Digital Signal Processing 2020
Laboratory Work nr. 3

A Study of Adaptive Digital Filters

Lukas Stasytis, E MEI-0 gr. Birutė Paliakaitė, PhD Student
Kaunas University of Technology, Faculty of Electrical and Electronics Engineering

Introduction

In the following laboratory work, digital adaptive filters [1],[2] are used to remove engine noise from a plane cabin's sound signal in order to obtain a clear plane captain's sound signal. A Least Mean Squares (LMS), Normalized Least Mean Squares (NLMS) and De-correlated Recursive Least Squares (RLS) algorithms shall be employed to implement the digital adaptive filters. Evaluations of different hyperparameters and the effectiveness of each algorithm for the task of noise cancellation are measured. The filters are implemented in the Python programming language with a heavy reliance on the Scipy library[[3],[4],[5]].

Problem statement

To implement and evaluate the adaptive filters, a dataset is required with a task to solve. In this report, a plane captain's audio signal is used. Three separate signals are presented: one of the captain, labeled $s(n)$, one of the engine sound, labeled $x(n)$ and one of the two combined, labeled $d(n)$. The pilot audio signal is effectively the target signal for the filters and would not normally be available, however the goal of this report is to evaluate the adaptive filters, thus a ground truth signal to measure the designed filter's error rate is required.

In Figure 1 three sets of signals can be seen. The topmost left and right plots show the engine noise. Nothing can really be inferred from the plots, other than a small dip in amplitude around the 25000th time step. The second pair of graphs show the time and frequency domains of the cabin sound signal. Nothing can really be distinguished from the time domain, while the frequency domain shows a steadily decreasing amplitude of higher frequency noise. This alone suggests there are some more distinct components of the signal rather than pure random noise. The final pair of plots features the pilot's sound signal. This final signal would not normally be available when constructing an adaptive filter, given that it is essentially the result of the entire operation. Clear jumps in amplitude can be seen in the time domain, which could match a speaker's voice tone. A sound test of the signals confirmed it to be a clear 22 second recording of a captain speaking to plane passengers. To obtain a signal matching this captain signal as closely as possible, adaptive filters will be implemented and input the cabin sound signal as well as the engine signal. The output of said filters should be the captain's voice.

Figure 2 features the system as a whole. Two source signals $x(n)$ and $s(n)$ merge into a single input signal $d(n)$ which is passed through the adaptive filter. The result should then be fed back into the adaptive filter to, as the name implies, adapt the filter to the signal. After some iterations, the adaptive filter should start to model the noise signal component $x(n)$ and eliminate it from the input signal, leaving only the true signal of interest - the captain's voice $s(n)$.

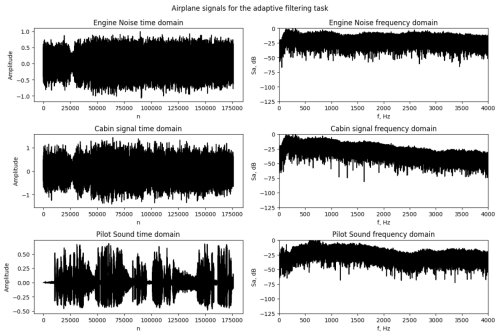

Figure 1. The Cabin, Engine and Pilot sound signals in time and frequency domains.

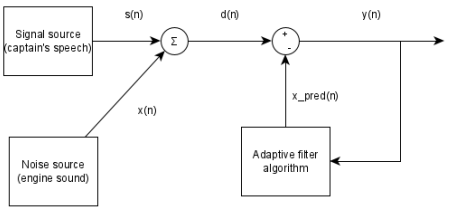

Figure 2. Diagram of the signal filtering scheme with the adaptive filter in place.

LMS filter

The first filter to be implemented is the Least Mean Squares filter. The filter features two hyperparameters: μ and M. μ represents the step size of the algorithm weight vector w's minimization function. This is much the same as the step size in conventional gradiant descent algorithms. [6]. Parameter M is used to describe the size of the w and x_a vectors and is effectively the order of the filter. x_a is a stack-style vector of the most recent M discrete input signals. M, μ, x_a and w are all initialized at the start of the algorithm. While M and μ are chosen, vectors x_a and w are initialized as column vectors of zeroes. Equations 1 and 2.

$$w(0) = [00...0]^T \tag{1}$$

$$x_a(0) = [00...0]^T \tag{2}$$

The algorithm is implemented in a for loop, with an iteration count equaling the length of the input signal. Each iteration, the w and x_a vectors are updated while taking into account their previous states. Equation 3 features the x_a vector's update function. Equations 4 to 6 feature the w weight vector's set of update functions. x_{pred} is the predicted x noise component value at timestep n by the filter. y is the output signal of the filter. In 6 the output signal is used to update the weights of the filter and improve future x_{pred} value accuracy.

$$x_a(n) = [x_a(n), x_a(n-1), ..., x_a(n-M+1)]^T \tag{3}$$
$$x_{pred}(n) = w^T(n)x_a(n) \tag{4}$$
$$y(n) = d(n) - x_{pred}(n) \tag{5}$$
$$w(n+1) = w(n) + 2\mu \cdot y(n)x_a(n) \tag{6}$$

Additionally, a second, normalized, adaptive LMS filter is constructed by changing equation 6 for equation 7. The modification makes parameter μ's effect on the slope descent vary depending on the energy of the signal's portion stored in the x_a vector. This can, both, decrease the chance of overshooting the local minimum as well as speed up the descent.

$$w(n+1) = w(n) + \frac{\mu}{x_a^T(n) \cdot x_a(n)} \cdot y(n)x_a(n) \tag{7}$$

Lastly, to accurately search for the optimal hyperparameters μ and M, the mean squared error (MSE) of the filter's output signal is measured versus the ground truth signal $s(n)$ given with the dataset. Equation 8 is used for calculating the MSE. The search for optimal hyperparameters is presented in the results section of the report.

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(s(n) - y(n))^2 \tag{8}$$

RMS filter

A recursive mean squared (RMS) filter with additional pre-whitening is additionally constructed for effectiveness comparison. By starting with the baseline LMS filter outlined in the previous section, the following modification is made: An additional eye matrix of size M x M is constructed and divided by a γ hyperparameter, equation 9. This acts as an inversion correlation matrix which takes indirect effect in the weight updating equation 13 via parameters $u(n)$ and $v(n)$. Due to the fact coefficient vectors u and v are affected by matrix P which has it's own update function 14, the modification effectively acts as an additional memory element for adjusting the weight vectors values based on past signal inputs further beyond the M order of the filter timesteps. y_{pred} and y output value functions remain unchanged.

$$P(0) = \gamma^{-1} \cdot I \tag{9}$$

Additional calculations as shown in equations 10 to 12 are done at each timestep.

$$v(n) = P(n-1) \cdot x_a(n) \tag{10}$$
$$u(n) = P^T(n-1) \cdot v(n) \tag{11}$$
$$k(n) = \frac{1}{\lambda + ||v(n)||^2 + \sqrt{\lambda}\sqrt{\lambda + ||v(n)||^2}} \tag{12}$$

$$w(n) = w(n-1) + \frac{y(n) \cdot u(n)}{\lambda + ||v(n)||^2} \tag{13}$$

$$P(n) = \frac{P(n-1) - k(n) \cdot v(n) \cdot u^T(n)}{\sqrt{\lambda}} \tag{14}$$

Results

Hyperparameter search

Using the previously outlined MSE function, multiple runs of each of the three filters are done on the dataset with varying parameters μ, M and γ as well as λ in the case of the RMS filter. Figure 3 features the results of varying parameters μ from 0.001 to 0.1 while M was adjusted from 2 to 60. A lower MSE value is better. Looking at the plot, it quickly becomes clear that hyperparameter M needs to be at least of the value 15 and that further increases past 20 do not yield any realistic gains in increasing the filter's effectiveness. It can also be seen that the overall error which the filters achieve by varying μ does not differ substantially and a higher μ value even increases the error rate as the order of the filter increases. However, this plot features only the total MSE of the whole dataset. Figure 4 is generated by limiting parameter M to 20 and only varying μ and then looking at the error value of the filter at specific time intervals of the entire signal's time domain. In the case of figure 4, the time increments are limited to 100ms each and total to 10s (the dataset goes to 22s but is limited to only the intial 10s for the figure for the sake of clarity). Two observations can be made from looking at the figure: a.) A lower μ value takes substantially longer to reach the local minimum, but remains stable thereafter, b.) a higher μ value reaches the local minimum much faster, but is prone to spikes in error around the signal at varying points. This matches the theory of gradient descent optimization and how too high of a minimization step can make the algorithm 'overshoot' the local minimum and possibly even diverge entirely. In the case of the dataset used in this report, a μ value of 0.01 and an M value of 20 seem to provide the most stable results.

Next, the Normalized LMS filter is used with the same set of parameters. The results were near identical in regards to which hyperparameter values are optimal relative to the regular LMS filter with a slightly higher error rate

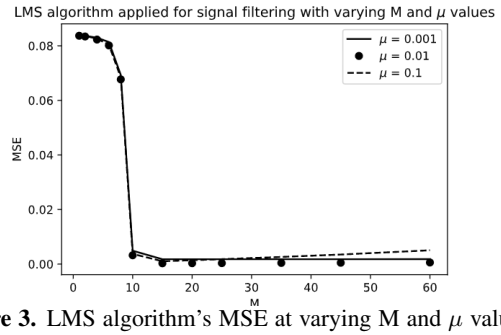

Figure 3. LMS algorithm's MSE at varying M and μ values. A quick descent to a minimum can be seen around M=10.

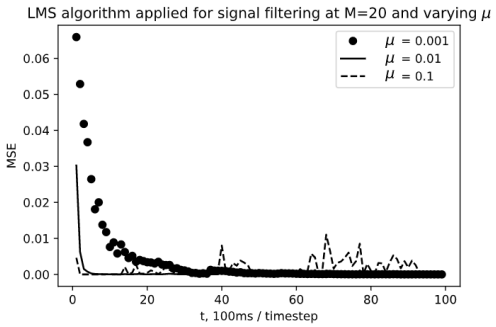

Figure 4. LMS algorithm's MSE at different timestep intervals with a fixed M and a varying μ.

overall, thus, no further exploration of optimal hyperparameters for said filter is done.

Lastly, the RMS filter is used. Given that the filter does not use parameter μ but instead uses λ and γ, an initial MSE test scheme was done with λ as the target parameter. Figure 5 illustrates the results when varying λ from 0.95 to 1. Higher λ values show a significantly better result while M shows a similar optimal point of around the order of 15 to 20. M = 20 is chosen for further parameter searching to have a better comparison with the previous LMS filter implementations. The λ parameter is further evaluated with a now fixed M=20 value and plotted with timestep intervals as was done in figure 4. The y axis scale is also adjusted to be logarithmic, as this would give a more clear picture of which λ values are performing best. Figure 6 is the result. It can be seen that the higher the λ value, the better the filter performs. It can also be noticed that the filter minimizes error extremely fast relative to the LMS filter. The choice is made to fix λ to 1 and plot the MSE in time intervals when varying γ. Figure 7 features the result of varying γ from 0.0001 to 1. The only observation to be made is that γ = 1 causes the filter to take longer to reach the minimum, but apart from that, γ has no effect on a filter's ability to converge. Given that the literature recommends small γ values, a value of 0.01 is kept.

With all the filter parameters explored, their MSE is compared versus each other. Table 1 features the results of lowest MSE values achieved with each filter. From the results, the LMS standard algorithm seems to perform the best, slightly ahead of the normalized version. The RMS algorithm performed substantially worse, although in the same order of magnitude. It should be noted, looking at Figure 8, that the LMS and nLMS algorithms have an initial 'boom' sound at the start of the output signal relative to the ground truth. The RMS has no such boom, given the substantially faster convergence time.

Table 1. Smallest MSE values achieved with each adaptive filter

LMS	$3.10 \cdot 10^{-4}$
nLMS	$3.91 \cdot 10^{-4}$
RMS	$8.13 \cdot 10^{-4}$

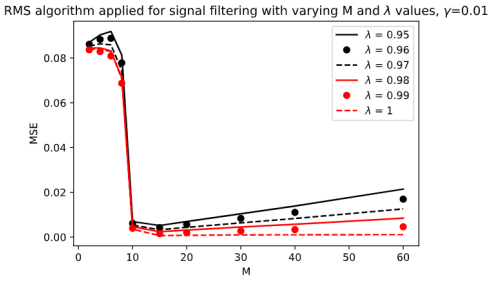

Figure 5. RMS algorithm's MSE with varying M and λ values. A similar result as with the LMS algorithm.

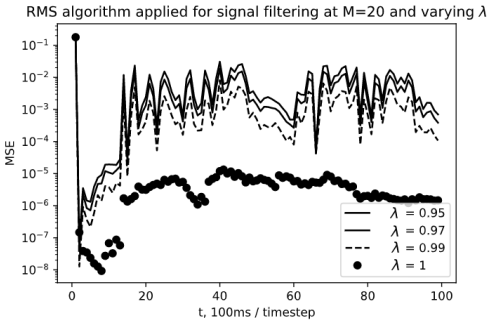

Figure 6. RMS algorithm's MSE at various timestep intervals with a fixed M value and varying λ.

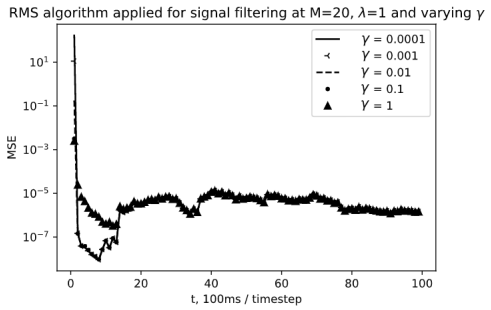

Figure 7. RMS algorithm's MSE at various timestep intervals with a fixed M and λ values while varying γ. A faster drop-off can be observed for lower γ values, but all values reach the same final MSE values.

Discussion

Python

Python was the language of choice for the assignment given it's open-source nature, ample literature and ease of use for prototyping algorithms. Two key nuances of Python's numerical computing library Numpy have to be noted. The first is array initialization. When a vector of 1 dimension is initialized in Numpy, it has to be explicitly stated as size (1,X) where X is the length of the vector. Conventional initialization methods like the function $zeroes(X)$ will return a numpy array of the shape (X,).

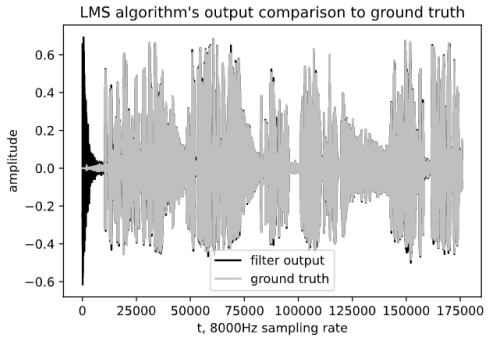

Figure 8. Output signal of the LMS filter as well as the ground truth provided in the dataset. A spike at the start of the signal can be observed that quickly descends as the filter adapts to the signal.

This is extremely important, because this type of numpy array shape blocks any sort of transposition of the vector. In other words, the vector is treated as a row vector and never as a column vector. For the $transpose()$ function to turn a vector from a row vector to a column vector, a 2 dimensional array has to be specified by explicitly stating that one of the dimensions is of size 1. In most applications, this is not a problem, because Numpy supports array broadcasting and reshapes the '1.5 dimension' array to fit any operation being applied on it when taking into account the second term, for example a different array. It is specifically when operations involving column vector x matrix or matrix multiplications as opposed to dot products are used that this reshaping can lead to unexpected outcomes. What is even more important is that Python will not throw errors due to array incompatibly, but silently reshape the arrays to fit the operation. This can lead to code that compiles, but outputs wrong results at runtime.

The second point is the need to be explicit about the types of operations being done on matrices. By default, a multiplication operation between two numpy arrays will result in a regular matrix multiplication. $dot()$ and $multiply()$ need to be specified to achieve dot product and element-wise multiplications respectively.

It is of the utmost importance when designing mathematical algorithms using the numpy library to pay careful attention to how the arrays are presented and which matrix multiplication functions are being called.

Conclusions

In this laboratory work, three separate adaptive filters were used to filter out engine noise from a cabin input sound signal on a plane to obtain the plane captain's voice signal. All three filters showed remarkable effects of quickly modeling the engine noise component and cleanly filtering the input signal. The hyperparameter search proved to be highly important when optimizing the filters, with vast performance differences, especially when considering the M parameter. The choice of using the Python programming language for the assignment proved to not be without issues, notable being the specifics of initiating column arrays and making sure the right matrix multiplication type is used rather than Python's assumptions.

Bibliography

[1] Poularikas a. d.; ramadan z. m. adaptive filtering perimer with matlab // crc press. 2006, p.240.

[2] Diniz. p. s. r. adaptive filtering: Algorithms and practical implementations // springer, third edition. 2008, p. 627.

[3] numpy python computing package. Link: https://numpy.org.

[4] scipy python scientific computing package. Link: https://www.scipy.org.

[5] matplotlib python package. Link: $https : //matplotlib.org/api/pyplot_api.html$.

[6] T. Zhang, Solving large scale linear prediction problems using stochastic gradient descent algorithms, in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 116.

Appendix

Source code

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'

# %%
import numpy as np
from numpy import cos, sin, pi, absolute, arange, sqrt
import scipy.signal as signal
import scipy.io
from scipy.io.wavfile import write
from scipy.ndimage.interpolation import shift
from utils import *

# %%
class Signal:
    def __init__(self,name,data,fd):
        self.name = name
        self.data = data
        self.fd = fd
        self.n = len(data)

    def plot(self,n=None,time=True,frequency=True):
        if n is None:
            n = self.n
        plot_signal(self.data[:n],self.fd,0,f"{self.name}")

# %%
inputs = {}
fd = 8000
datapack = scipy.io.loadmat('lab3_signalai.mat')
signal_names = ['variklioSig','kabinosSig','pilotoSig']
for s in signal_names:
    inputs[s] = Signal(s,datapack[s][0],fd)

# %%
inputs

def plot_multiple_signals(x,fd,start_delay,title,count,titles,domains=["time","freq"],filename=None):
    # plotting a signal in time and freq domains
    fig, axs = plt.subplots(count,len(domains),figsize=(4*count,8),dpi=100,constrained_layout=True)
    for i in range(count):
        f, s = convert_to_frequency_domain(x[i],fd)
        if "time" in domains:
            if len(domains) > 1:
                plot_amplitude(x[i],fd,start_delay,axs[i][0],sub_title=titles[i])
            else:
                plot_amplitude(x[i],fd,start_delay,axs[i],sub_title=titles[i])
        if "freq" in domains:
            if len(domains) > 1:
                plot_frequency(f,s,fd,axs[i][1],sub_title=titles[i])
            else:
                plot_frequency(f,s,fd,axs[i],sub_title=titles[i])
    fig.suptitle(title)

    if filename is None:
        plt.show()
    else:
        plt.savefig(filename)

# %%
x = [inputs['variklioSig'].data,inputs['kabinosSig'].data,inputs['pilotoSig'].data]
fd = 8000
title = "Airplane signals for the adaptive filtering task"
titles = ["Engine Noise","Cabin signal", "Pilot Sound"]
count = len(x)
plot_multiple_signals(x,fd,0,title,count,titles,domains=["time","freq"],filename='f1.png')

# %%
inputs['variklioSig'].plot()

# %%
inputs['pilotoSig'].plot()

# %%
inputs['kabinosSig'].plot()

# %%
def mse(x,y):
    return ((x - y)**2).mean(axis=0)

class Filter:
    def __init__(self,name):
        self.name = name

class AdaptiveFilter(Filter):
    def __init__(self,name,M,miu):
        super().__init__(name)
        self.M = M
        self.miu = miu
        self.w = np.zeros(M).transpose()
        self.xa = np.zeros(M).transpose()

    def adapt_step(self,x,d,normalized=False):
        self.xa = shift(self.xa,1)
        self.xa[0] = x

        xiv = self.w.dot(self.xa)
        siv = d - xiv

        if normalized:
            self.w = self.w + (self.miu / (self.xa.transpose().dot(self.xa))) * siv*self.xa
        else:
            self.w = self.w + 2*self.miu * siv*self.xa

        return siv

    def filter_signal(self,x,d,normalized=False,rmk=False):
        y = np.zeros(len(x))
        for i in range(len(x)):
            y[i] = self.adapt_step(x[i],d[i],normalized)
        return y
```

```
def mse_list(x,y,step):
   mses = []
   for i in range(len(x)):
      if i % step == 0:
         mses.append(mse(x[i-step:i],y[i-step:i]))
   return mses

# %%
dn = inputs['kabinosSig'].data # dn
xa = inputs['variklioSig'].data # xa
sn = inputs['pilotoSig'].data # sn
mius = [0.001,0.01,0.1]
Ms = [20]
mses = []
step = 800
for miu in mius:
   m0 = []
   for M in Ms:
      af = AdaptiveFilter("adaptive filter",M,miu)
      y = af.filter_signal(xa,dn,True)
      m0.append((miu,M,mse_list(sn,y,step)))
      #m0.append((miu,M,mse(sn,y)))
      print(miu,M," done")
   mses.append(m0)

# %%
mses[0][0][2]

# %%
k = ['ko','k-','k--']
i = 0
for m0 in mses:
   plt.plot(m0[0][2][:100],k[i],label=f'miu = {m0[0][0]}')
   i+=1
plt.title("nLMS algorithm applied for signal filtering at M=20 and varying $\mu$")
plt.xlabel("t, 100ms / timestep")
plt.ylabel("MSE")
plt.legend()

# %%
#x0 = [x[2] for x in mses[0]]
k = ['k','ko','k--']
i = 0
for m0 in mses:
   plt.plot([x[1] for x in m0],[x[2] for x in m0],k[i],label=f'$\mu$ = {m0[2][0]}')
   i+=1
plt.title("LMS algorithm applied for signal filtering with varying M and $\mu$ values")
plt.xlabel("M")
plt.ylabel("MSE")
plt.legend()

# %%
plt.plot(y,'r')
plt.plot(sn,'--b')
plt.show()
print(f"mse: {mse(sn,y)}")

# %%
from scipy.io.wavfile import write
write('y.wav',8000, y)
write('sn.wav',8000, sn)
write('dn.wav',8000, dn)
write('xa.wav',8000, xa)

# %%
dn = inputs['kabinosSig'].data # dn
xa = inputs['variklioSig'].data # xa
sn = inputs['pilotoSig'].data # sn
miu = 0.01
M = 20
mses = []
step = 800
af = AdaptiveFilter("adaptive filter",M,miu)
y = af.filter_signal(xa,dn)

# %%
mses = mse_list(sn,y,step)
plt.plot(mses)

# %%
dn = inputs['kabinosSig'].data # dn
xa = inputs['variklioSig'].data # xa
sn = inputs['pilotoSig'].data # sn
af = AdaptiveFilter("adaptive filter",20,0.01)
y = af.filter_signal(xa,dn)
write('y_optimal.wav',8000, y)

# %%
#dn = inputs['kabinosSig'].data # dn
#xa = inputs['variklioSig'].data # xa
#sn = inputs['pilotoSig'].data # sn
af = AdaptiveFilter("adaptive filter",20,0.01)
y = af.filter_signal(xa,dn,True)
write('y_normalized.wav',8000, y)

# %%
af.xa.transpose().dot(af.xa)

# %%
plt.plot(y,'k',label="filter output")
plt.plot(sn,'k',label='ground_truth',c='0.75')
plt.title("LMS algorithm's output comparison to ground truth")
plt.ylabel("amplitude")
plt.xlabel("t, 8000Hz sampling rate")
```

```
plt.legend()
plt.show()
print(f"mse: {mse(sn,y)}")

# %%
class AdaptiveRMKFilter(Filter):
   def __init__(self,name,M,miu):
      super().__init__(name)
      self.M = M
      self.miu = miu
      self.w = np.zeros(M).transpose()
      self.xa = np.zeros(M).transpose()
      self.gamma = 0.01
      self.I = np.identity(M)
      self.P = self.I / self.gamma

      self.lambda0 = 1

   def norm(self,x):
      return np.matmul(x.transpose(),x)

   def adapt_step(self,x,d):
      self.xa = shift(self.xa,1)
      self.xa[0] = x
      xa = self.xa.reshape(self.M,1)
      P = self.P
      w = self.w.reshape(self.M,1)
      l = self.lambda0

      #print(f'l: {l}\nxa:\n{xa} \nP:\n{P}\nw:\n{w}')

      # RMS algorithm

      # np.matmul returns matrix product of two given arrays
      # np.multiply returns element-wise multiplication of two given arrays
      # np.dot    returns scalar or dot product of two given arrays

      v = np.matmul(P,xa)
      u = np.matmul(P.transpose(),v)
      #print(v)
      #print(v.shape)
      v = v.reshape(len(v),1)
      #print(v.shape)
      k = 1 / ( l + self.norm(v) + sqrt(l) * sqrt( l + self.norm(v) ) )

      xiv = np.matmul(w.transpose(),xa)
      siv = d - xiv

      self.P = ( P - k * np.matmul(v,u.transpose())) / sqrt(l)
      self.w = w + (siv*u)/(l + self.norm(v))
      #print(w.shape,xa.shape,v.shape,u.shape)
      #print("\nafter calculations\n")
      #print(f"\nv: {v} \nu: {u} \nk:{k} \nnew P: \n{self.P} \nxiv:\n{xiv}\nnew w:\n{self.w}")
      return siv

   def filter_signal(self,x,d):
      y = np.zeros(len(x))
      for i in range(len(x)):
         #print(f"\niteration: {i}")
         y[i] = self.adapt_step(x[i],d[i])
      return y

dn = inputs['kabinosSig'].data # dn
xa = inputs['variklioSig'].data # xa
sn = inputs['pilotoSig'].data # sn


af = AdaptiveRMKFilter("adaptive filter",20,0.01)
#y = af.filter_signal(xa[0:20],dn[0:20])
y = af.filter_signal(xa,dn)
#y = af.filter_signal(xa[:2],dn[:2])
#write('y_rmk.wav',8000, y)

# %%
plt.plot(y[100:],'r')
plt.plot(sn[100:],'b--')
plt.show()
print(f"mse: {mse(sn,y)}")

# %%
write('y_rmk.wav',8000, y)

# %%
dn = inputs['kabinosSig'].data # dn
xa = inputs['variklioSig'].data # xa
sn = inputs['pilotoSig'].data # sn
mius = [0.95,0.96,0.97,0.98,0.99,1]
Ms = [2,4,6,8,10,15,20,30,40,60]
mses = []
step = 800
for miu in mius:
   m0 = []
   for M in Ms:
      #af = AdaptiveFilter("adaptive filter",M,miu)
      af = AdaptiveRMKFilter("adaptive filter",M,miu)
      y = af.filter_signal(xa,dn)
      #m0.append((miu,M,mse_list(sn,y,step)))
      m0.append((miu,M,mse(sn,y)))
      print(miu,M," done")
   mses.append(m0)

# %%
k = ['k','ko','k--','r','ro','r--']
i = 0
for m0 in mses:
   plt.plot([x[1] for x in m0],[x[2] for x in m0],k[i],label=f'$\lambda$ = {m0[2][0]}')
   i+=1
plt.title("RMS algorithm applied for signal filtering with varying M and $\lambda$ values, $\gamma$=0.01")
plt.xlabel("M")
```

```
plt.ylabel("MSE")
plt.legend()

# %%
dn = inputs['kabinosSig'].data # dn
xa = inputs['variklioSig'].data # xa
sn = inputs['pilotoSig'].data # sn
mius = [0.01]
Ms = [20]
mses = []
step = 800
for miu in mius:
    m0 = []
    for M in Ms:
        #af = AdaptiveFilter("adaptive filter",M,miu)
        af = AdaptiveRMKFilter("adaptive filter",M,miu)
        y = af.filter_signal(xa,dn)
        #m0.append((miu,M,mse_list(sn,y,step)))
        m0.append((miu,M,mse(sn,y)))
        print(miu,M," done")
    mses.append(m0)

# %%
k = ['k','k3','k--','k.','k^']
i = 0
for m0 in mses:
    plt.plot(m0[0][2][:100],k[i],label=f'miu = {m0[0][0]}')
    i+=1
plt.title("RMS algorithm applied for signal filtering at M=20, $\lambda$=1 and varying $\gamma$")
plt.xlabel("t, 100ms / timestep")
plt.ylabel("MSE")
plt.yscale('log')
plt.legend()

# %%
```