


T_david = david_EFR_trig['Trigs']

#find all the triggers
indx_david = np.where(np.diff(T_david)==1)[1]+1

avg_ref1_ref2_david = [(abs(ref1_david[0][i] - ref2_david[0][i]) / 2) for i in␣
 ↪range(len(ref1_david[0]))]
re_reference_david = [(Sig_david[0][i] - avg_ref1_ref2_david[i]) for i in␣
 ↪range(len(ref1_david[0]))]

print(david_data.keys())
```

```
dict_keys(['__header__', '__version__', '__globals__', 'Ch'])
```

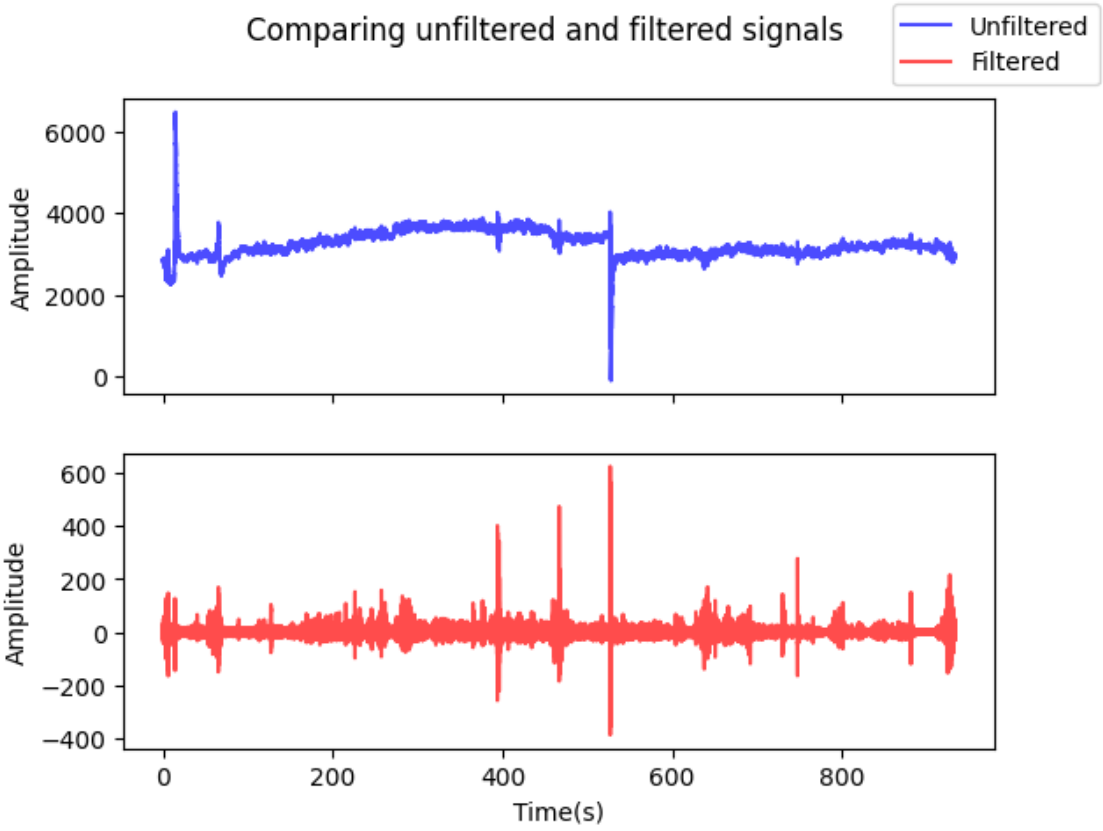
The same filtering can be applied as in the previous case (60-1000 Hz). Since the stimulus used is

not periodic over time, the whole 500 ms response after the trigger onset needs to be used here. Follow the same procedure to:

(1) Plot the signal before and after filtering.
(2) Epoch the signal and subtract the mean of each epoch from the same epoch to apply the baseline correction.
(3) Plot the extracted 500-ms epochs.

[95]:
```
#                                                    FILTERING THE SIGNAL
print(f"time_david: {len(re_reference_david) / FS_EFR}")
time_david = np.linspace(0, len(re_reference_david) / FS_EFR,␣
 ↪len(re_reference_david))

# function for the cutoff filter
def filtering(signal, order, FS, lowcut, highcut, btype_filt):

    F1 = lowcut / (FS/2)
    F2 = highcut / (FS/2)

    b, a = sig.butter(order, [F1, F2], btype=btype_filt)
    Filtered_sig = sig.filtfilt(b, a, signal)

    return Filtered_sig


Filtered_sig_david = filtering(re_reference_david, 4, FS_EFR, 60, 1000, 'band')


# plotting original signal and re_reference_davidd
fig, ax = plt.subplots(2,1,sharex=True)

ax[0].plot(time_david, re_reference_david, alpha=0.7, color='b',␣
 ↪label="Unfiltered")
ax[0].set_ylabel("Amplitude")

ax[1].plot(time_david, Filtered_sig_david, color='r', label="Filtered", alpha=0.
 ↪7)
ax[1].set_ylabel("Amplitude")

fig.suptitle("Comparing unfiltered and filtered signals")
fig.legend()

plt.xlabel("Time(s)")
plt.show()
```

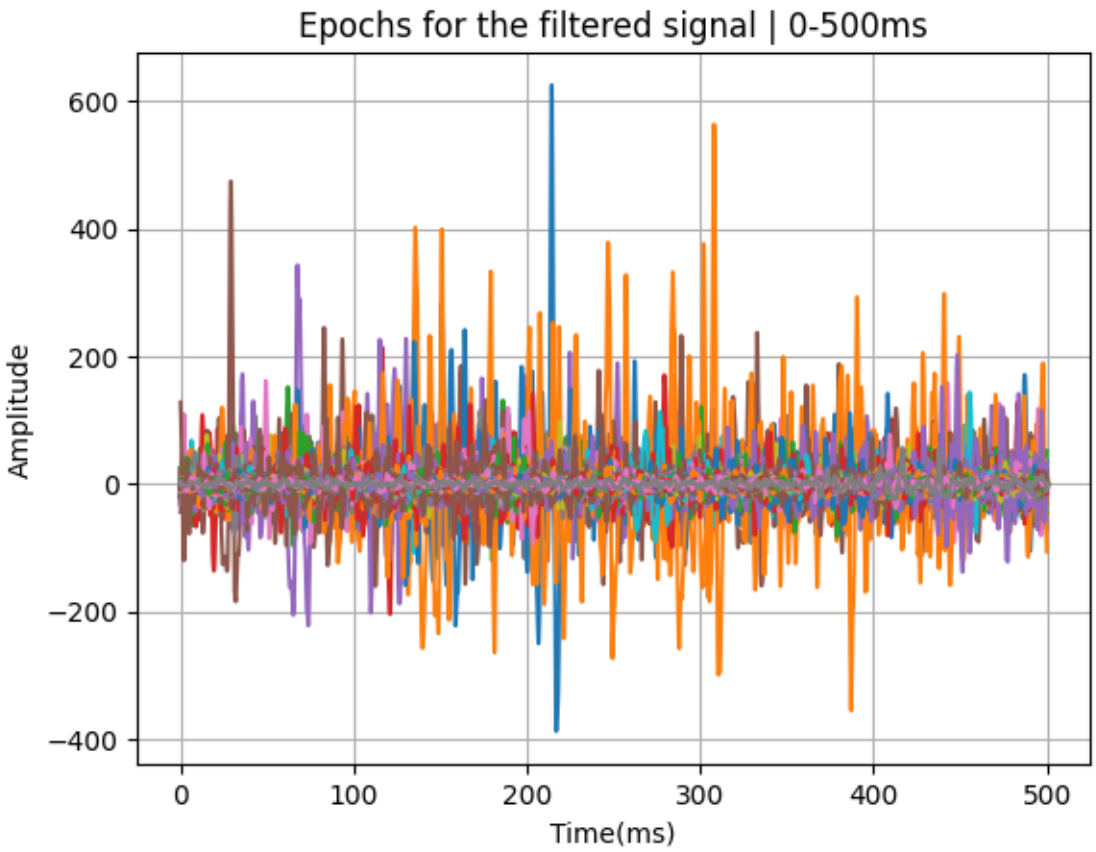
```
time_david: 933.0
```

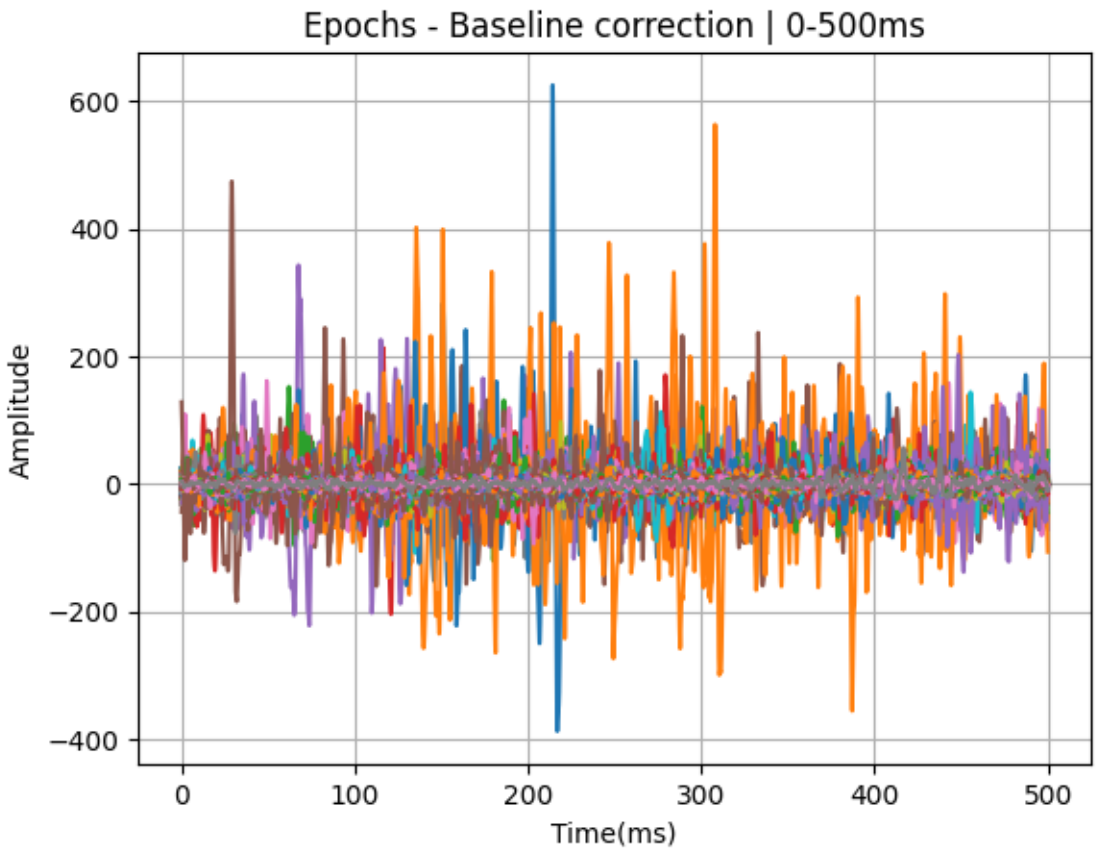


[96]:
```
#                                                    EPOCHS FOR DAVID:
  ↪0-500ms
print(f"fs: {FS_EFR}")
FS_EFRm = FS_EFR / 1000
nsamples_epoch_0_500 = int(500*FS_EFRm) # int is 8192
print(f"# of samples in an epoch: {nsamples_epoch_0_500}")

epochs_0_500 = np.zeros([len(indx_david), int(nsamples_epoch_0_500)]) # rows x
  ↪columns -> len(indx) x samples
time_epoch_0_500 = np.linspace(0, 500, nsamples_epoch_0_500)

print(f"shape epochs_0_500: {epochs_0_500.shape}") # 1000 triggers, 8192
  ↪samples in 25ms

# creating epochs_0_500 matrix
for i in range(len(indx_david)):
    click = indx_david[i]
    epochs_0_500[i] = Filtered_sig_david[click: click + int(500*FS_EFRm)] #
  ↪from click to (click + 500ms)
    plt.plot(time_epoch_0_500, epochs_0_500[i])
```

```
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.title("Epochs for the filtered signal | 0-500ms")
plt.grid()
plt.show()


# baseline correction for each epoch
epochs_baseline_0_500 = [(epoch - np.mean(epoch)) for epoch in epochs_0_500]

# plotting every epoch (baseline)
for i in range(len(epochs_0_500)):
    plt.plot(time_epoch_0_500, epochs_baseline_0_500[i])

plt.title("Epochs - Baseline correction | 0-500ms")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()
```

```
fs: 16384
# of samples in an epoch: 8192
shape epochs_0_500: (1208, 8192)
```

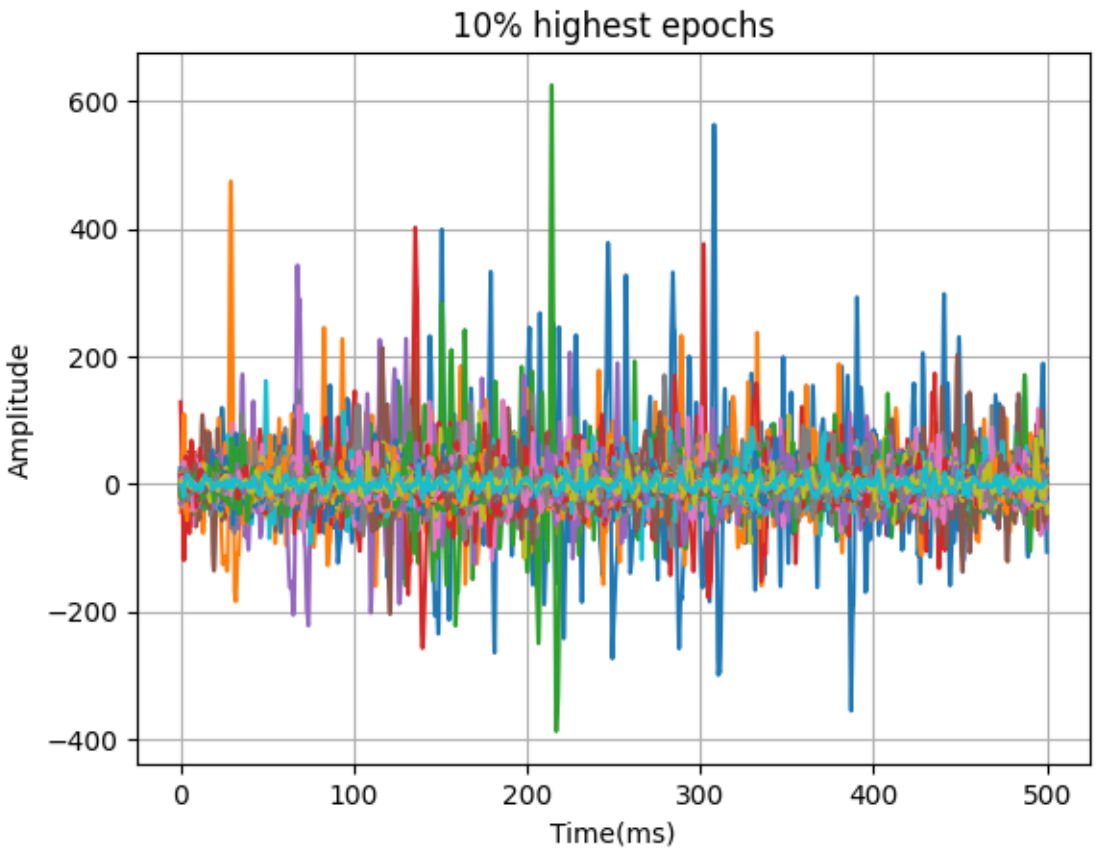

Epochs for the filtered signal | 0-500ms
Amplitude
600
400
200
0
−200
−400
0
100
200
300
400
500
Time(ms)

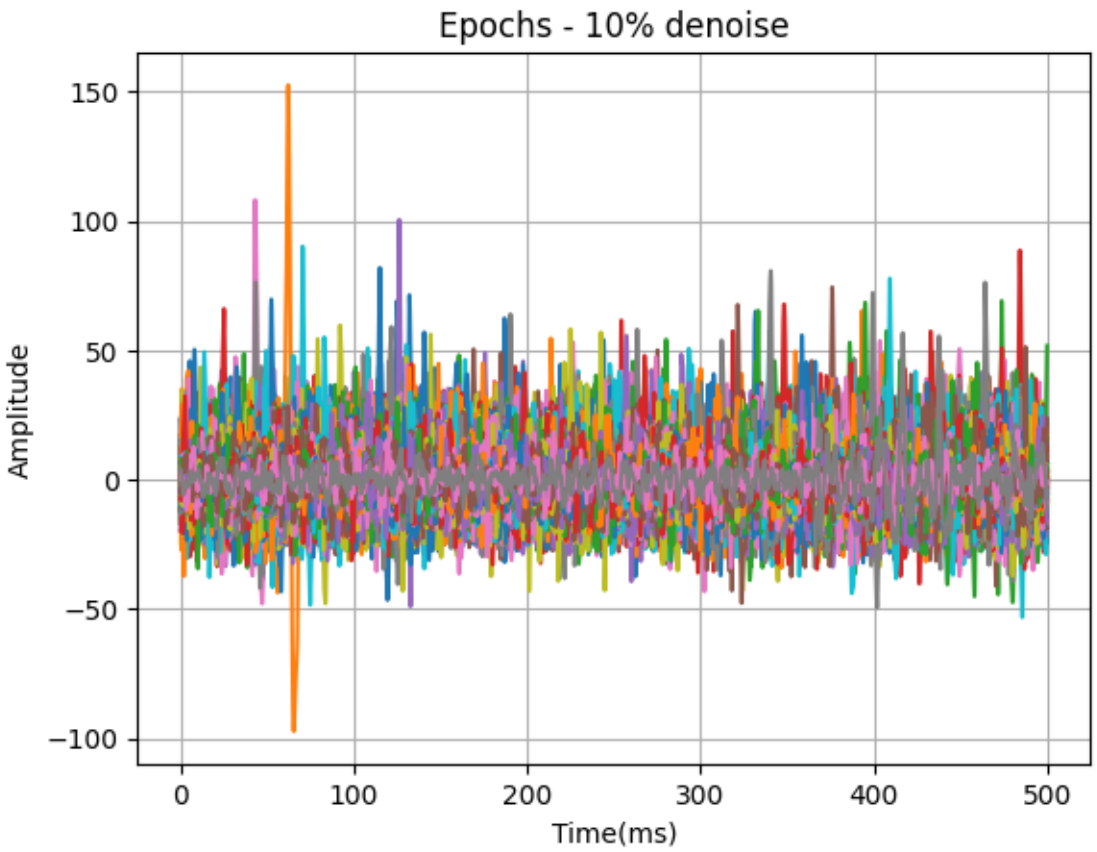


- Based on the triggers, separate the positive and negative polarity epochs and plot the sorted maximum values of epochs of each polarity in separate figures.
- Remove the 10% of epochs with the highest amplitudes and plot the remained epochs maximum values on top of the figures of the previous step (for both polarities).
- After removing the artifact-contaminated epochs, plot the artifact-free positive and negative epochs in separate figures.

[97]:
```
#                                                                 Denoising␣
 ↪algorithm
def denoise_SAM(epochs, denoise_size):
    n_epochs = np.array(epochs).shape[0] # 1000 epochs

    sum_epochs = np.zeros(n_epochs)
    # summation of every epoch to categorize into the 10% epochs with highest␣
 ↪abs amplitudes
    for i in range(n_epochs):
        sum_epochs[i] = np.sum(np.abs(epochs[i][:]))

    sorted_sum_epochs = np.array(np.argsort(sum_epochs))[::-1] # sort in␣
 ↪descending order -> want to see indices for the noisiest epochs
```

```
    to_delete = sorted_sum_epochs[ : int(n_epochs * denoise_size)] # indices of␣
 ↪10% noisiest epochs

    # removing noisy epochs
    denoised_epochs_SAM = []
    for i in range(n_epochs):
        if i in to_delete: # if the index is in the list of indices to delete,␣
 ↪just go to the next epoch
            continue
        denoised_epochs_SAM.append(epochs[i]) # if the epoch is not on the list␣
 ↪"to delete" then it's a good value, append it

    return to_delete, denoised_epochs_SAM


sorted_max_values_david, denoised_epochs_david =␣
 ↪denoise_SAM(epochs_baseline_0_500, 0.1)

# plotting the max 10% values
for indx_sort in sorted_max_values_david:
    plt.plot(time_epoch_0_500, epochs_baseline_0_500[indx_sort])
plt.title("10% highest epochs")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()


# plotting every denoised epoch
for i in range(len(denoised_epochs_david)):
    plt.plot(time_epoch_0_500, denoised_epochs_david[i])
plt.title("Epochs - 10% denoise")
plt.xlabel("Time(ms)")
plt.ylabel("Amplitude")
plt.grid()
plt.show()

print(f"epochs after denoising - 10%: {np.array(denoised_epochs_david).shape}")␣
 ↪# 900 remaining epochs
```

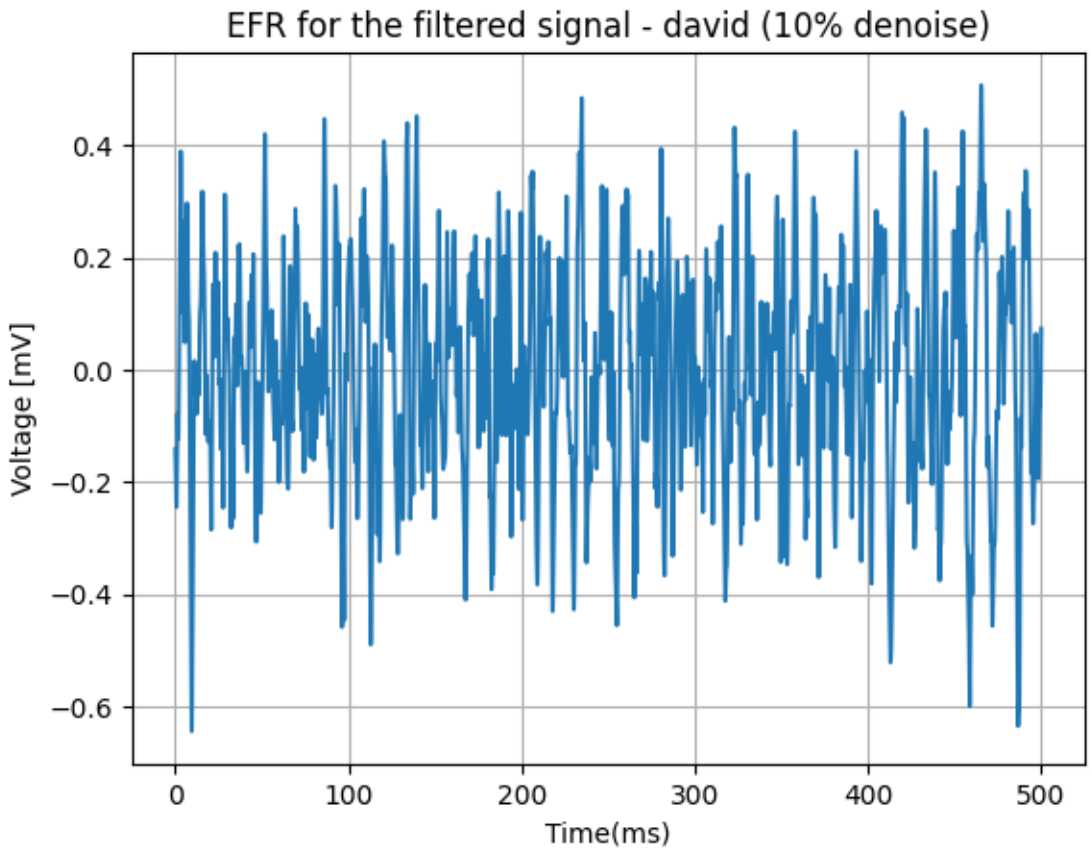

10% highest epochs
600
400
200
0
−200
−400
Amplitude
0
100
200
300
400
500
Time(ms)




```
epochs after denoising - 10%: (1088, 8192)
```

Next, concatenate the positive and the negative epochs and average the response across all epochs to get the EFR time-domain waveform.

[98]:
```
EFR_david = np.mean(denoised_epochs_david, axis=0)

plt.plot(time_epoch_0_500, EFR_david)
plt.title('EFR for the filtered signal - david (10% denoise)')
plt.xlabel('Time(ms)')
plt.ylabel('Voltage [mV]')
plt.grid()
plt.show()
```

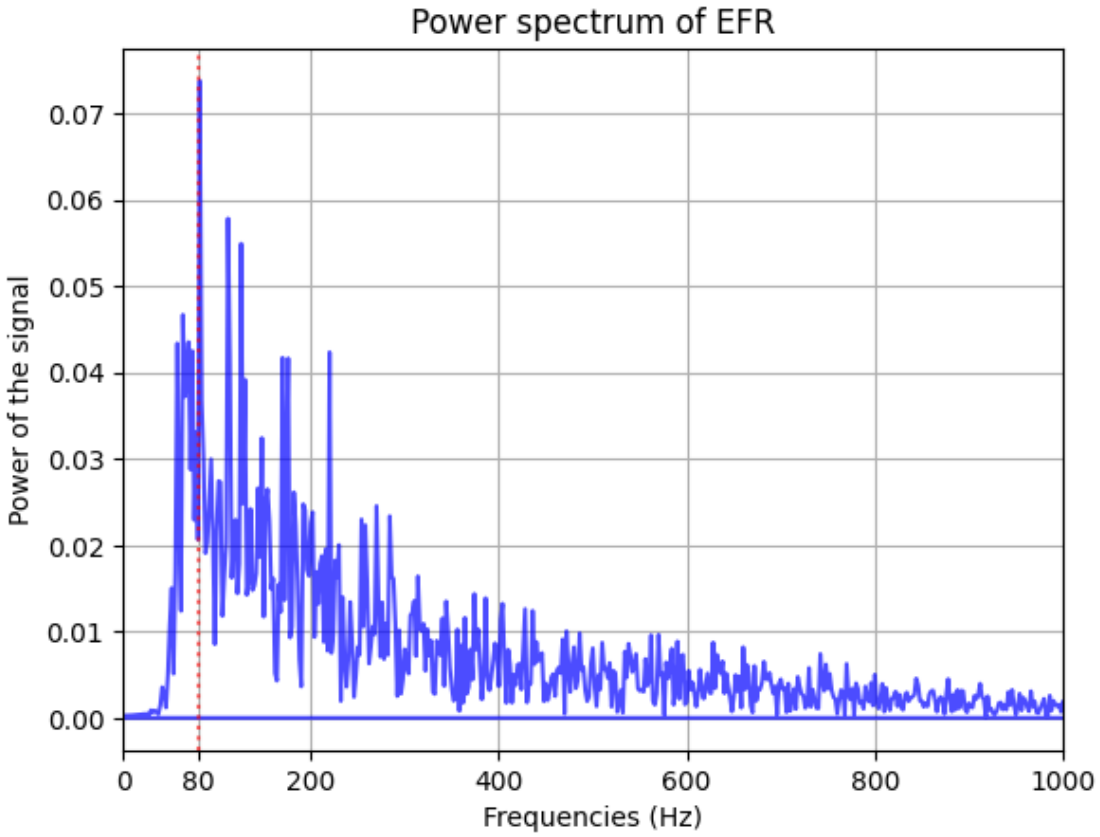


Now plot the power spectrum of the EFR (2*abs(fft(X))/length(X)) for the averaged signal across all epochs. After plotting the spectrum, limit your frequency axis to the range of [0-1000] Hz.

Are there any spectral peaks observed that are related to the speech stimulus in your opinion?

[99]:
```
pow_EFR_david = 2 * np.abs(np.fft.fft(EFR_david)) / len(EFR_david)

window_length_EFR_david = EFR_david.size
sample_rate = 1/FS
freq_EFR_david = np.fft.fftfreq(window_length_EFR_david, sample_rate)

# plotting power spectrum of EFR
fig = plt.figure()
plt.plot(freq_EFR_david, pow_EFR_david, color='b', alpha=0.7)
plt.xlim([0, 1000])
x_ticks = np.append(plt.xticks()[0], 80)
plt.axvline(x=80, color='r', label="80Hz", alpha=0.6, ls='dotted')
plt.xticks(x_ticks)
plt.title('Power spectrum of EFR')
plt.xlabel("Frequencies (Hz)")
plt.ylabel('Power of the signal')
```

```
plt.grid()
plt.show()


'''
There is a noticeable number of peaks in the range of 100 to 250ms, which falls␣
 ↪in the range of peaks that can represent speech recognition.
'''
```



[99]: '\nThere is a noticeable number of peaks in the range of 100 to 250ms, which falls in the range of peaks that can represent speech recognition.\n'