

SC42150_PythonAssignment2_Denoising

October 29, 2021

1 SC42150 - Statistical Signal Analysis

1.1 Python Assignment II: Denoising with an FIR filter.

Team members

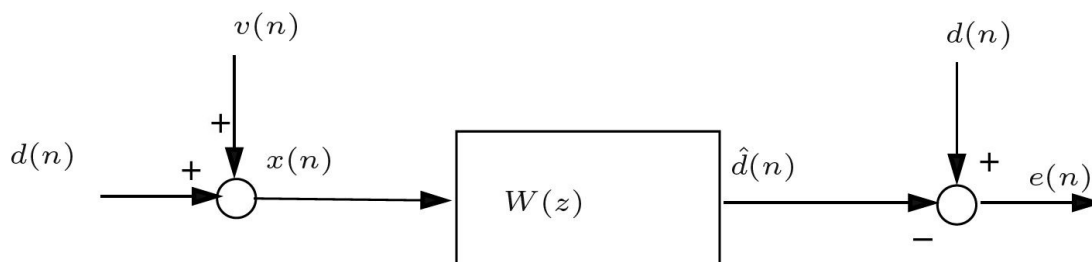
- Abina Boesjes (5094011):
-

1.2 Luca de Laat (4135040):

1.3 Introduction

You have recently gotten involved in amateur RF communication, and you have finally managed to transmit a signal wirelessly! Your setup consists of one broadcasting antenna and three receiving ones, placed at different distances from the emitter, specifically 5cm, 100cm, and 200cm. The signal you sent was a short recording of a piano note. Upon receiving it however, you realise that it has been corrupted by noise. So now you want to apply what you have learned in the Statistical Signal Analysis course and try to remove the noise. Since the nature of the interference is stochastic, you want to subtract the effects of the distortion by using the theory of Wiener filters.

In the configuration of the figure below, all signals are discrete-time and real. The corrupted signal $x(n) = d(n) + v(n)$ contains both the signal of interest $d(n)$ and a noise $v(n)$ which is *uncorrelated* to $d(n)$. In this assignment you will design a FIR filter $W(z)$, such that we can estimate the ground truth signal from $x(n)$ and from the statistical properties of the noise.



1.4 Reporting

The report of this python exercise consists of this Jupyter Notebook file, with your answers added after each question. Answers will consist of code and/or text (markdown) cells. Please use the

already indicated cells and format. You will **also** need to deliver a PDF version of this file.

In order to create a PDF version of this file, go to the top left, click on **File** then **Export Notebook As** and Choose **PDF**. If this feature is not supported by your system, you can also choose **LaTeX**. Then, you can use your preferred **LaTeX** compiler to create the PDF. After generation you PDF report, please double check if all answers/plots are correctly displayed.

Moreover, you will also need to send the **.mat** files with the signals used in this exercise. Don't forget to edit your team member's name and student number above. You will submit one Jupyter notebook per team.

1.4.1 Loading files

As mentioned before, you have three recordings available in the form of **piano_xxx.mat** files, one for each receiver distance (5cm, 100cm, 200cm). Furthermore, you have obtained a recording of the passive noise present in the system (**noise.mat**). Run this section in order to load the **.mat** files with the noise audios and the background noise signals.

```
[1]: # packages
from time import sleep
import numpy as np
import sounddevice as sd
import matplotlib.pyplot as plt

import scipy.io as sio
import scipy.linalg as lin
import scipy.signal as sig

# use this function to play the signals
# be careful when using headphones!
def play_signal(signal):
    sd.play(signal.astype("int16"), 22050)

# modeling distance in centimeters (cm)
distances = np.array([5.0, 100.0, 200.0])

# number of experiments with different distances
N_exp = distances.shape[0]

# print( f'Number of experiments = {N_exp}' )
print("experiment number", N_exp)

# list of files for all distance experiments with audio
filenames = ["piano_5cm.mat", "piano_100cm.mat", "piano_200cm.mat"]
titles = ["5 cm", "100 cm", "200 cm"]

# Number of data points
```

```

N = len(sio.loadmat(filenamees[0])["audio"][0])

# create array with received data
data_noised = np.zeros((N_exp, N), dtype=float)
for iexp in range(N_exp):
    data_noised[iexp] = sio.loadmat(filenamees[iexp])["audio"][0]

# array for the time steps
time_steps = np.linspace(0, N - 1, N).astype(int)
#print( f'Number of recording time steps = {N}' )
print("number of recording time steps", N)
# load background noise signal that was measured

filename_noise = "noise.mat"
data_mat = sio.loadmat(filename_noise)["audio"][0]

Nnoise = len(data_mat)
noise_measured = np.array(data_mat, dtype=float)

# print( f'Number of noise time steps = {Nnoise}' )
print("number of noise time steps", Nnoise)

```

```

experiment number 3
number of recording time steps 65411
number of noise time steps 88200

```

```

[2]: # play_signal(noise_measured)
      # play_signal(data_noised[0])
      # play_signal(data_noised[1])
      # play_signal(data_noised[2])

```

2 Exercises

This part should be handled after all audio files were uploaded into `data_noised` array and the noise was uploaded into `noise_measured` file.

2.1 Question 1

Formulate the denoising problem as a Minimum Variance FIR Wiener Problem, using the symbols $x(n)$, $v(n)$, $d(n)$, $\hat{d}(n)$, $e(n)$ as in the figure presented in the **Introduction**. No numerical values are required, but you need to specify the meaning of all the symbols and their relation to the sent and received data.

2.1.1 Answer (2 points)

In this problem, we received a corrupted signal $x(n)$. This signal consists of the corrupting noise $v(n)$, and our desired signal $d(n)$. Our goal is to somehow remove the corrupting noise $v(n)$ so that we can identify our desired signal, $d(n)$. In this case, we use a denoising method that consists

of developing a Wiener Filter, $W(z)$, which appropriately receives our corrupted signal $x(n)$ and generates an approximation of the desired signal. This approximation is called $\hat{d}(n)$, and we will check that it's an accurate approximation of $d(n)$ by assuring that this approximation $\hat{d}(n)$ produces an error signal, $e(n) = d(n) - \hat{d}(n)$, with minimum variance.

More precisely, we need to choose a filter $W(z)$ through minimizing this variance equation :

$$\min_{W(z)} E[|e(n)|^2] = \min_{W(z)} E[|d(n) - \hat{d}(n)|^2]$$

As $x(n)$ is convolved with the system $W(s)$ this is equivalent to :

$$\min_{w(k)} E[|d(n) - \sum_{k=0}^{m-1} w(k)x(n-k)|^2]$$

Using the definition of correlation, and by differentiating over $w(k)$, this eventually yields the Wiener-Hopf equation for optimal estimation:

$$\hat{w}(n) = R_x^{-1} r_{dx} \quad (1)$$

This is the filter we will solve for to estimate our desired signal, $d(k)$

2.2 Question 2

Plot the graphs for the received audio signals and the received noise signal. Use a separate plot for the noise signal.

```
[3]: ### Answer (1 point)
t = np.linspace(0, N, N)
tnoise = np.linspace(0, Nnoise, Nnoise)

fig = plt.figure(figsize=(15,10))
axs = fig.subplots(2, 2, sharey=True)

axs[0, 0].plot(t, data_noised[0, :])
axs[0, 0].set_title("Audio Signal (5 cm)")
axs[0, 0].set_ylabel("Amplitude")
axs[0, 0].set_xlabel("Sample")

axs[0, 1].plot(t, data_noised[1, :])
axs[0, 1].set_title("Audio Signal (100 cm)")
axs[0, 1].set_ylabel("Amplitude")
axs[0, 1].set_xlabel("Sample")

axs[1, 0].plot(t, data_noised[2, :])
axs[1, 0].set_title("Audio Signal (200 cm)")
axs[1, 0].set_ylabel("Amplitude")
```

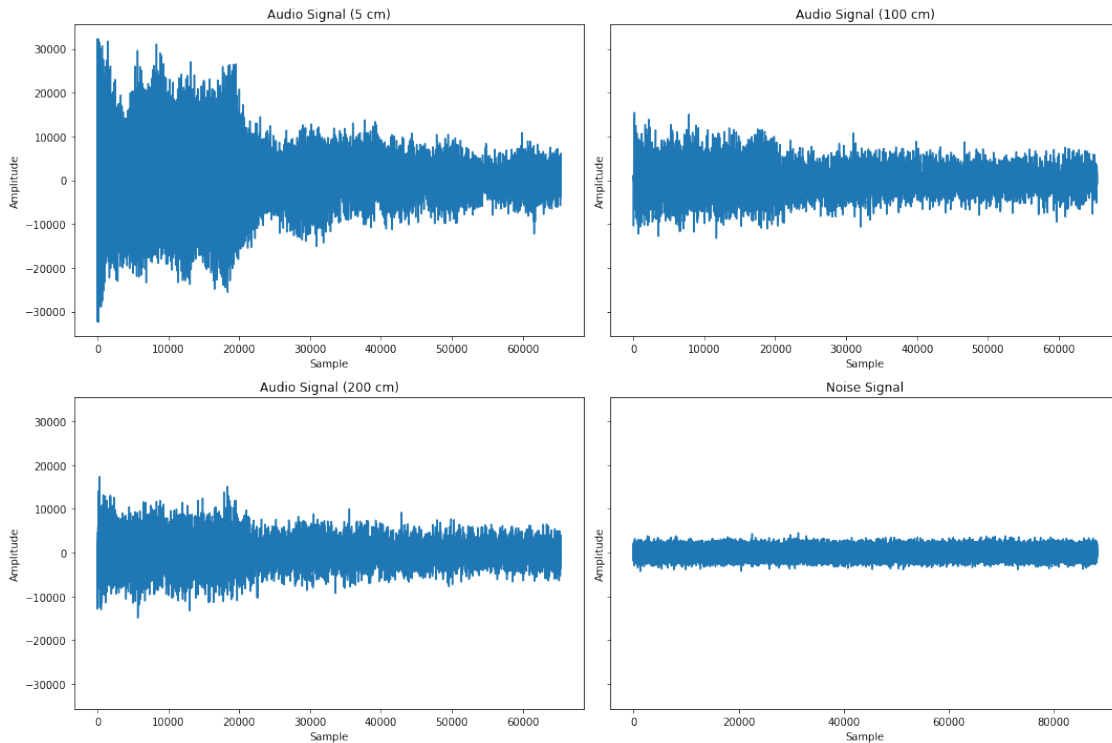
```

axs[1, 0].set_xlabel("Sample")

axs[1, 1].plot(tnoise, noise_measured)
axs[1, 1].set_title("Noise Signal")
axs[1, 1].set_ylabel("Amplitude")
axs[1, 1].set_xlabel("Sample")

plt.tight_layout()

```



2.3 Question 3

Determine the Wiener-Hopf equation for the denoising case with filter order m . What assumptions are we making for the audio signal and the noise in order to obtain the values of the filter w ? Derive how we can calculate the right hand side of the Wiener-Hopf equation assuming that we know $r_x(k)$, $\forall k$ and the noise variance σ_v^2 .

2.3.1 Answer (2 points)

Since the cost function defined at Question 1 quadratic, the minimum of this equation is located at the point where it's derivative (with respect to w) is zero. This becomes :

$$0 = E \left(\begin{bmatrix} x_0^* \\ x_1^* \\ \vdots \\ x_{n-m+1}^* \end{bmatrix} d(n)^T \right) - E \left(\begin{bmatrix} x_0^* \\ x_1^* \\ \vdots \\ x_{n-m+1}^* \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix} \right) \quad (2)$$

Where $d(n) = [d(0) \ d(1) \ \dots \ d(n)]^T$

Which we can use to solve for $\hat{w}(n) = R_x^{-1} r_{dx}$

Where:

$$R_x = \begin{bmatrix} r_x(0) & r_x(1)^* & \dots & \dots & r_x(m-1)^* \\ r_x(1) & r_x(0) & r_x(1)^* & \dots & r_x(m-2)^* \\ \vdots & & & & \vdots \\ r_x(n-m+1) & \dots & \dots & \dots & r_x(0) \end{bmatrix}$$

And r_{dx} is a vector of covariance between $d(n)$ and $x(n)$.

Since covariance is a linear function, and since $x(k) = d(k) + v(k)$, this covariance vector can be simplified as the sum of the autocorrelation of $d(k)$ and the cross correlation between $v(n)$ and $d(n)$.

$$x(k) = d(k) + v(k) \quad (3)$$

$$r_{dx} = r_d + r_{dv} \quad (4)$$

$$(5)$$

And since the desired signal $d(k)$ is uncorrelated with the noise $v(k)$, the cross correlation, r_{dv} is 0, therefore :

$$x(k) = d(k) + v(k) \quad (6)$$

$$r_{dx} = r_d \quad (7)$$

$$(8)$$

The autocorrelation r_x can be simplified using these two definitions, yielding only the autocorrelation for both terms, since, once again, $v(k)$ is uncorrelated with $x(k)$. The autocorrelation of ZMWN $v(n)$ is σ_v^2 at $k=0$ and 0 for all other values, so this gives us the final equations :

$$r_{dx}(k) = r_d(k) \quad (9)$$

$$r_x(k) = r_d(k) + r_v(k) \quad (10)$$

$$r_x(k) = r_d(k) + \sigma_v^2 \delta(k) \quad (11)$$

$$r_d(k) = r_{dx}(k) = r_x(k) - \sigma_v^2 \delta(k) \quad (12)$$

2.4 Question 4

Using your background noise measurement, estimate its mean and variance (σ_v^2). Print these estimates.

```
[4]: ### Answer (1 point)
mean_n = np.mean(noise_measured)
var_n = np.var(noise_measured)
print(f'Mean of the noise: {mean_n}')
print(f'Variance of the noise: {var_n}')
```

Mean of the noise: -0.7130270818895483

Variance of the noise: 993764.5058174969

2.5 Question 5

1. Assuming the ergodicity property of the audio signals, calculate the auto-correlation function $r_x(k)$ of every received audio signal for lags $0, \dots, 9$. Print these values.
2. Calculate the cross-correlation function $r_{dx}(k)$ of every received audio signal for lags $0, \dots, 9$. Print these values.
3. Using the auto-correlation estimation formula of the measured noise, compute and plot the noise auto-correlation function for lags $0..9$. Is it ZMWN? Explain your answer.

Note: if the signal $x(n)$ is auto-correlation ergodic, the auto-correlation function $r_x(k)$ can be estimated as (if we have only values x_1, x_2, \dots, x_N):

$$r_x(k) = \frac{1}{N-k} \sum_{i=k+1}^N x(i)x^*(i-k)$$

Note: if you are unfamiliar with python, take care when copying `np.arrays`. Read more [here](#).

```
[5]: ### Answer (3 points)

# order of the filter
f_order = 10

# list of lags
ks = list(range(f_order))

# initialize arrays
rx = np.ndarray([len(ks), N_exp])
rv = np.ndarray([len(ks), 1])

# Autocorrelation rx
for kk in ks:
    # correlation of noise signal
    corr_v = np.dot(noise_measured[kk:], noise_measured[: Nnoise - kk])
    rv[kk] = corr_v / (Nnoise - kk)
    for ii in range(N_exp):
```

```

    assert np.size(data_noised[0, kk:]) == np.size(data_noised[0, : N - kk])
    corr1 = np.dot(data_noised[ii, kk:], data_noised[ii, : N - kk])
    rk = corr1 / (N - kk)
    rx[kk, ii] = rk

# creaty copy of rx (not just reference)
rdx = rx.copy()

# subtract noise variance when k=0 i.e. first row
rdx[0, :] = rdx[0, :] - var_n

# 1. auto-correlation function r_x(k)
print(f"Rx :\n column 1: {titles[0]}, column 2 = {titles[1]}, column 3 = \u2192{titles[2]}\n{rx}\n")

# 2. Cross-correlation function r_dx(k)
print(f"Rdx :\n column 1: {titles[0]}, column 2 = {titles[1]}, column 3 = \u2192{titles[2]}\n{rdx}\n")

# 3. plot noise of auto-correlation function
plt.plot(rv, label="$r_v$")
plt.plot(ks, np.full(len(ks), var_n), label="$\sigma_v^2$ for reference")
plt.title("Noise utocorrelation function")
plt.xlabel("lag")
plt.ylabel("Autocorrelation")
plt.legend()

```

Rx :

```

column 1: 5 cm, column 2 = 100 cm, column 3 = 200 cm
[[42539267.27186559  9299701.76916726  9521284.27354726]
 [41425877.28277022  8653637.78700504  8875855.51602202]
 [40271794.85527986  8355775.09465058  8576127.26597257]
 [38773396.26707742  7993341.93370841  8204578.96271098]
 [36800737.66569328  7594321.52017368  7800731.35121623]
 [34672741.2251934   7171642.48928233  7365230.87209125]
 [32451796.98021558  6755829.94495834  6927127.89739317]
 [30227588.11485536  6336697.92764969  6489613.46523148]
 [28041970.59575249  5921686.49009984  6063385.06510405]
 [25931901.29424177  5534152.40088988  5653902.83408764]]

```

Rdx :

```

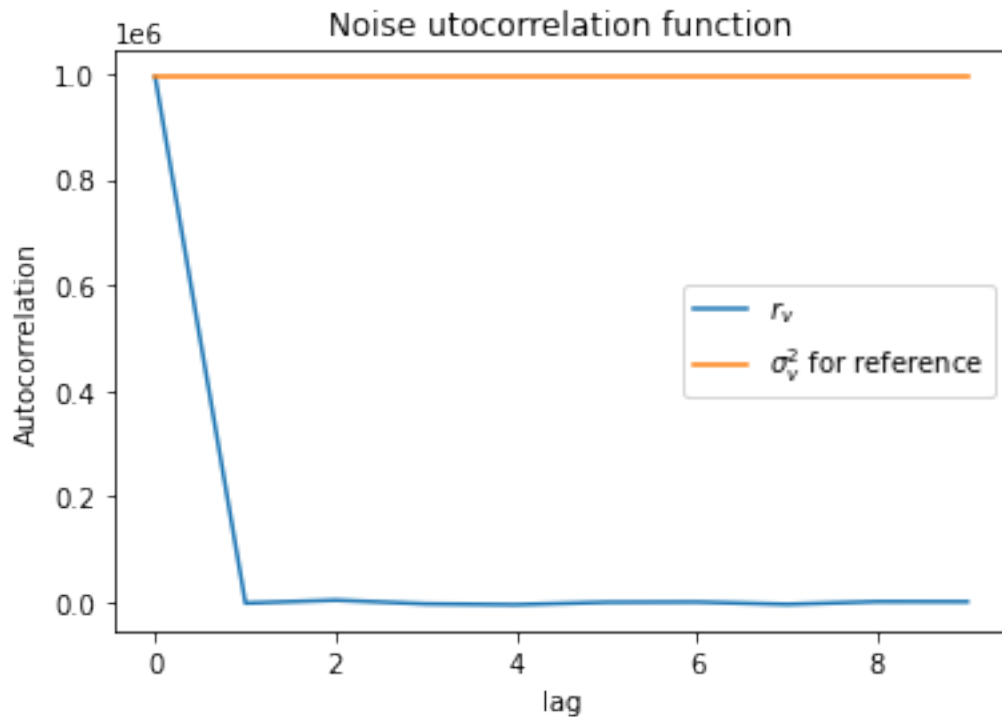
column 1: 5 cm, column 2 = 100 cm, column 3 = 200 cm
[[41545502.7660481  8305937.26334977  8527519.76772976]
 [41425877.28277022  8653637.78700504  8875855.51602202]
 [40271794.85527986  8355775.09465058  8576127.26597257]
 [38773396.26707742  7993341.93370841  8204578.96271098]
 [36800737.66569328  7594321.52017368  7800731.35121623]

```



```
[34672741.2251934  7171642.48928233  7365230.87209125]
[32451796.98021558  6755829.94495834  6927127.89739317]
[30227588.11485536  6336697.92764969  6489613.46523148]
[28041970.59575249  5921686.49009984  6063385.06510405]
[25931901.29424177  5534152.40088988  5653902.83408764]]
```

[5]: <matplotlib.legend.Legend at 0x7f82b142c280>



Yes, the noise seems to be ZMWN because its autocorrelation function resembles a dirac-delta function centered at 0 lag with value σ_v^2

2.6 Question 6

Using a 10-th order FIR Wiener filter, calculate the optimal filter coefficients for every received audio signal. Print these values.

Note: The `scipy.linalg.toeplitz()` function may be useful.

```
[6]: ### Answer (2 points)

# initialize arrays
Nrows, Ncols = np.shape(rx)
Rx = np.zeros((Nrows, Nrows, N_exp))
Rx_inv = np.zeros_like(Rx)
```

```
w_hat = np.zeros((f_order, N_exp))

# compute optimal filter coefficients for each signal
for ii in range(N_exp):
    Rx[:, :, ii] = lin.toeplitz(rx[:, ii])
    Rx_inv[:, :, ii] = lin.inv(Rx[:, :, ii])
    w_hat[:, ii] = np.matmul(Rx_inv[:, :, ii], rdx[:, ii])
    print(f"Optimal filter coefficients for signal recorded at {titles[ii]}:\n_
    ↳{w_hat[:, ii]}\n")
```

Optimal filter coefficients for signal recorded at 5 cm:

```
[ 0.50196601  0.46724797  0.08919879  0.0439224 -0.07684607 -0.03830459
 -0.0238235  0.00579324  0.00624896  0.00461774]
```

Optimal filter coefficients for signal recorded at 100 cm:

```
[ 1.44618736e-01  5.95365317e-01  2.18868922e-01  5.41296528e-02
 -3.69682771e-03 -2.67438067e-02 -8.95563579e-03 -9.90785295e-03
 -1.80772838e-02 -4.22867187e-04]
```

Optimal filter coefficients for signal recorded at 200 cm:

```
[ 0.14380427  0.59766937  0.22213355  0.05046705  0.00228241 -0.02442471
 -0.01830301 -0.01478027 -0.00939658 -0.00421749]
```

2.7 Question 7

Using the computed filter coefficients, compute an estimate of the denoised audio signals from the received ones. Plot together the denoised signal with the originally received one. Draw one plot for every audio signal.

Use the example code from this jupyter notebook (`play_signal()` function) in order to play the denoised signals - do you hear the difference between the originally received and the denoised one?

Note: the `scipy.signal.lfilter()` function can be useful when using the Wiener filter.

[7]: *### Answer (3 points)*

```
# initialize array
filtered = np.zeros_like(data_noised)

for ii in range(N_exp):
    filtered[ii, :] = sig.lfilter(w_hat[:, ii], 1, data_noised[ii, :])
    play_signal(data_noised[ii, :])
    sleep(2)
    play_signal(filtered[ii, :])
```

We can here some difference. Especially in the high frequency range. There the filter seems to have removed some of the noise.

2.8 Question 8

Now we will attempt to understand what the Wiener filter is doing. To that end we will use the periodogram on the noisy and filtered signals. Calculate P_x and P_d for each recording. For each distance, create two side-by-side plots: the first subplot should show P_x and P_d over the whole positive discrete frequency spectrum $[0, \pi]$ rad/sample, while the second subplot should show the exact same graph but zoomed in on the lower frequencies, e.g. $[0, 0.8]$ rad/sample. The y -axis should be in decibels, i.e. $20 \log_{10} P$. What do you conclude?

Note: Plot first P_x and then overlay on top P_d . Also play with transparencies to achieve a readable plot. In order to avoid indexing mistakes, it's suggested to use the `numpy.fft.fftfreq` function.

```
[8]: ### Answer (2 points)
SR = 22050
timestep = 1 / SR

# compute periodogram of different signals
# The Periodogram is defined as the Fourier Transform of the Autocorrelation
# Computing the Periodogograms and packing them up in a matrix :
for nn in range(N_exp):
    # This correlation function may not be the best way to do a autocorrelation
    rx1 = np.correlate(data_noised[nn, :], data_noised[nn, :], "full")
    rf1 = np.correlate(filtered[nn, :], filtered[nn, :], "full")

    # then compute the fourier transform
    fft_noised = np.fft.fft(rx1, axis=0)
    fft_filt = np.fft.fft(rf1, axis=0)

    # then get a magnitude by multiplying by the conjugate to yield pure real
    ↪ values
    Px = np.abs(fft_noised)
    Pf = np.abs(fft_filt)

    # check that there is no imaginary part
    assert sum(np.imag(Px)) == 0
    assert sum(np.imag(Pf)) == 0

    # then we're packing everything up in a matrix the way you had it ;)
    if nn == 0:
        rx2 = rx1.copy()
        rf2 = rf1.copy()
        Pxm = Px.copy()
        Pfm = Pf.copy()
    else:
        rx2 = np.vstack([rx2, rx1])
        rf2 = np.vstack([rf2, rf1])
        Pxm = np.vstack([Pxm, Px])
        Pfm = np.vstack([Pfm, Pf])
```

```

# generate the frequency indexes for plotting
nf = Pxm.shape[1]
freq = np.fft.fftfreq(nf, d=timestep)

# take only the positive part (first half)
halfway = nf // 2 + 1 # // is floor division (returns an int)
freq = freq[0: halfway]

# do the same for the powerspectrum
Pxm = Pxm[:, 0: halfway]
Pfm = Pfm[:, 0: halfway]

# get machine precision for left limit of loglog plot
# setting left limit to 0 gives problems
eps = np.finfo(float).eps
lowlim = 0.1

# transparency settings
alphaf = 0.7 # of filtered signal
alphan = 1 # of noisy signal

# color settings
colorf = "r" # of filtered signal
colorn = "b" # of noisy signal

for ii in range(N_exp):
    # First plot
    plt.figure(figsize=(15, 15))
    plt.subplot(211)
    plt.loglog(
        freq, 20 * np.log(Pxm[ii, :]), color=colorn, alpha=alphan,
        label="noised"
    )
    plt.loglog(
        freq, 20 * np.log(Pfm[ii, :]), color=colorf, alpha=alphaf,
        label="filtered"
    )
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Magnitude (dB)")
    plt.title(f"Periodogram of signal recorded at {titles[ii]}")
    plt.legend(loc=2)
    plt.xlim(eps, np.pi / (2 * np.pi) * SR)

    # Zoom plot
    plt.subplot(212)

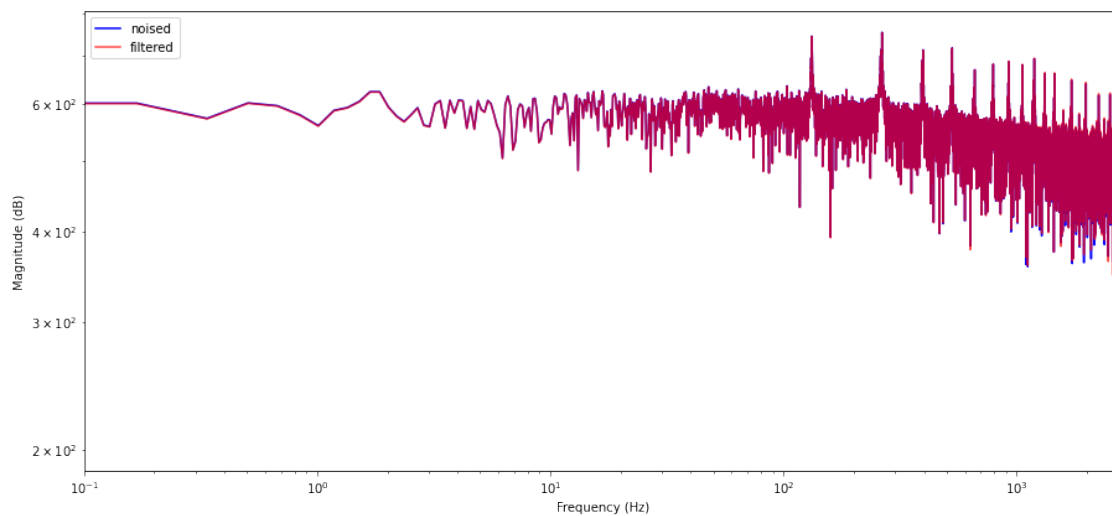
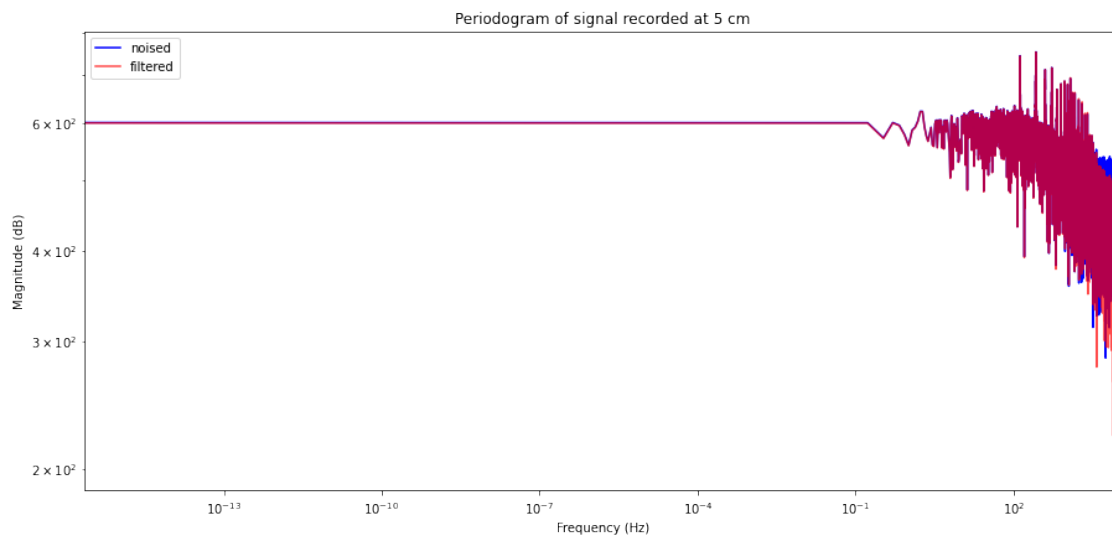
```

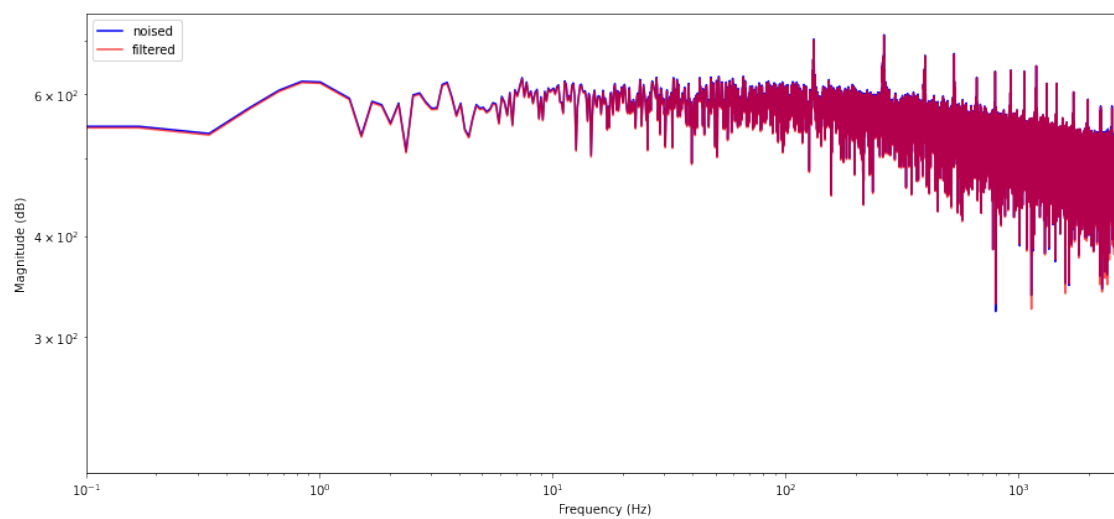
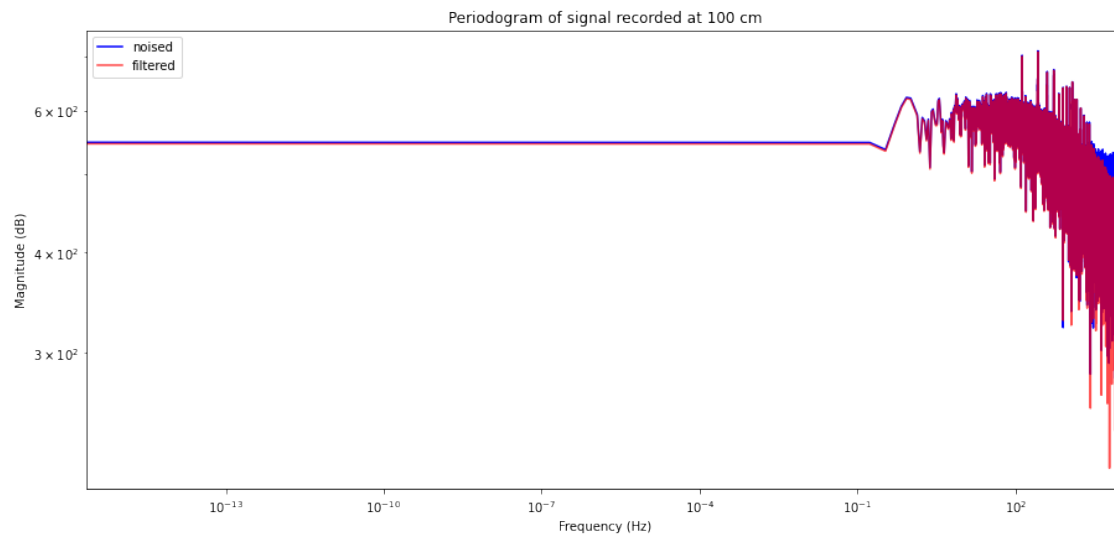
```

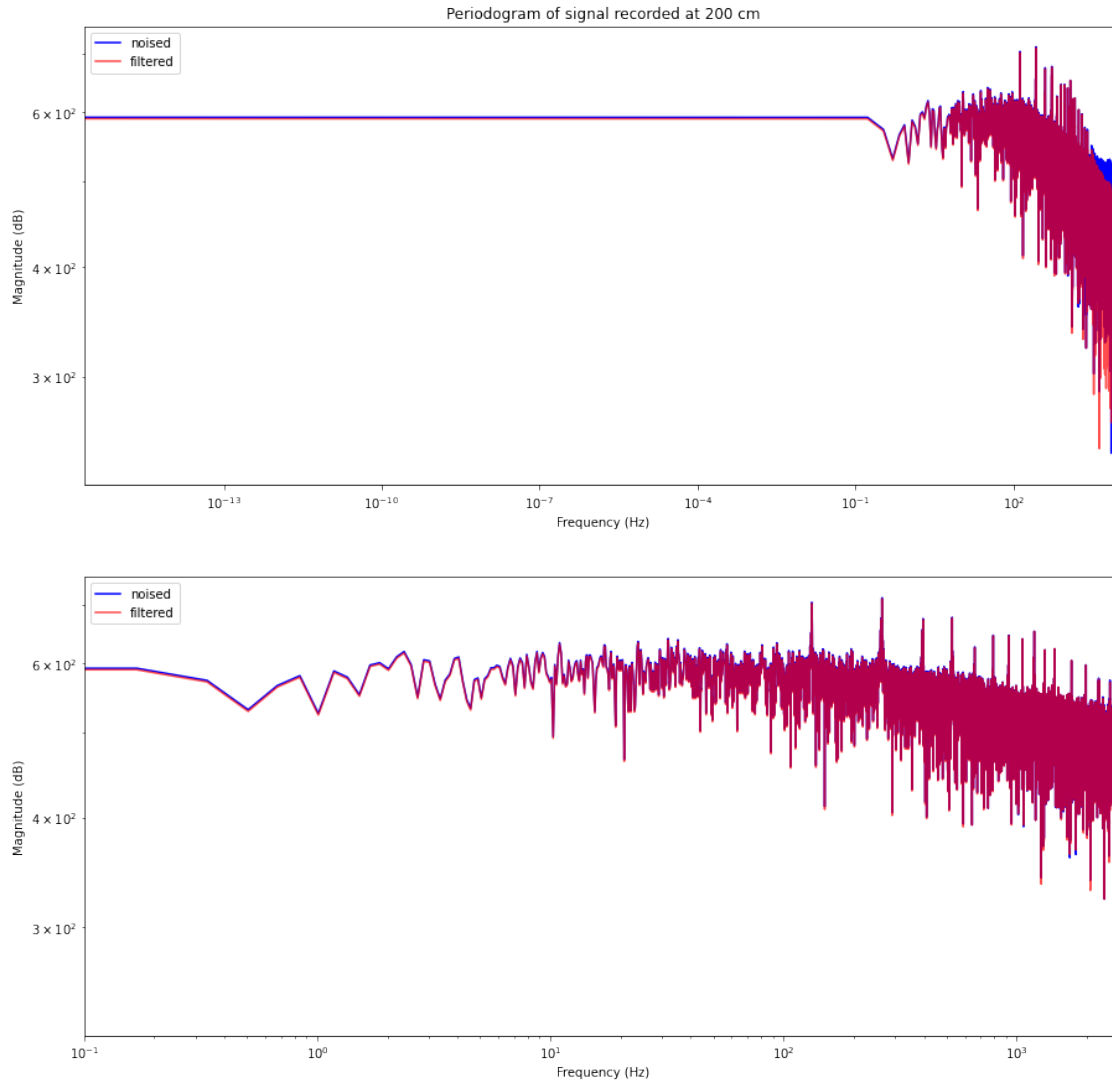
plt.loglog(
    freq, 20 * np.log(Pxm[ii, :]), color=colorn, alpha=alphan,
    label="noised"
)
plt.loglog(
    freq, 20 * np.log(Pfm[ii, :]), color=colorf, alpha=alphaf,
    label="filtered"
)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude (dB)")

plt.legend(loc=2)
plt.xlim(lowlim, 0.8 / (2 * np.pi) * SR)

```







We conclude that the filters are most active in the high frequency region above approximately 3kHz. The filtered signals have reduced gain at frequencies above that limit. Also, the filters seems to increase the gain in the 1 to 3kHz range. At frequencies below 1kHz, no noticeable difference can be observed in the plots.

2.9 Question 9

To further investigate what the Wiener filter is doing, we will plot its frequency response. In a single plot the frequency response for each Wiener filter over the positive discrete frequency spectrum. Make sure the y -axis is in decibels. What do you conclude?

[9]: *### Answer (2 points)*
The frequency response is the same as the fourier transform of the signal...
we had a list of experiment numbers to go through :

```

# Pad filter with zeros to plot continuous line
w_padded = np.zeros((nf, N_exp))
w_padded[0: f_order, :] = w_hat

for nn in range(N_exp):
    rw = np.correlate(w_padded[:, nn], w_padded[:, nn], "full")
    # take only the first half
    fft_what = np.abs(np.fft.fft(rw, axis=0))[0: len(rw)//2 + 1]
    what_m = fft_what

    if nn == 0:
        fr_what = fft_what.copy()
        what_M = what_m.copy()
    else:
        fr_what = np.vstack([fr_what, fft_what])
        what_M = np.vstack([what_M, what_m])

freq = np.fft.fftfreq(len(rw), d=timestep)

# take only the positive part (first half)
halfway = len(freq) // 2 + 1 # // is floor division
freq = freq[0: halfway]

```

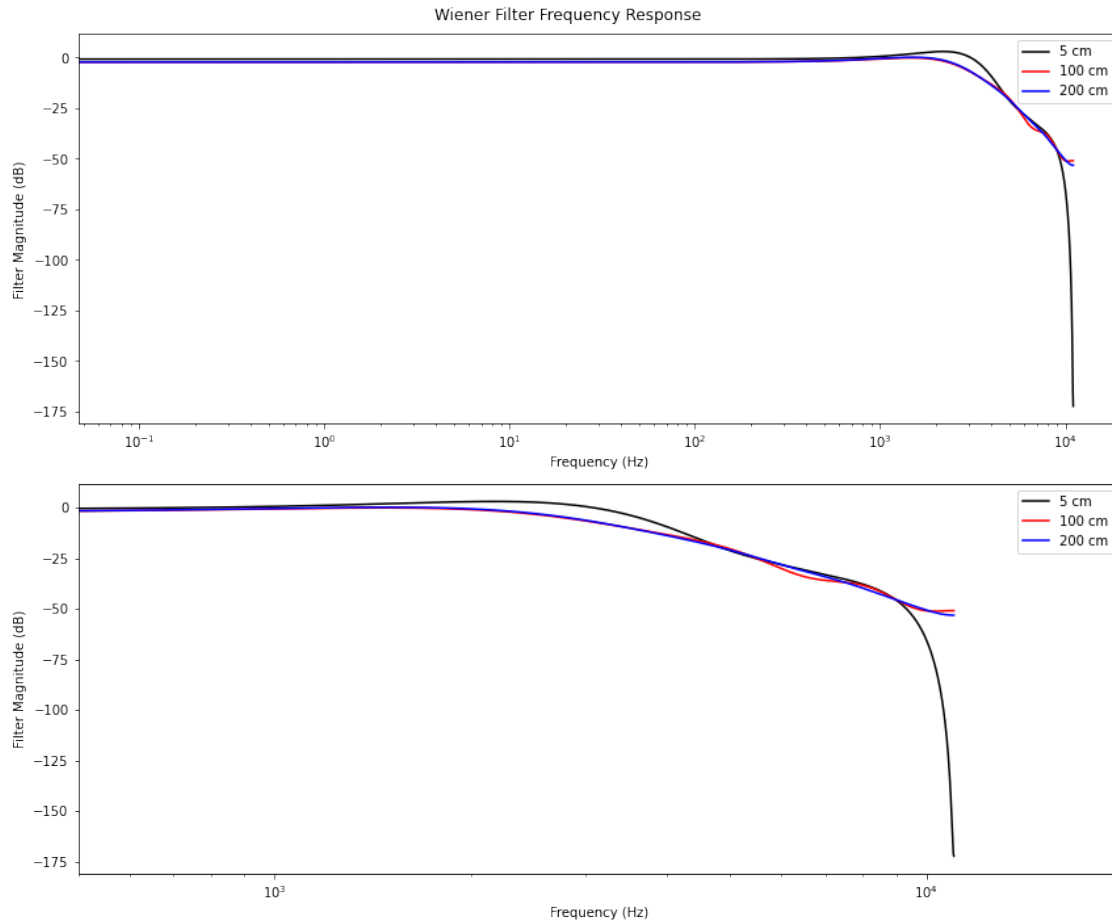
```

[10]: # plot the results
colors = ["k", "r", "b"] # list of line colors

fig = plt.figure(figsize=(12, 10))
for pp in range(2):
    plt.subplot(2, 1, pp+1)
    for ii in range(N_exp):
        plt.plot(freq, 20 * np.log(what_M[ii, :].real), colors[ii],
        label=titles[ii])
    plt.ylabel("Filter Magnitude (dB)")
    plt.xlabel("Frequency (Hz)")
    plt.xscale("log")
    plt.legend()
plt.suptitle("Wiener Filter Frequency Response")
plt.xlim(left=500)

plt.tight_layout()

```

The optimal filter for the 5cm signal indeed increases the gain in the 1 to 3kHz range after which its amplitude drops significantly. For all the 3 filters, we see that the frequency content is mostly unaltered from 0 to 1kHz. Most of the action happens at frequencies above 1kHz. Above 3kHz, all filters dampen the signal. This effect is most significant for the 5cm filter. We see that the filter amplifies the signal in between 1 to 3kHz. This is probably the range where the signal has highest power and thus where the signal to noise ration is best. The filter therefore increases gain in that range.

2.10 Question 10 (bonus)

Do you think you can change something to achieve better results than the above (by using this setup)? If so, how? If not, elaborate.

2.10.1 Answer (2 bonus points)

Increasing the order of the filters may lead to slightly better results. However, this improvement will most likely be small. Since the frequency content of the signal and the corrupting noise overlap a lot, except at high frequencies above 3kHz, where the signal has lower power than the noise.

Additionally, subtracting or dampening high frequency information may add noise or corrupt fea-

tures of the existing signal. Therefore, increasing the gain of the filter or other such operations will likely not improve performance at a certain point. One can of course optimize this trade-off, but the Wiener filter already is an optimal filter in terms of error, therefore, it already does an optimal job of trading off added noise and removed noise.

[]: