Chapter 11
Adaptive Signal Processing

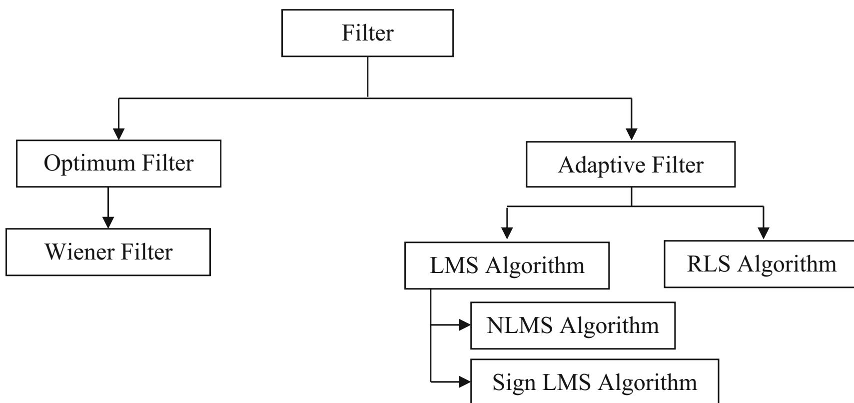
Learning Objectives

After reading this chapter, the reader is expected to

- Implement and analyse the Wiener filter.
- Write a python code to implement the LMS algorithm and its variants.
- Perform system identification using the LMS algorithm.
- Perform inverse system modelling using the NLMS algorithm.
- Implement adaptive line enhancer using the LMS algorithm and its variants.
- Implement the RLS algorithm.

Roadmap of the Chapter

The roadmap of this chapter is depicted below. This chapter starts with the Wiener filter, least mean square (LMS) algorithm and its variant approaches for adaptive signal processing applications like system identification and signal denoising. Next, the RLS algorithm is discussed with the suitable python code.



© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

S. Esakkirajan et al., *Digital Signal Processing*,
https://doi.org/10.1007/978-981-99-6752-0_11

PreLab Questions

1. List out the valid differences between the optimal filter and the adaptive filter.
2. What is an adaptive filter? How it differs from the ordinary filter.
3. Examples of adaptive filter.
4. When are adaptive filters preferred?
5. List out the performance measures of the adaptive filter.
6. What is an LMS algorithm?
7. What do you mean by least square estimation?
8. List out the variants of LMS algorithm.
9. How the step size impacts the LMS algorithm?
10. What is the RLS algorithm, and how it differs from LMS?

11.1 Wiener Filter

Wiener filter is the mean square error (MSE) optimal stationary linear filter for signal corrupted by additive noise. The Wiener filter computation requires the assumption that the signal and noise are in the random process. The general block diagram of the Wiener filter is shown in Fig. 11.1. The main objective of the Wiener filter is to obtain the filter coefficient of the LTI filter, which can provide the final output ($y[n]$) as much as the minimum MSE between the output and the desired signal or target ($d[n]$). In Fig. 11.1, $s[n]$ denotes the original signal, which is a clean signal, and it is corrupted by additive noise $\eta[n]$ to give the signal $x[n]$. The parameters of the filter have to be designated has to be designed in such a way that the output of the filter $y[n]$ should resemble the desired signal $d[n]$ such that the error '$e[n]$' is minimum.

The expression for the optimal Wiener filter is given by

$$h_{\text{opt}} = R^{-1}p \tag{11.1}$$

The above expression is termed as 'Wiener-Hopf' expression, which is named after American-born Norbert Wiener and Austrian-born Eberhard Hopf. The expression for optimal filter depends on the autocorrelation matrix (R) of the observed signal ($x[n]$) and the cross-correlation vector (p) between the observed signal ($x[n]$) and the desired signal ($d[n]$). h_{opt} denotes the optimal filter coefficients.

Experiment 11.1 Wiener Filtering

The aim of this experiment is to implement the Wiener filtering using python. Here the optimal filter coefficients are obtained using the Wiener-Hopf equation given in

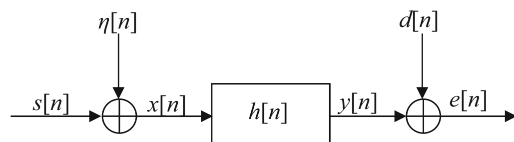


Fig. 11.1 Block diagram of Wiener filter

```
#Wiener filter
import numpy as np
from numpy.random import randn
import matplotlib.pyplot as plt
from scipy.linalg import toeplitz
from scipy import signal
#Step 1: Generation of signal s[n]
t=np.linspace(0,1,100)
s=np.sin(2*np.pi*5*t)
Ns=len(s)
#Step 2: Generation of random noise
# n=randn(len(t))*0.1
n=np.random.normal(0,.2,len(s))
#Step 3: Observed signal x[n]
x=s+n
#Step 4: Autocorrelation of observed signal
rxx=np.correlate(x,x,mode='full')
#Step 5: Cross-correlation between desired and observed signal
rsx=np.correlate(s,x,mode='full')
#Step 6: Deciding the length of the filter
Nh=11
#Step 7: Trimming the autocorrelation and cross-correlation values
rxx1=rxx[Ns-1:Ns+Nh-1]
rsx1=rsx[Ns-1:Ns+Nh-1]
#Step 8: Obtaining the autocorrelation matrix
Rx=toeplitz(rxx1)
#Step 9: Inverse of the autocorrelation matrix
Rx1=np.linalg.inv(Rx)
#Step 10: Obtaining the filter coefficient
w1=np.matmul(Rx1,rsx1)
#Step 11: Filtering the noisy signal
y=signal.lfilter(w1,1,x)
plt.subplot(3,1,1),plt.plot(t,s),plt.xlabel('t-->'),plt.ylabel('Amplitude'),
plt.title('Clean signal'),plt.subplot(3,1,2),plt.plot(t,x),plt.xlabel('t-->'),
plt.ylabel('Amplitude'),plt.title('Noisy signal'),plt.subplot(3,1,3), plt.plot(t,y)
plt.xlabel('t-->'),plt.ylabel('Amplitude'),plt.title('Filtered signal'),plt.tight_layout()
```

Fig. 11.2 Python code for Wiener filtering

Eq. (11.1). The python code for Wiener filter is shown in Fig. 11.2. Simulation result of the python code given in Fig. 11.2 is depicted in Fig. 11.3.

The built-in functions used in python code shown in Fig. 11.2 is summarized in Table 11.1.

Inference

From Fig. 11.3, it can be made the following observations:

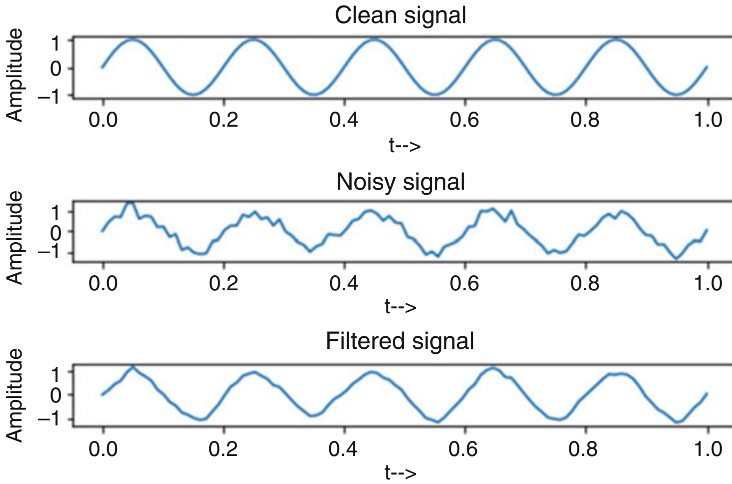


Fig. 11.3 Simulation result of Wiener filter

Table 11.1 Built-in functions used in the python code given in Fig. 11.2

S. No.	Objective	Built-in function	Library
1	To generate a clean sinusoidal signal of 5 Hz frequency	np.sin()	Numpy
2	To add white noise, which follows normal distribution to clean signal	np.random.normal()	Numpy
3	To perform autocorrelation	np.correlate()	Numpy
4	To obtain the inverse of the matrix	np.linalg.inv()	Scipy
5	To perform convolution	signal.lfilter()	Scipy

1. The input or clean signal frequency is 5 Hz, and it is a smooth sine waveform.
2. The additive noise added signal as input to the Wiener filter, and it is a distorted signal.
3. The filtered signal is not a smooth sine waveform. However, this waveform is far better than the noisy signal. Hence, the Wiener filter has a capability to minimize the impact of additive noise in a signal.

Task

1. Change the value of standard deviation in random noise generation python command 'np.random.normal(0,**.2**,len(s))' given in Fig. 11.2. Execute and make the appropriate changes in this python code to get 'filtered signal' as similar as 'clean signal'.

Experiment 11.2 Wiener Filter Using Built-In Function

This experiment performs the Wiener filtering using built-in function in '*scipy*' library. The built-in function is available in *the* '*scipy*' library '*wiener*' can be used to filter out the noisy components. In this experiment, noise-free sinusoidal signal of 5 Hz frequency is generated. The clean signal is corrupted by adding

Table 11.2 Steps followed and built-in functions

S. No.	Objective	Built-in function	Library
1	To generate a clean sinusoidal signal of 5 Hz frequency	np.sin()	Numpy
2	To add white noise, which follows normal distribution to clean signal	np.random.normal()	Numpy
3	To minimize the impact of noise using Wiener filter	signal.wiener()	Scipy

```
#Wiener filter
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
#Step 1: Generation of clean signal
t=np.linspace(0,1,100)
s=np.sin(2*np.pi*5*t)
#Step 2: Adding noise
n=np.random.normal(0,.2,len(s))
x=s+n
#Step 3: Wiener filter
y=signal.wiener(x)
#Step 4: Plotting the results
plt.subplot(3,1,1),plt.plot(t,s),
plt.xlabel('Time'),plt.ylabel('Amplitude'),plt.title('Clean signal')
plt.subplot(3,1,2),plt.plot(t,x),plt.xlabel('Time'),plt.ylabel('Amplitude'),
plt.title('Noisy signal'),plt.subplot(3,1,3),plt.plot(t,y)
plt.xlabel('Time'),plt.ylabel('Amplitude'),plt.title('Filtered signal')
plt.tight_layout()
```

Fig. 11.4 Wiener filtering using built-in function

random noise, which follows the normal distribution with zero mean and 0.2 standard deviation. The corrupted signal is then passed through the Wiener filter to minimize the impact of noise. The steps followed along with the built-in functions used in the program are given in Table 11.2.

The python code which performs this task is shown in Fig. 11.4, and the corresponding output is shown in Fig. 11.5.

Inference

From Fig. 11.5, it is possible to infer that the impact of noise is minimized after passing the noisy signal through Wiener filter.

11.1.1 Wiener Filter in Frequency Domain

From Wiener-Hopf equation, the expression for the optimal Wiener filter is given by

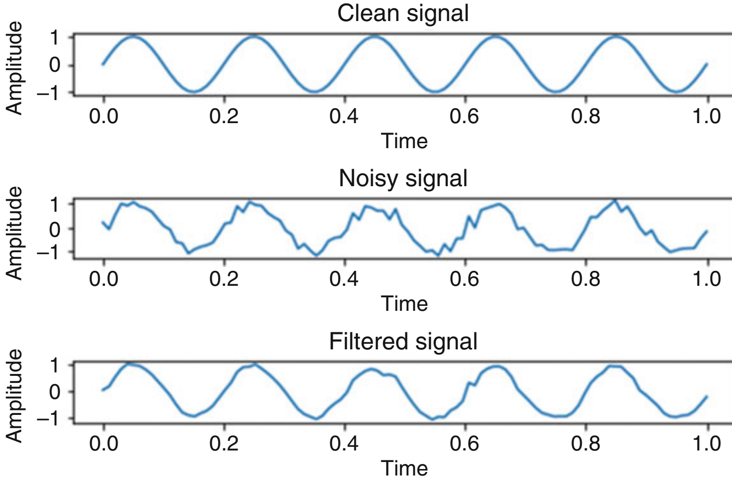


Fig. 11.5 Result of Wiener filtering

$$h_{\text{opt}} = R^{-1} p \tag{11.2}$$

The above equation can be expressed as

$$h_{\text{opt}} = \frac{p}{R} \tag{11.3}$$

In the above expression, 'p' represents the cross-correlation between desired signal and the observed signal, and 'R' represents the autocorrelation of the observed signal. Taking Fourier transform on both sides of Eq. (11.3), we get

$$\text{FT}\{h_{\text{opt}}\} = \frac{\text{FT}\{p\}}{\text{FT}\{R\}} \tag{11.4}$$

According to the Wiener-Khinchin theorem, Fourier transform of autocorrelation function gives power spectral density. Using this theorem, Eq. (11.4) is expressed as

$$H(e^{j\omega}) = \frac{S_{dx}(e^{j\omega})}{S_{xx}(e^{j\omega})} \tag{11.5}$$

In Eq. (11.5), $H(e^{j\omega})$ represents the frequency response of the Wiener filter, $S_{dx}(e^{j\omega})$ represents the cross-power spectral density estimation between desired and observed signal and $S_{xx}(e^{j\omega})$ represents the power spectral density of the observed signal.

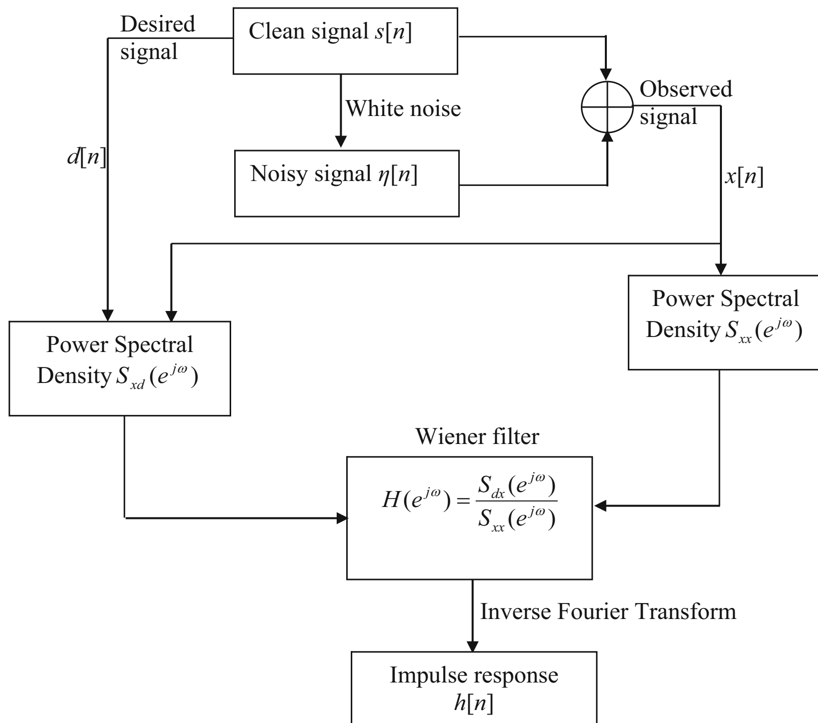


Fig. 11.6 Wiener filter in frequency domain

Experiment 11.3 Wiener Filter in Frequency Domain
The steps followed in the implementation of Wiener filter in frequency domain are given in Fig. 11.6. The noisy signal is obtained by adding white noise, which follows normal distribution to the clean signal. The observed signal is a clean signal with white noise added to it. The power spectral density of the observed signal is represented by $S_{xx}(\mathrm{e}^{j\omega})$. The power spectral density between the desired and observed signal is represented by $S_{dx}(\mathrm{e}^{j\omega})$. The Wiener filter is obtained in the frequency domain using the relation $H(\mathrm{e}^{j\omega}) = \frac{S_{dx}(\mathrm{e}^{j\omega})}{S_{xx}(\mathrm{e}^{j\omega})}$. Here the desired signal is the clean signal $s[n]$. Upon taking inverse Fourier transform of $H(\mathrm{e}^{j\omega})$, the impulse response of the Wiener filter is obtained.

The python code used to implement the Wiener filter in frequency domain is shown in Fig. 11.7, and the corresponding output is in Fig. 11.8.

The built-in functions used in the program and its purpose are given in Table 11.3.

Inference
From Fig. 11.8, the following observations can be made:

```
#Wiener filter in frequency domain
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from matplotlib import patches
t=np.linspace(0,1,100)
s=np.sin(2*np.pi*5*t) #Step1: Generation of clean signal s[n]
n=np.random.normal(0,0.1,len(t)) #Step 2: Generation of noise
x=s+n #Step 3: Generation of observed signal x[n]
Nh=25
f,Pxx=signal.csd(x,x,nperseg=Nh) #Step 4: Power spectral density of observed signal
f,Psx=signal.csd(s,x,nperseg=Nh) #Step 5: PSD of desired and observed signal
H=Psx/Pxx #Step 6: Wiener filter in frequency domain
h=np.fft.irfft(H) #Step 7: Wiener filter in time domain
w, H1 = signal.freqz(h, 1)
y=signal.filtfilt(h,1,x) #Step 8: Filtered signal
plot1 = plt. figure(1)
bx=plt.subplot(3,1,1)
bx.plot(t,s),bx.set(title='Clean signal',xlabel='Time',ylabel='Amplitude')
bx=plt.subplot(3,1,2)
bx.plot(t,x),bx.set(title='Noisy signal',xlabel='Time',ylabel='Amplitude')
bx=plt.subplot(3,1,3)
bx.plot(t,y),bx.set(title='Filtered signal',xlabel='Time',ylabel='Amplitude')
plt.tight_layout()
plot2 = plt. figure(2)
#Pole-zero plot of the filter
ax = plt.subplot(2,2,3);
unit_circle = patches.Circle((0,0),radius = 1 , fill = False,color='black',ls='solid',alpha = 0.1)
ax.add_patch(unit_circle),ax.axhline(0,color='black',alpha = 0.5)
ax.axvline(0,color='black',alpha = 0.5)
b,a = h,[1]
z,p,k = signal.tf2zpk(b,a)
ax.plot(np.real(z),np.imag(z),'or',label='zeros')
ax.plot(np.real(p),np.imag(p),'xb',label = 'poles')
ax.set(title='Zeros and poles',xlabel='$\sigma$', ylabel='$j\omega$'),ax.legend(loc = 2),ax.grid()
ax = plt.subplot(2,2,1)
ax.stem(h),ax.set(title='Impulse response',xlabel='n-->',ylabel='Amplitude')
ax = plt.subplot(2,2,2)
ax.plot(w/np.pi,20*np.log10(abs(H1))),
ax.set(title='Magnitude response',xlabel='w',ylabel='Magnitude')
ax=plt.subplot(2, 2, 4)
ax.plot(w/np.pi, 180/np.pi*np.unwrap(np.angle(H1)))
ax.set(title='Phase response',xlabel='w',ylabel='Phase'),plt.tight_layout()
```

Fig. 11.7 Python code to implement Wiener filter in frequency domain

1. The impact of noise is minimized by applying the Wiener filter.
2. The impulse response of the Wiener filter is not symmetric; hence, the phase response of the filter is not a linear curve.
3. From the magnitude response, it is possible to observe that the filter is a lowpass filter, and it performs smoothing actions to minimize the impact of noise.

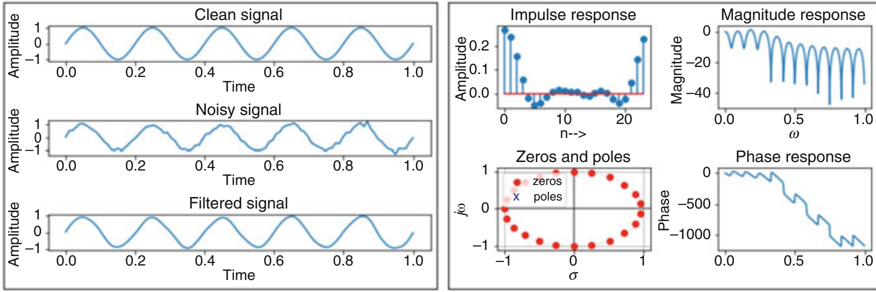


Fig. 11.8 Result and characteristics of Wiener filtering

Table 11.3 Built-in functions used in this experiment

S. No.	Objective	Built-in function	Library
1	To generate clean sinusoidal signal of 5 Hz frequency	np.sin()	Numpy
2	To add white noise which follows normal distribution to clean signal	np.random.normal()	Numpy
3	To compute the power spectral density	signal.csd()	Scipy
4	To compute the impulse response of the filter from the frequency response	np.fft.irfft()	Numpy
5	To obtain the frequency response of the filter	signal.freqz()	Scipy
6	To obtain the poles, zeros and the gain of the filter from the transfer function	signal.tf2zpk()	Scipy

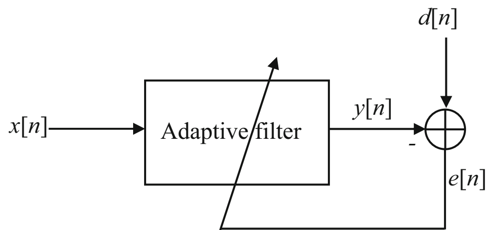


Fig. 11.9 General block diagram of adaptive filtering

4. From the pole-zero plot, it is possible to observe that poles and zeros lie within the unit circle; hence the filter is stable.

11.2 Adaptive Filter

The adaptive filter is a non-linear filter, which updates the value of the filter coefficients based on some specific criterion. The general block diagram of the adaptive filter is shown in Fig. 11.9. From this figure, it is possible to observe that

the filter coefficients are updated based on the error, $e[n]$ between the output of the filter $y[n]$ and reference data $d[n]$. Examples of adaptive filters are LMS filter and RLS filter.

11.2.1 LMS Adaptive Filter

The LMS is a least mean square algorithm that works based on the stochastic gradient descent approach to adapt the estimate based on the current error. The estimate is called the weight or filter coefficient. The weight or filter coefficient update equation of the LMS algorithm is given by.

$$w[n+1] = w[n] + \mu x[n]e[n] \tag{11.6}$$

where $w[n + 1]$ represents the new weight or updated weight, $w[n]$ denotes the old weight, μ indicates the step size or learning rate, $x[n]$ is the input signal or data and the error signal $e[n] = d[n] - y[n]$. $d[n]$ is the reference data or target data, and $y[n]$ is the actual output of the adaptive filter of the system.

Experiment 11.4 Implementation of LMS Algorithm
This experiment discusses the implementation of LMS algorithm for adaptive filtering using python. The python code to define the LMS algorithm as a function is shown in Fig. 11.10. This code can be called a function in the different applications of the LMS algorithm, which will be discussed in the subsequent experiments. From Fig. 11.10, it is possible to see that the weight updation formula of the LMS algorithm given in Eq. (11.6) exists in it.

Inference

1. From Fig. 11.10, it is possible to observe that the LMS algorithm is written as a function, and it can be called a signal processing application whenever needed.

```
# This python code for LMS algorithm
def LMS_algorithmm(x,mu,N,t):
    # x = input data, mu = step size, t = reference data, N = Filter length
    N1=len(x)
    w = np.zeros(N) # Initial filter
    e = np.zeros(N1-N)
    for n in range(0, N-F):
        xn = x[n+N:n:-1]
        en = t[n+N] - np.dot(xn,w) # Error
        w = w + mu * en * xn # Update filter (LMS algorithm)
        e[n] = en # Record error
    return w,e
```

Fig. 11.10 Python code for LMS algorithm

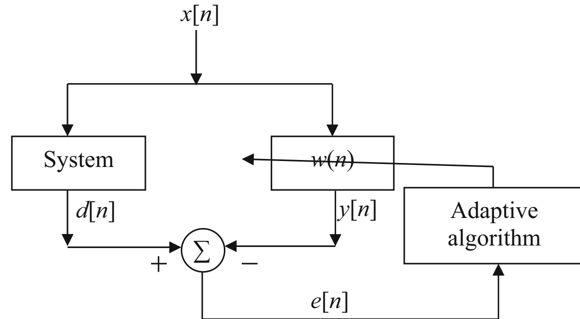


Fig. 11.11 Block diagram of system identification

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
N1 = 500 # Size of the Input data
N = 25 # Filter size
n_iter=[10,50,100,150]# it must be less than (N1-N)
x = np.random.randn(N1) # Input to the filter
h = signal.firwin(N, 0.25) # FIR filter to be identified
t = signal.convolve(x, h) # Target/desired signal
t = t + 0.01 * np.random.randn(len(t)) # with added noise
mu = 0.04 # LMS step size
plt.figure(),plt.title('Filter to be Identified'),plt.stem(h),plt.xlabel('n-->'),plt.ylabel('h[n]')
for i in range(0,len(n_iter)):
   [w,e]=LMS_algorithmm(x,mu,N,t,n_iter[i]);
   plt.figure(),plt.title('Error signal at iteration %d' % n_iter[i])
   plt.stem(e),plt.xlabel('n-->'),plt.ylabel('e[n]')
   plt.figure(),plt.title('Identified Filter at iteration %d' % n_iter[i])
   plt.stem(w),plt.xlabel('n-->'),plt.ylabel('w[n]')
```

Fig. 11.12 Python code for unknown system identification

2. The inputs to the LMS function are '*x*', '*mu*', '*N*' and '*t*'. '*x*' denotes the input data, '*mu*' represents step size, '*t*' denotes the reference data or target data and '*N*' indicates the length of the adaptive filter.
3. The outputs from this LMS function are '*w*', which denotes the adaptive filter coefficients, and '*e*' is an error between the estimate and target data.

Experiment 11.5 System Identification Using LMS Algorithm

This experiment deals with unknown system identification using the LMS algorithm. Let us consider the unknown system as an FIR filter with a length of 25. In this experiment, the output filter coefficients are obtained by using LMS algorithm with different number of iterations. The block diagram of the system identification is shown in Fig. 11.11. The python code to find the unknown system using the LMS algorithm is given in Fig. 11.12, and its simulation result is shown in Fig. 11.13.

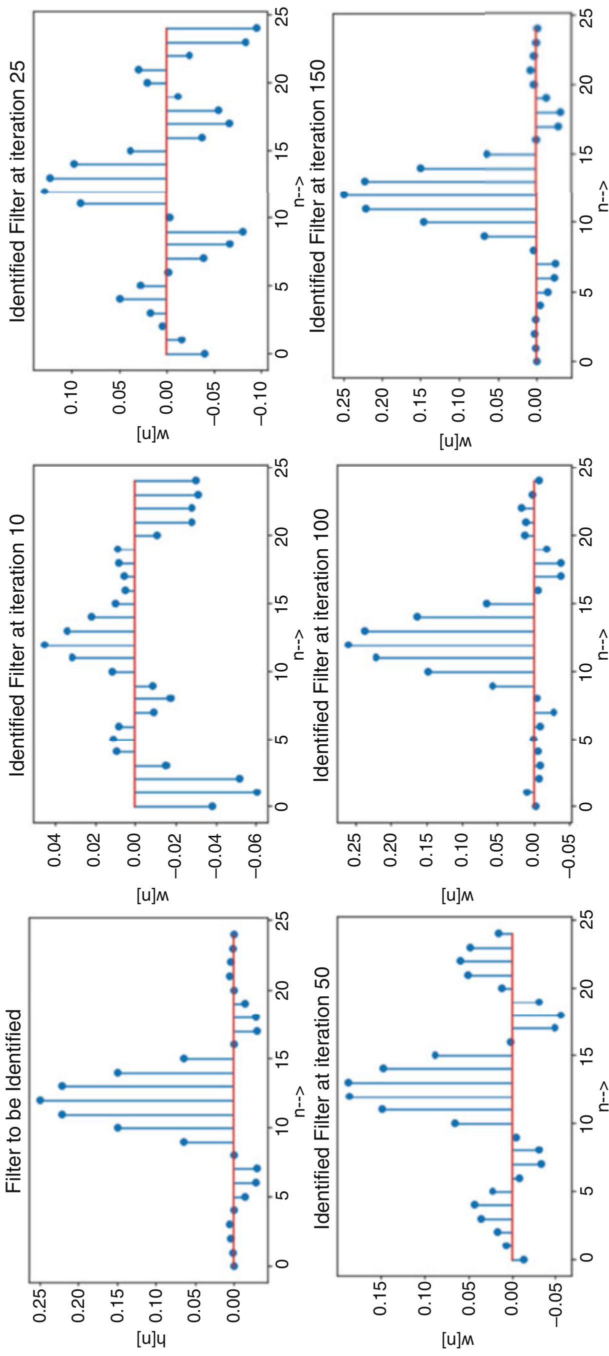


Fig. 11.13 Simulation results of Experiment 11.5

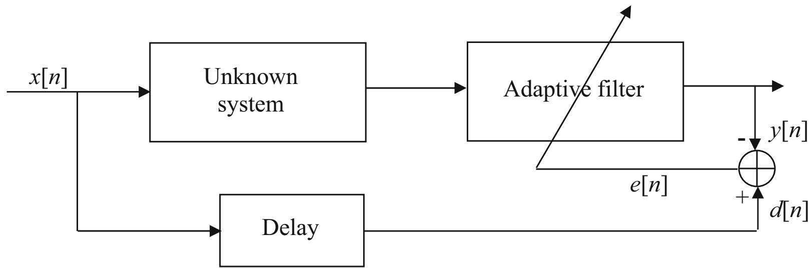


Fig. 11.14 Inverse system modelling using adaptive filter

Figure 11.12 indicates that the number of iterations is considered as 10, 50, 100 and 150, and the length of the unknown FIR filter is chosen as 25. The input to the LMS algorithm is a random signal with a length of 500 samples. The targeted or desired or reference data is obtained by convolving the input random signal with the unknown FIR filter coefficients along with the random noise.

Note that the inputs to the LMS algorithm ([*w*,*e*]=*LMS_algorithmm*(*x*,*mu*,*N*,*t*, *n_iter*[*i*])) are random signal (*x*), learning rate (*mu*), length of the filter (*N*), a reference signal (*t*) and number or iteration (*n_iter*). Also, note that the filter coefficients (*h*) are not given as input to the LMS algorithm. The outputs of the LMS algorithm are error signal (*e*) and identified filter output (*w*).

The simulation result of the python code given in Fig. 11.12 is displayed in Fig. 11.13.

Inference

From Fig. 11.13, it is possible to observe that the adaptive filter result approaches the original filter coefficients while increasing the number of iterations.

Task

Increase/decrease the length of the FIR filter and fix the number of iterations is 50. Comment on the observed result.

Experiment 11.6 Inverse System Modelling Using LMS Algorithm

This experiment discusses the inverse system modelling using LMS algorithm. The general block diagram of inverse system modelling using adaptive filter is shown in Fig. 11.14. From this figure, it is possible to understand that the unknown system and the adaptive filter are connected in a cascade form, and the delayed version of the input signal act as a reference signal. The aim of adaptive filtering in this experiment is to obtain the inverse system of the unknown system so that $y[n]$ and $d[n]$ will be similar. If $y[n]$ and $d[n]$ are similar, then the adaptive filter is equal to the inverse of the unknown system.

In communication systems, inverse system modelling is used as channel equalization. In such scenario, the adaptive filter is termed as ‘equalizer’. Adaptive equalizer can combat intersymbol interference. Intersymbol interference arises because of the spreading of a transmitted pulse due to the dispersive nature of the channel.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import fft
mu,W=0.04,2.2 # learning rate,Channel Capacity
filt_order,t_samples,delay,trial=7,200,4,1000
noise_var,data_var=0.001,1
for i in range(0,trial):
   inp=np.zeros(filt_order)
   data=np.zeros(filt_order+t_samples)
   v=np.zeros(filt_order+t_samples)
   w=np.zeros(filt_order)
   #Generation of random data and random noise
   for j in range(filt_order-1,t_samples+filt_order):
      data[j]=np.fix(np.random.rand(1)+0.5)*2-1
      v[j]=np.fix(np.random.rand(1)+0.5)*2*np.sqrt(noise_var)-np.sqrt(noise_var)
   # Impusle response of the channel
   h=np.zeros(3)
   for j in range(0,3):
      h[j]=(1/2)*(1+np.cos(2*np.pi/W)*(j-(3-1)))
   C_out=signal.convolve(h,data) # Output from Channel
   Err_square=np.zeros(len(C_out))
   data=np.append(np.zeros(len(h)-1), data)
   v=np.append(np.zeros(len(h)-1), v)
   C_outn=C_out+v;
   [w,e]=LMS_algorithmm(C_outn,mu,filt_order,data,len(C_outn)-filt_order);
   e=np.append(e,np.zeros(filt_order))
   Err_square=Err_square+(e**2)
mse=Err_square/trial
plt.figure,plt.subplot(2,2,1),plt.stem(h),plt.title('Impulse Resp. of Channel filter')
plt.xlabel('n-->'),plt.ylabel('h1[n]'),plt.subplot(2,2,2),plt.stem(w),
plt.title('Impulse Resp. of Inverse filter'),plt.xlabel('n-->'),plt.ylabel('h2[n]')
cas=signal.convolve(w,h);#Cascade operation
mag=fft(cas);#Frequency Response
plt.subplot(2,2,3),plt.stem(cas),plt.title('Impulse Resp. of Cascaded filter'),plt.xlabel('n-->'),
plt.ylabel('h1[n]*h2[n]'),plt.subplot(2,2,4),plt.plot(np.abs(mag)),
plt.title('Mag. Resp. of Cascaded filter'),plt.xlabel('$\omega$-->'), plt.ylabel('|H($\omega$)|'),
plt.ylim(0,10),plt.tight_layout()
```

Fig. 11.15 Python code for Inverse system modelling

The impulse response of the channel is given by

$$h[n] = \begin{cases} \frac{1}{2}\left[1 + \cos\left(\frac{2\pi}{W}(n-2)\right)\right], & n = 1, 2, 3 \\ 0, & \text{otherwise} \end{cases} \tag{11.7}$$

In the above equation, 'W' represents the channel capacity. Higher value of 'W' implies that the channel is more complex.

The python code to obtain the inverse of unknown system using LMS algorithm is given in Fig. 11.15, and its corresponding simulation result is shown in Fig. 11.16.

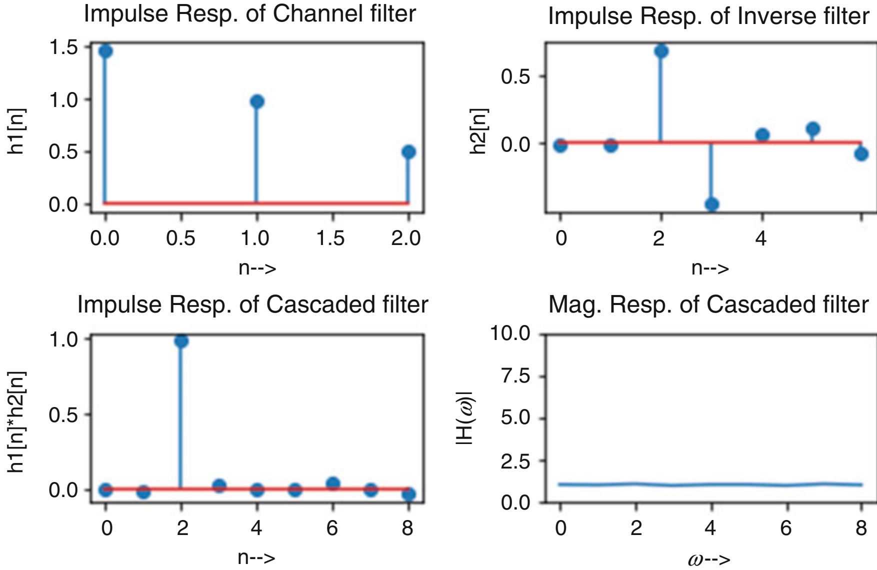


Fig. 11.16 Simulation result of inverse system modelling

Inference
From Fig. 11.16, it is possible to perceive the following facts

1. The impulse response of the cascaded system is an impulse. This implies that the cascade of channel filter and its inverse system results in an identity system.
2. The Fourier transform of an impulse response will result in a flat spectrum. This is obvious by observing the spectrum of the cascaded system.

Task
1. Increase the order of the adaptive filter and obtain the impulse response of the inverse system.

11.2.2 Normalized LMS Algorithm

The weight updation formula for the normalized LMS algorithm is given by

$$w[n+1] = w^T[n] + \frac{\beta}{\|x\|^2 + c} e[n]x[n] \tag{11.8}$$

where 'β' is a positive constant, which controls the convergence speed of the algorithm. 'c' is a small regularization parameter; it is added with the norm of the signal $x[n]$ to avoid the divide by zero error.

```
# This code for NLMS algorithm
def NLMS_algorithmm(x,N,t,beta,c,n_iter):
  # x = input data, N = Filter length t = reference data,
  # beta = Convergence parameter, c = regularization constant,
  # n_iter = number of iteration
  N1=len(x)
  w = np.zeros(N) # Initial filter
  e = np.zeros(N1-N)
  for n in range(0, n_iter):
    xn = x[n+N:n:-1]
    en = t[n+N] - np.dot(xn,w) # Error
    mu=beta/((xn*(np.transpose(xn)))+c)#Learning rate update
    w = w + mu * en * xn # Update filter (NLMS algorithm)
    e[n] = en # Record error
  return w,e
```

Fig. 11.17 Python code for NLMS algorithm

Experiment 11.7 Normalized LMS (NLMS) Algorithm

The python code for the normalized LMS algorithm is given in Fig. 11.17.

Inference

1. From Fig. 11.17, it is possible to observe that it is in the form of a function, and it can be called for the adaptive signal processing applications whenever required.
2. Also, it is possible to know that step size or learning rate is not given as a direct input to the function.
3. The step size is calculated using the input data, β and 'c'.

Experiment 11.8 Inverse System Modelling Using NLMS Algorithm

This experiment is a repetition of the inverse system modelling experiment, which was discussed earlier. Here, Experiment 11.6 is repeated with the same specifications, and NLMS is used for adaptive filtering instead of LMS algorithm. The python code of this experiment is shown in Fig. 11.18, and its corresponding simulation result is displayed in Fig. 11.19.

Inference

The following conclusions can be made from this experiment:

1. From this Fig. 11.19, it is possible to conclude that the cascade of channel and inverse filter gives the impulse response as unit impulse sequence.
2. The magnitude response confirms that the cascaded filter spectrum is a dc.
3. Therefore, the channel filter and the adaptive filter are inverse to each other.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import fft
c,beta,W=1.5,0.25,2.2 # learning rate,Channel Capacity
filt_order,t_samples,delay,trial=7,200,4,1500
noise_var,data_var=0.001,1
for i in range(0,trial):
   inp=np.zeros(filt_order)
   data=np.zeros(filt_order+t_samples)
   v=np.zeros(filt_order+t_samples)
   w=np.zeros(filt_order)
   #Generation of random data and random noise
   for j in range(filt_order-1,t_samples+filt_order):
      data[j]=np.fix(np.random.rand(1)+0.5)*2-1
      v[j]=np.fix(np.random.rand(1)+0.5)*2*np.sqrt(noise_var)-np.sqrt(noise_var)
   # Impusle response of the channel
   h=np.zeros(3)
   for j in range(0,3):
      h[j]=(1/2)*(1+np.cos(2*np.pi/W)*(j-(3-1)))
   C_out=signal.convolve(h,data) # Output from Channel
   Err_square=np.zeros(len(C_out))
   data=np.append(np.zeros(len(h)-1), data)
   v=np.append(np.zeros(len(h)-1), v)
   C_outn=C_out+v;
   [w,e]=NLMS_algorithmm(C_outn,filt_order,data,beta,c,len(C_outn)-filt_order);
   e=np.append(e,np.zeros(filt_order))
   Err_square=Err_square+(e**2)
mse=Err_square/trial
plt.figure,plt.subplot(2,2,1),plt.stem(h),plt.title('Impulse Resp. of Channel filter')
plt.xlabel('n-->'),plt.ylabel('h1[n]'),plt.subplot(2,2,2),plt.stem(w),
plt.title('Impulse Resp. of Inverse filter'),plt.xlabel('n-->'),plt.ylabel('h2[n]')
cas=signal.convolve(w,h);#Cascade operation
mag=fft(cas);#Frequency Response
plt.subplot(2,2,3),plt.stem(cas),plt.title('Impulse Resp. of Cascaded filter')
plt.xlabel('n-->'),plt.ylabel('h1[n]*h2[n]'),plt.subplot(2,2,4),plt.plot(np.abs(mag)),
plt.title('Mag. Resp. of Cascaded filter'),plt.xlabel('$\omega$-->'),
plt.ylabel('|H($\omega$)|'),plt.ylim(0,10),plt.tight_layout()
```

Fig. 11.18 Python code for Experiment 11.8

11.2.3 Sign LMS Algorithm

The weight updation formula for Sign LMS algorithm is given by

$$w[n+1] = w[n] + \mu \ \text{sign}\{e[n]x[n]\} \tag{11.9}$$

where ‘sign’ indicates the sign of the number, ‘$w[n + 1]$’ represents new weight and ‘$e[n]$’ denotes the error signal between target and estimated signal.

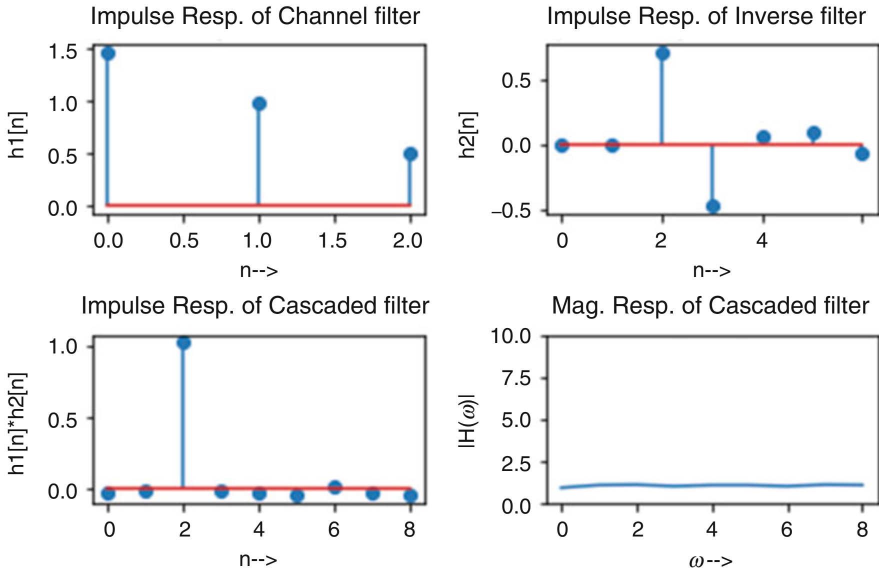


Fig. 11.19 Simulation result of the python code given in Fig. 11.18

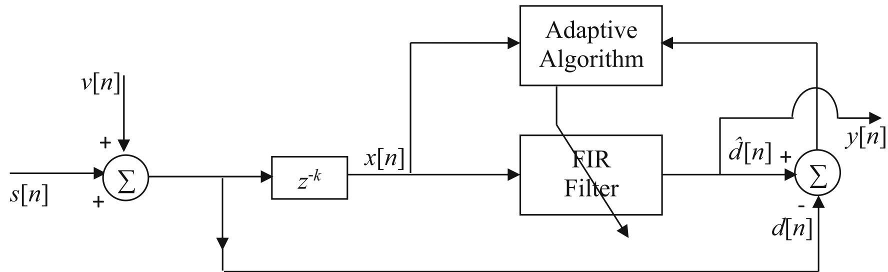


Fig. 11.20 Block diagram of adaptive line enhancer

Experiment 11.9 Adaptive Line Enhancer Using Sign LMS Algorithm

This experiment discusses the python implementation of adaptive line enhancer using sign LMS algorithm. The block diagram of adaptive line enhancer is shown in Fig. 11.20. From this figure, it is possible to observe that input to the FIR filter is a noisy version of the input signal ($x[n]$), and the final output ($y[n]$) is the enhanced input signal or noise-free signal. The aim of this experiment is to remove the noisy components present in the input signal using sign LMS adaptive algorithm. The python code for the "sign LMS algorithm" is given in Fig. 11.21 as a function.

The python code for adaptive line enhancer using sign LMS is given in Fig. 11.22. In this experiment, the input signal has 500, 2000 and 3500 Hz frequencies. The sampling frequency is considered as 8000 Hz. The input signal is added with the external random noise, which is the input to the adaptive filter. The number

```
# This Code for Sign LMS algorithm
def Sign_LMS_algorithmm(x,mu,N,t,n_iter):
   # x = input data, mu = step size, t = reference data, N = Filter length
   # n_iter = number of iteration
   N1=len(x)
   w = np.zeros(N) # Initial filter
   e = np.zeros(N1-N)
   for n in range(0, n_iter):
      xn = x[n+N:n:-1]
      en = t[n+N] - np.dot(xn,w) # Error
      w = w + mu * np.sign(en * xn) # Update filter (LMS algorithm)
      e[n] = en # Record error
   return w,e
```

Fig. 11.21 Python code for Sign LMS algorithm

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import fft
f1,f2,f3,Fs=500,2000,3500,8000 # Signal and sampling freq
T=1/Fs
t=np.arange(0,1,T)
noise=np.random.randn(len(t));
d=np.sin(2*np.pi*f1*t)+np.sin(2*np.pi*f2*t)+np.sin(2*np.pi*f3*t)+noise;
delay,N,mu=10,25,0.001 # Delay,Filter length and step size
x=np.append(np.zeros(delay),d);
 [w,e]=Sign_LMS_algorithmm(x,mu,N,d,len(t)-N)
y1=signal.convolve(w,x)
mag_x=fft(x)/len(x);#Frequency Response
mag_y=fft(y1)/len(y1);#Frequency Response
plt.figure(),plt.subplot(2,2,1),plt.plot(x),plt.title('Input noisy signal')
plt.xlabel('t-->'),plt.ylabel('x(t)')
plt.subplot(2,2,2),plt.plot(y1),plt.title('Denoised signal')
plt.xlabel('t-->'),plt.ylabel('y(t)')
plt.subplot(2,2,3),plt.plot(np.abs(mag_x[0:4000])),plt.title('Spectrum of noisy signal')
plt.xlabel('$\omega$-->'),plt.ylabel('|X($\omega$)|')
plt.subplot(2,2,4),plt.plot(np.abs(mag_y[0:4000])),plt.title('Spectrum of denoised signal')
plt.xlabel('$\omega$-->'),plt.ylabel('|Y($\omega$)|')
plt.tight_layout()
```

Fig. 11.22 Python code for adaptive line enhancer using sign LMS

of delay is chosen as 10, and length of the adaptive FIR filter is fixed as 25. The main objective of this experiment is to recover or enhance the original signal from the noisy input data using sign LMS algorithm. The simulation result of the python code

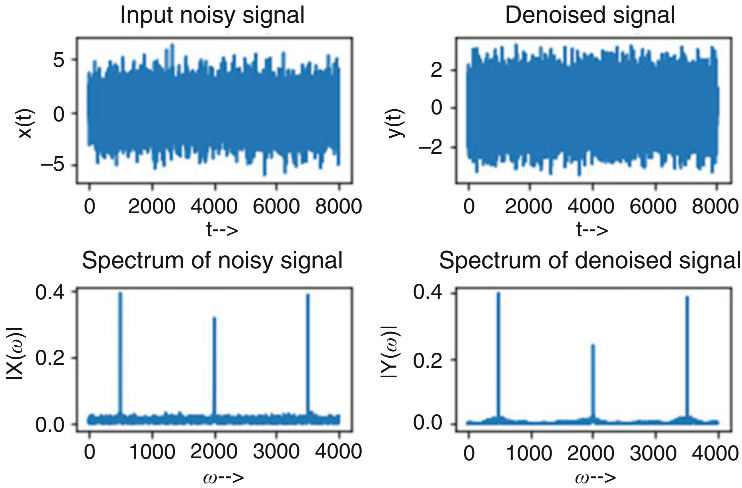


Fig. 11.23 Simulation result of the adaptive line enhancer using sign LMS

given in Fig. 11.22 is shown in Fig. 11.23. From the magnitude spectrum, it is possible to observe that the noise impact is reduced by the sign LMS algorithm.

Inference
From this experiment, the following observations can be drawn:

1. From Fig. 11.23, the magnitude response of the noisy signal indicates that the signal has three unique frequency components and noisy components.
2. The magnitude response of denoised signal has three spikes, and the impact of the noisy components is lesser than the input magnitude response.

Task
1. Do the suitable adjustments in the parameters used in the python code given in Fig. 11.22 to reduce the effect of noise in the denoised or enhanced signal?

11.3 RLS Algorithm

Recursive least square (RLS) is an adaptive algorithm based on the idea of least squares. The block diagram of the adaptive filter based on RLS algorithm is shown in Fig. 11.24. From the figure $x[n]$ is the input to the filter, $d[n]$ is the desired signal and the difference between the desired signal and the output of the filter is the error signal $e[n]$. Forgetting factor is used in RLS algorithm to remove or minimize the influence of old measurements. A small forgetting factor reduces the influence of old samples and increases the weight of new samples; as a result, a better tracking can be realized at the cost of a higher variance of the filter coefficients. A large forgetting factor

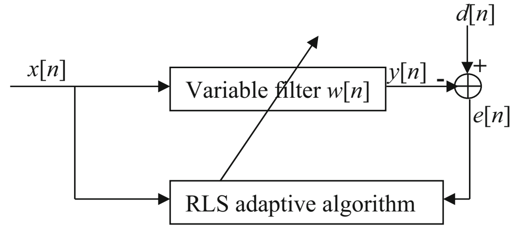


Fig. 11.24 Block diagram of adaptive filter based on RLS algorithm

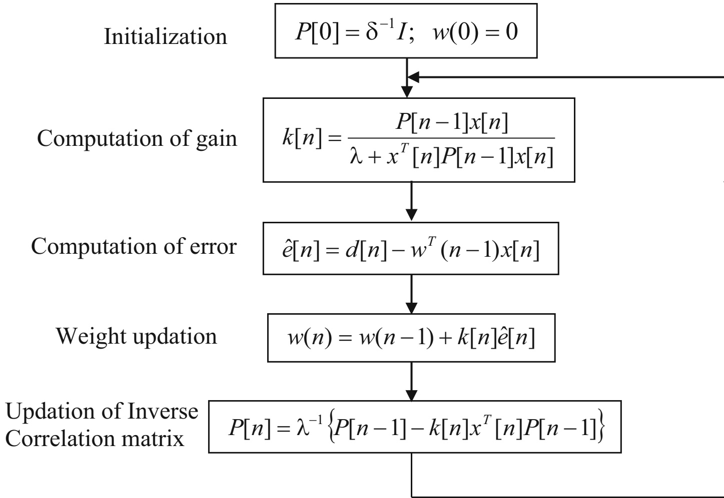


Fig. 11.25 Flow chart of sequence of steps in RLS algorithm

keeps more information about the old samples and has a lower variance of the filter coefficients, but it takes a longer time to converge.

Let us define the a priori error as $\hat{e}[n] = d[n] - w^T[n-1]x[n]$ and the weight updation formula for the RLS algorithm is given by

$$w[n] = w[n-1] + \frac{P[n-1]x[n]\hat{e}[n]}{\lambda + x^T[n]P[n-1]x[n]} \tag{11.10}$$

If $k[n] = \frac{P[n-1]x[n]}{\lambda + x^T[n]P[n-1]x[n]}$ represents the gain, then the above expression can be written as

$$w[n] = w[n-1] + k[n]\hat{e}[n] \tag{11.11}$$

The flow chart of the sequence of steps followed in RLS algorithm is shown in Fig. 11.25. From the flow chart, it is possible to observe that the algorithm is iterative. Proper initialization of filter coefficients is necessary for convergence.

```
# This Code for RLS algorithm
def RLS_algorithmm(x,lamda,delta,N,t,n_iter):
    # x = input data, lamda = Forgetting factor, delta = Regularization parameter
    # t = reference data, N = Filter length, n_iter = number of iteration
    N1=len(x)
    w = np.zeros(N) # Initial filter
    w=np.transpose(w)
    e = np.zeros(N1-N)
    P=np.eye(N)/delta
    x=np.transpose(x)
    for n in range(0, n_iter):
        xn = x[n+N:n:-1]
        k1=np.dot(P,xn)
        k2=np.dot(np.transpose(xn),P)
        k3=np.dot(k2,xn)
        k =k1/(lamda+k3)
        en = t[n+N] - np.dot(np.transpose(w),xn);# Error
        w = w + np.dot(k,np.conjugate(en)) # Update filter (RLS algorithm)
        P=(1/lamda)*P
        e[n] = en # Record error
    return w,e
```

Fig. 11.26 Python code for RLS algorithm

Experiment 11.10 Implementation of RLS Algorithm
This experiment discusses the implementation of RLS algorithm using python. The python code for RLS algorithm is given in Fig. 11.26, and it is in the form of a function so that this function can be used for different applications.

Experiment 11.11 Adaptive Line Enhancer Using RLS Algorithm
This experiment is a repetition of Experiment 11.9; instead of sign LMS, RLS algorithm is used to filter out the noisy component present in the input signal. The python code for this experiment is given in Fig. 11.27, and its corresponding simulation result is displayed in Fig. 11.28.

Inference
From Fig. 11.28, it is possible to confirm that the magnitude response of the filtered or denoised output is better than the magnitude response of the noisy input. Therefore, RLS algorithm can act as an adaptive line enhancer.

Experiment 11.12 Comparison of System Identification with Different Adaptive Filters
The main objective of this experiment is to compare the simulation result of different adaptive algorithms like LMS, NLMS, Sign LMS and RLS for the system identification process. The python code to compare the simulation results of system identification is given in Fig. 11.29, and its simulation results are shown in Fig. 11.30.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import fft
f1,f2,Fs=500,2000,8000 # Signal and sampling freq
T,lamda,delta=1/Fs,1.9,0.05
t=np.arange(0,1,T)
noise=np.random.randn(len(t));
d=np.sin(2*np.pi*f1*t)+np.sin(2*np.pi*f2*t)+noise;
delay,N=10,50 # Delay,Filter length
x=np.append(np.zeros(delay),d);
[w,e]=RLS_algorithmm(x,lamda,delta,N,d,len(d)-N)
y1=signal.convolve(w,x)
mag_x=fft(x)/len(x);#Frequency Response
mag_y=fft(y1)/len(y1);#Frequency Response
plt.figure(),plt.subplot(2,2,1),plt.plot(x),plt.title('Input noisy signal')
plt.xlabel('t-->'),plt.ylabel('x(t)')
plt.subplot(2,2,2),plt.plot(y1),plt.title('Denoised signal')
plt.xlabel('t-->'),plt.ylabel('y(t)')
plt.subplot(2,2,3),plt.plot(np.abs(mag_x[0:4000])),plt.title('Spectrum of noisy signal')
plt.xlabel('$\omega$-->'),plt.ylabel('|X($\omega$)|')
plt.subplot(2,2,4),plt.plot(np.abs(mag_y[0:4000])),plt.title('Spectrum of denoised signal')
plt.xlabel('$\omega$-->'),plt.ylabel('|Y($\omega$)|')
plt.tight_layout()
```

Fig. 11.27 Python code for adaptive line enhancer using RLS

Inference

From Fig. 11.30, it is possible to observe that proper selection of the adaptive filter parameters like step size or learning rate, forgetting factor and regularization plays a major role in using the adaptive filtering algorithm for the system identification application in signal processing.

Task

Write a python code to compare the simulation result of different adaptive algorithms like LMS, NLMS, sign LMS and RLS for adaptive line enhancement application in signal processing.

Exercises

1. Execute the python code given in Fig. 11.12 and compare the estimated filter '*w*' with the original filter coefficients '*h*' for different length of the filter. Also, execute the same python code and comment on the convergence of the LMS algorithm with different values of learning rate 'mu', including negative value.
2. Use the python code for the sign LMS algorithm given in Fig. 11.22 to compute the impulse response of the inverse filter and comment on the role of learning rate.
3. Modify the sign LMS algorithm based on the equation of the sign regressor algorithm is given by $w[n+1] = w[n] + \mu e[n]$ sign $\{x[n]\}$, and compute the impulse response of the inverse filter and comment on the simulation result.

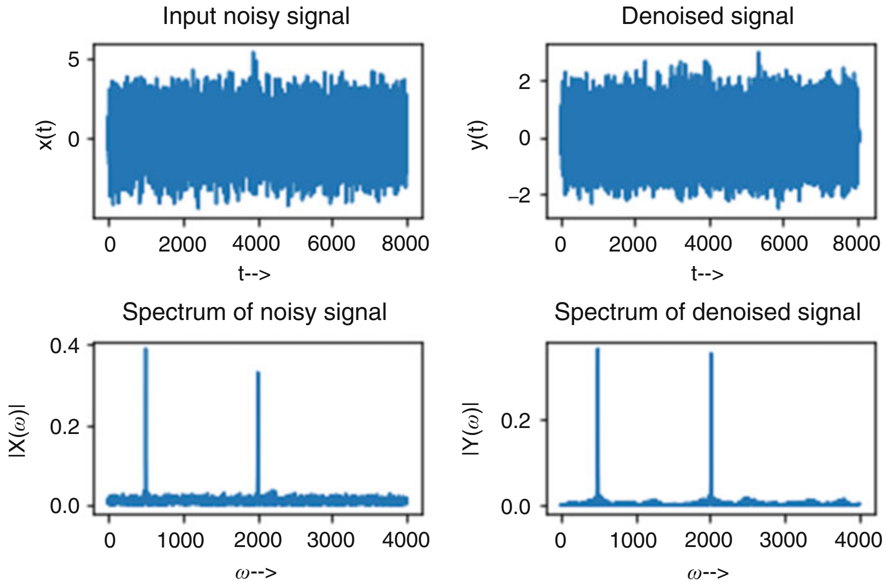


Fig. 11.28 Simulation result of the python code given in Fig. 11.27

```
# Python code for the comparison of adaptive algorithms for system identification
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
N1 = 1500 # Size of the Input data
N = 25 # Filter size
n=np.arange(0,N,1)
n_iter=200# it must be less than (N1-N)
x = np.random.randn(N1) # Input to the filter
h = signal.firwin(N, 0.25) # FIR filter to be identified
t = signal.convolve(x, h) # Target/desired signal
t = t + 0.01 * np.random.randn(len(t)) # with added noise
mu,mu1,beta,c,lamda,delta = 0.05,0.0005,0.05,1.5,1,0.25 # LMS step size
plt.figure(1),plt.title('Filter to be Identified')
plt.stem(h),plt.xlabel('n-->'),plt.ylabel('h[n]')
[w,e]=LMS_algorithmm(x,mu,N,t,n_iter);
[w1,e1]=NLMS_algorithmm(x,N,t,beta,c,n_iter)
[w2,e2]=Sign_LMS_algorithmm(x,mu1,N,t,n_iter)
[w3,e3]=RLS_algorithmm(x,lamda,delta,N,t,n_iter)
plt.figure(2),plt.subplot(2,2,1),plt.stem(n,w,'g'),plt.xlabel('n-->'),plt.ylabel('w[n]')
plt.title('Identified by LMS'),plt.subplot(2,2,2),plt.stem(n,w1,'k'),plt.xlabel('n-->'),
plt.ylabel('w[n]'),plt.title('Identified by NLMS'),plt.subplot(2,2,3),
plt.stem(n,w2,'r'),plt.xlabel('n-->'),plt.ylabel('w[n]'),plt.title('Identified by Sign LMS')
plt.subplot(2,2,4),plt.stem(n,w3,'b'),plt.xlabel('n-->'),plt.ylabel('w[n]')
plt.title('Identified by RLS'),plt.tight_layout()
```

Fig. 11.29 Python code for unknown system identification

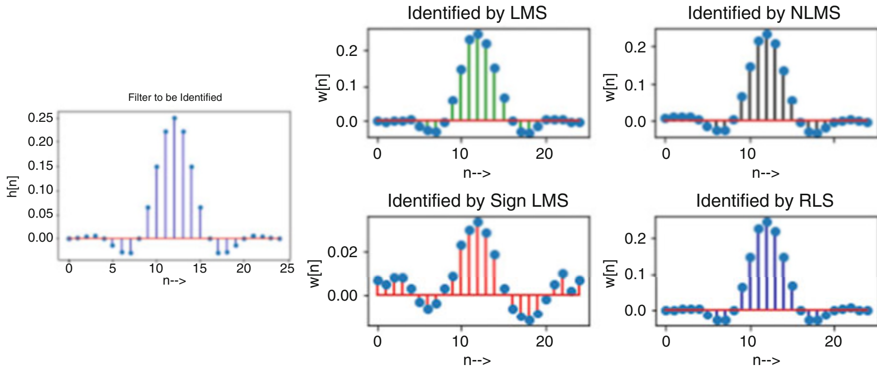


Fig. 11.30 Simulation result of the python code given in Fig. 11.29

4. Modify the sign LMS algorithm based on the equation of sign-sign LMS algorithm is given by $w[n+1] = w[n] + \mu$ sign $\{e[n]\}$ sign $\{x[n]\}$, and compute the impulse response of the inverse filter and comment on the simulation result.
5. Use the python code for RLS algorithm given in Fig. 11.26 to obtain the inverse filter coefficients and comment on the simulation result. Also, comment on the selection of the forgetting factor and regularization parameter.

Objective Questions

1. The filter which is based on the minimum mean square error criterion, is

   A. Wiener filter
   B. Window-based FIR filter
   C. Frequency sampling-based FIR filter
   D. Savitsky Golay filter

2. If 'R' is the autocorrelation matrix of the observed signal and 'p' represents the cross-correlation between the desired signal and the observed signal, then the expression for the Wiener-Hopf equation is

   A. $w_{\text{opt}} = R \times p$
   B. $w_{\text{opt}} = R + p$
   C. $w_{\text{opt}} = R - p$
   D. $w_{\text{opt}} = p/R$

3. The weight update expression of the standard LMS algorithm is

   A. $w(n+1) = w(n) + \mu x[n]e[n]$
   B. $w(n+1) = w(n) - \mu x[n]e[n]$
   C. $w(n+1) = w(n) + \mu x[n]e^2[n]$
   D. $w(n+1) = w(n) - \mu x[n]e^2[n]$

4. If μ refers to the step size and λ refers to the eigen value of the autocorrelation matrix, then the condition for convergence of LMS algorithm is given by

   A. $0 < \mu < \frac{2}{\lambda_{\min}}$
   B. $0 < \mu < \frac{2}{\lambda_{\max}}$
   C. $0 < \mu < \frac{2}{\lambda_{\max}^2}$
   D. $0 < \mu < \frac{2}{\lambda_{\min}^2}$

5. Statement 1: Wiener filter is based on the statistics of the input data.
   Statement 2: Wiener filter is an optimal filter with respect to minimum mean absolute error

   A. Statements 1 and 2 are true
   B. Statement 1 is correct, and Statement 2 is wrong
   C. Statement 1 is wrong, Statement 2 is correct
   D. Statements 1 and 2 are wrong

6. The filter which changes its characteristics in accordance with the environment is termed as

   A. Optimal filter
   B. Non-linear filter
   C. Adaptive filter
   D. Linear filter

Bibliography

1. Simon Haykin, “Adaptive Filter Theory”, Pearson, 2008.
2. Bernard Widrow, Samuel D. Stearns, “Adaptive Signal Processing”, Pearson, 2002.
3. Dimitris G. Manolakis, Vinay K. Ingle, and Stephen M. Kogon, “Statistical and Adaptive Signal Processing: Spectral Estimation, Signal Modeling, Adaptive Filtering and Array Processing”, Artech House Publishers, 2005.
4. Behrouz F. Boroujey, “Adaptive Filters: Theory and Applications”, Wiley -Blackwell, 2013.
5. Alexandar D. Poularikas, “Adaptive Filtering”, CRC Press, 2015.