ECE 5650/4650 Computer Project 2
Adaptive Filters

This project is to be treated as a take-home exam, meaning each student or student **team of at most three** (undergraduates students only) is to do his/her own work without consulting others. The grading for this second project is merged with the first project to create the *projects grade category*. The computer projects category can count up to 20% percent of the final grade (details given below). **The project due date is 12:00 PM Thursday, December 19, 2024 (Final Exam week)**.

From the syllabus the grading options are as follows: (1) Computer projects 20%, final exam 25%; (2) Computer project #2 15%, final exam 30%. The homework and exam percentages are constant over the two options.

Introduction

In this project you will be investigating adaptive noise cancellation (correlation cancellation) techniques using adaptive FIR and adaptive IIR filters. **For adaptation you use the popular least mean square (LMS) algorithm, today found in machine learning.** The input signal will be modeled as a real sinusoid(s) in additive white noise. Specifically the systems investigated are called *adaptive line enhancers* or ALEs. A rework of the ALE for adaptive interference cancellation, is also considered using a real speech waveform. A sample Jupyter notebook can be found in the accompanying ZIP package `Project2_f2024_old.zip`.

Background Theory

A common estimation theory problem is to estimate a random process of interest (signal) from an observed random process (e.g. signal plus noise). A solution is to use a causal minimum mean-square error (MMSE) filter (Wiener filter) to process the observations. A discrete-time Wiener filter takes the form of an FIR filter with $M + 1$ weights.

Let $y[n]$ be the observations, $x[n]$ be the signal to estimate, and a_m be the filter weights. The MMSE estimate is of the form

$$\hat{x}[n] = \sum_{m=0}^{M} a_m y[n-m] \tag{1}$$

The weights an, a_m, $m = 0, 1, \ldots, M$ are chosen such that

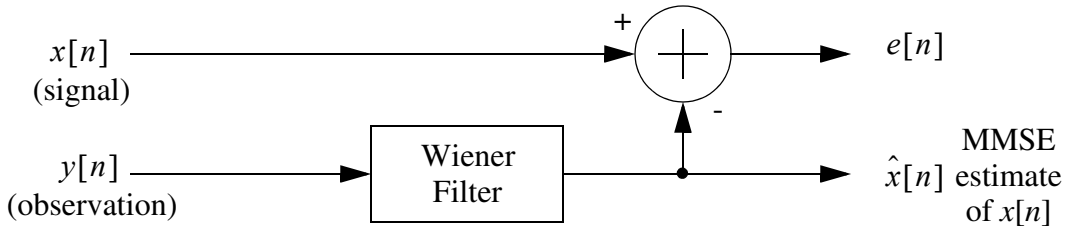

Figure 1: Figure 1 Wiener (MMSE) Filter.

$$E\{e^2[n]\} = E\{(x[n]-\hat{x}[n])^2\} \tag{2}$$

is minimized. A block diagram of the filter is shown in [1].

The optimal weights are found by solving the system of linear equations defined by

$$\frac{\partial}{\partial a_m}E\{e^2[n]\} = 0, m = 1, 2\ldots, M \tag{3}$$

The solution is a result of the projection theorem or orthogonality principle[1], which says that we choose constants a_m such that the error $e[n]$ is orthogonal to the observations (*data*), i.e.,

$$E\{(x[n]-\hat{x}[n])y[n-k]\} = 0, k = 0, 1, \ldots, M \tag{4}$$

Expanding the above we have

$$\sum_{m=0}^{M} a_m \phi_{yy}[k-m] = \phi_{xy}[k], k = 0, 1, \ldots, M \tag{5}$$

which are known as the *normal equations*, or in Papoulis the Yule-Walker equations. The function $\phi_{yy}[m]$ is the autocorrelation sequence corresponding to $y[n]$ and $\phi_{xy}[m]$ is the cross correlation sequence between $x[n]$ and $y[n]$.

In an adaptive Wiener filter the error signal $e[n]$ is fed back to the filter weights to adjust them using a *steepest-descent algorithm*. Note that the error surface generated by $e[n]$ over the $M+1$ parameter space $\{a_m\}$, is convex cup (i.e. a bowl shape) as shown in [2]. The filter decorrelates the output error $e[n]$ so that signals in common to both $x[n]$ and $y[n]$ in a correlation sense are *canceled*. A block diagram of an adaptive FIR filter is shown in [3].

In the ALE the signals of the adaptive Wiener filter are redefined slightly. Let $x[n]$ be the *noisy* signal and $y[n]$ be a delayed replica of $x[n]$. The general structure of the ALE is shown in [4].

We assume that $x[n]$ consists of a narrowband component (e.g. sinusoid) and a broadband com-

1. A. Papoulis and S. Pillai, *Probability, Random Variables, and Stochastic Processes, fourth edition*, McGraw-Hill, New York, 2002.

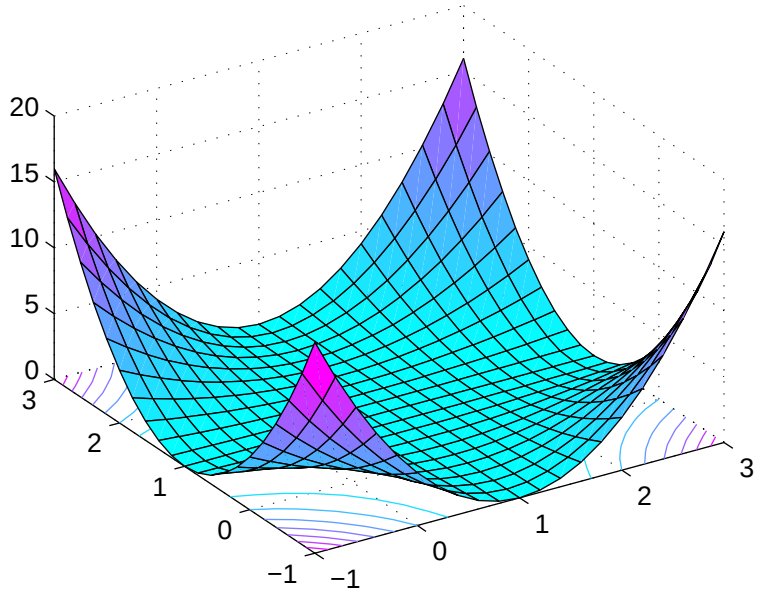

Figure 2: Error surface for $M = 1$ with the optimum values (1.0,1.0)

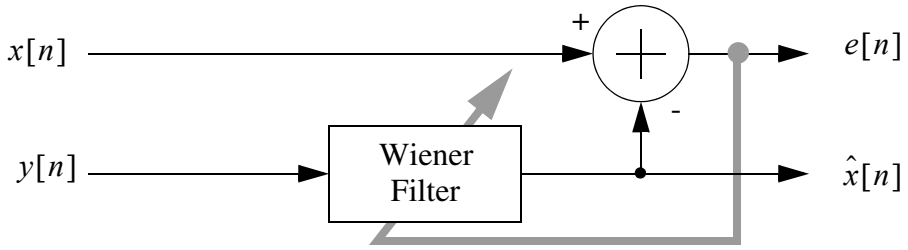

Figure 3: Adaptive Wiener (MMSE) Filter

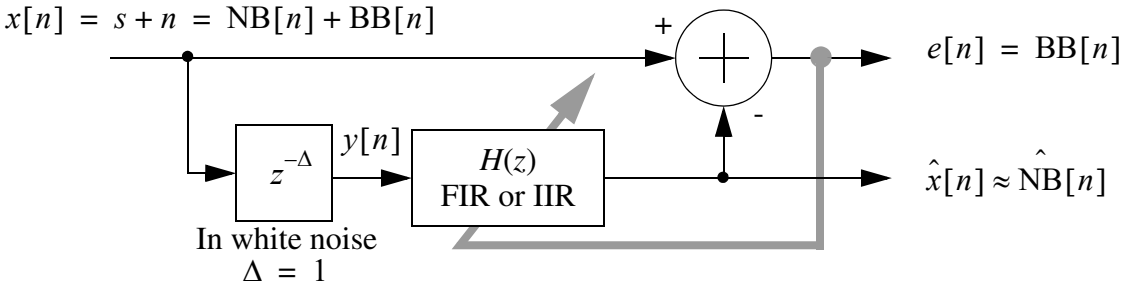

Figure 4: General form of the adaptive line enhancer

ponent (e.g. white noise). By setting the delay Δ to be greater than the decorrelation time of the broadband component (for white noise $\Delta = 1$), and less than the decorrelation time of the narrowband signal, the adaptive filter will correlation cancel only the narrowband component from $x[n]$. Thus the output $e[n]$ will contain primarily the broadband signal and $\hat{x}[n]$ will contain

primarily the *desired* narrowband signal.

Note that since filter coefficients change over time, thus the filter has a time-varying impulse response. Linearity still holds.

FIR Structure and Adaptation Equations

Here the filter input/output relationship is

$$\hat{x}[n] = \sum_{m=0}^{M} a_m[n]x[n-m-\Delta] = \sum_{m=0}^{M} a_m[n]y[n-m] \tag{6}$$

The coefficient notation is saying that at time n the filter coefficients are $a_m[n]$, $m = 0, 1, \ldots, M$. The computational algorithm known as the Widrow-Hoff *least-mean square* (LMS) adaptation algorithm is composed of three steps:

1. $\hat{x}[n] = \sum_{m=0}^{M} a_m[n]y[n-m]$
2. $e[n] = x[n] - \hat{x}[n]$
3. $a_m[n+1] = a_m[n] + 2\mu e[n]y[n-m], m = 0, 1, \ldots, M$

The third equation is the actual coefficient update equation. The parameter μ is known as the convergence parameter. This parameter must be properly chosen to at one extreme insure convergence (i.e. algorithm stability), and at the other extreme insure that convergence occurs rapidly. Typically the filter is initialized with all coefficients set equal to zero.

Selection of μ

Formally, the convergence of the LMS algorithm requires that

$$0 < \mu < \frac{1}{\lambda} \tag{7}$$

for each eigenvalue of the autocorrelation matrix **R**. Here **R** is defined as

$$\mathbf{R} = E\{\mathbf{y}[n]\mathbf{y}^t[n]\} \tag{8}$$

where

$$\mathbf{y}[n] = \begin{bmatrix} y[n] & y[n-1] & \cdots & y[n-M] \end{bmatrix}^t \tag{9}$$

is a vector of signal samples at the filter input. Note however that the maximum eigenvalue upper bounds μ, and the maximum eigenvalue is bounded by the trace of the autocorrelation matrix, e.g.,

$$\lambda_{\max} < \text{Trace}[\mathbf{R}] = \sum_{m=0}^{M} \phi_{yy}[0] = (M+1)\phi_{yy}[0] \tag{10}$$

A more conservative upper bound is thus

$$\mu < \frac{1}{(M+1)\phi_{yy}[0]} \tag{11}$$

Further note that since $\phi_{yy}[0] = \sigma_y^2$ is just the power in the signal, $y[n]$, it is not that difficult to calculate the bound of (11). Since the convergence parameter changes with input signal power, we may replace μ with a normalized value which depends on the input signal power, e.g.,

$$\mu_n = \frac{\mu}{(M+1)\sigma_y^2}, \; 0 < \mu < 1 \tag{12}$$

which forms a self regulating version of μ, much like having an automatic gain control (AGC) to keep the LMS algorithm from becoming unstable when operating in environments where both signal and noise power levels vary. The quantity σ_y^2 is the input signal power (signal plus noise). If desired we can form an updating estimate for σ_y^2 using the observed filter input directly

$$\hat{\sigma}_y^2[n] = \alpha y^2[n] + (1-\alpha)\hat{\sigma}_y^2[n-1] \tag{13}$$

where $y[n]$ is the current input sample to the filter, and α is a forgetting factor chosen as $0 < \alpha \ll 1$. This is a very simple recursive estimator for the variance or power of a signal.

IIR Structure and Adaptation Equations

Here the filter system function is a second-order bandpass resonator containing just two control parameters

$$H(z) = \frac{\left(\frac{1-r^2}{1+r^2}\right)w - (1-r^2)z^{-1}}{1 - wz^{-1} + r^2 z^{-2}}, \quad \begin{array}{l} 0 \ll r < 1 \text{ bandwidth parameter} \\ -2r < w < 2r \text{ center freq. parameter} \end{array} \tag{14}$$

It can be shown that the bandpass center frequency is given by

$$\gamma = \cos^{-1}\left(\frac{w}{1+r^2}\right) \tag{15}$$

In practice we may let r be fixed and then w becomes the only filter parameter to adaptively adjust. That is $w \rightarrow w[n]$. The maximum allowable range for γ is

$$\cos^{-1}\left(\frac{2r}{1+r^2}\right) < \gamma < \cos^{-1}\left(\frac{-2r}{1+r^2}\right) \tag{16}$$

The adaptation algorithm is the following[1]:

1. $\hat{x}[n] = (1-r^2)/(1+r^2)w[n]y[n] - (1-r^2)y[n-1] + w[n]\hat{x}[n-1] - r^2\hat{x}[n-2]$
2. $e[n] = x[n] - \hat{x}[n]$
3. $w[n+1] = w[n] + \rho e[n]\alpha_{\text{norm}}[n]$

where ρ is the convergence parameter and $\alpha_{\text{norm}}[n]$ is obtained recursively via

$$\alpha_{\text{norm}}[n] = \frac{\alpha[n]}{\psi[n]} \tag{17}$$

where

$$\alpha[n] = \left(\frac{1-r^2}{1+r^2}\right)y[n] + \hat{x}[n-1] \tag{18}$$

and

$$\psi[n] = \nu\psi[n-1] + (1-\nu)\alpha^2[n] \tag{19}$$

In (19) ν is a forgetting factor in the range $0 \ll \nu < 1$ (ν around 0.9 works well). Due to the division in (17), make sure that you start the power estimate with some nonzero initial condition, e.g., $\psi[0] = \psi[-1] = 1$. When using the normalized form of α, the gradient estimate is more likely to have the same sign (direction) as the true gradient, thus faster convergence can be achieved. It is worth noting that $\psi[n]$ forms an estimate of the power in $\alpha[n]$ and hence why the expression of (17) is termed normalized.

Selection of ρ

Selection of ρ is important to insure proper convergence of the algorithm. In the paper by Hush, referenced earlier, they found values ranging from 0.01 to 0.001 worked well. An analytical solution is difficult to obtain for the IIR ALE.

Problems

In this investigation of adaptive line enhancers you will conduct experiments for both an FIR and IIR implementation.

1. Write a computer simulation using Python to implement the FIR ALE. Your Python function (see the template in the sample notebook `5650 Final Project_Sample.ipynb`) should work for any value of $M > 16$ to be as large as 255 (i.e. $M + 1 = 256$ taps). Let the input signal be

1. Don R. Hush, Nasir Ahmed, Ruth Davis, and Samuel Stearns, "An Adaptive IIR Structure for Sinusoidal Enhancement, Frequency Estimation, and Detection," *IEEE Trans., Acoust., Speech, Signal Processing*, vol ASSP-34, pp. 1380–1390, December 1986.

$$x[n] = A\cos(\omega_o n) + w[n] \tag{20}$$

where $\omega_o = \pi/10$ and $w[n]$ is a zero mean white Gaussian noise process with variance σ_w^2.

In Python you can generate a sequence of white Gaussian noise, variance σ_w^2, using

```
w = sqrt(var_w)*randn(N+1) #generate N+1 samples of noise
```

Define the input signal-to-noise ratio (SNR) to be

$$\text{SNR} = \frac{A^2}{2\sigma_w^2}, \text{SNR}_{\text{dB}} = 10\log_{10}[\text{SNR}] \tag{21}$$

In your simulation it is best to fix $A = 1$ and let σ_w^2 vary with the desired SNR value in dB. Initialize the filter coefficient values to be zero.

a.) For SNR = 10 dB and $M = 256$ observe the MSE convergence versus n by plotting $|e[n]|^2$ for about $0 \le n \le 600$. Compare the enhancement provided by the ALE by plotting $x[n]$ and $\hat{x}[n]$ for about $1000 \le n \le 1500$. At this point in time the filter should have converged nicely. Plot the magnitude frequency response in dB of the FIR filter for the final value of filter coefficients, when you simulation stops, i.e., $n = 1500$, this is just the Fourier transform of the FIR filter at time n

$$20\log_{10}|H(e^{j\omega})| = 20\log_{10}\left|\sum_{m=0}^{M} a_m[n]e^{-j\omega m}\right| \tag{22}$$

In Python `signal.freqz()` makes the calculation and plotting easy. Plot the filter gain in dB versus the normalized value of ω, e.g., $\omega/(2\pi)$ (`F` in the code). To insure reliable convergence make sure that μ is small enough. Comment on your results. The filter should form a tight passband around the sinusoid frequency ω_0. Verify that the passband is centered about the normalized frequency $f_0 = 0.05$, where $f_0 = \omega_0/(2\pi)$.

b) Repeat part (a) with SNR = 0 dB. Comment on your results.

2. Repeat Problem 1 in its entirety except now let the signal component of the input be a squarewave of amplitude $\pm A$, 50% duty cycle, and sequence period $N = 20$. Redefine the SNR to be given by

$$\text{SNR} = \frac{A^2}{\sigma_w^2}, \tag{23}$$

so that SNR is still the ratio of signal power over noise power. Comment on your results. Since the squarewave has odd harmonics, the filter should form multiple passbands in an attempt to *clean* the noise off of $x[n]$.

3. **Optional Bonus Problem for 24 points**. Write a computer simulation to implement the IIR

ALE. Your program should allow for variable r, even though you will be asked to use a particular fixed value when you run simulations. Let the input signal be

$$x[n] = A\cos(\omega_o n) + w[n] \tag{24}$$

where again $\omega_o = \pi/10 = 2\pi/20$ and $w[n]$ is a zero mean white Gaussian noise process with variance σ_w^2. Define the SNR as in (22). Initialize the filter with $w = 0$

a.) For SNR = 10 dB and $r = 0.9$ observe the MSE convergence versus n by plotting $|e[n]|^2$ for $0 \le n \le 600$. Compare the enhancement provided by the ALE by plotting $x[n]$ and $\hat{x}[n]$ for $1000 \le n \le 1500$. Plot the magnitude frequency response of the IIR filter for when the simulation has stopped, e.g., $n = 1500$. Also calculate the approximate sinusoid frequency in terms of the filters parameters r and w for $n = 1500$. It will be helpful to monitor $w[n]$ over the entire simulation so that you can see the convergence trajectory of this parameter. The action of $w[n]$ is somewhat like the control voltage on a VCO in a phase-locked loop (PLL).

Adaptive Interference Cancellation

4. The adaptive filter structure of Figure 4 can be utilized for narrowband interference cancellation as shown in Figure 5 by using the error output, $e[n]$, to obtain an estimate of the broadband input component in $x[n]$. This is an **easy extension** to the code you have devel-

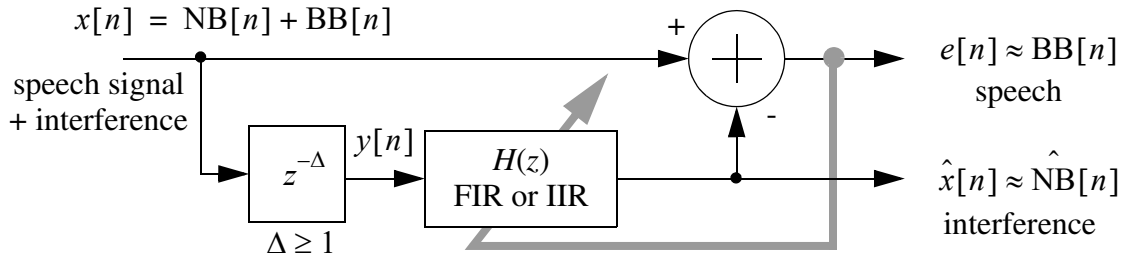

Figure 5: Adaptive interference canceling (AIC) filter

oped in Problem 1. In this problem the broadband signal will be an 8 ksps speech signal and the narrowband signal will be a sinusoid, as before. In Figure 5, the signal roles are now reversed compared with Figure 4, in the sense that the desired signal is now broadband, and the *noise/interference* is narrowband. We have:

$$\begin{aligned} \text{BB}[n] &= s[n] = \text{8 ksps recorded speech} \\ \text{NB}[n] &= A\cos(\omega_o n) = \text{interfering tone} \end{aligned} \tag{25}$$

We again let $\omega_0 = \pi/10$. In the project ZIP package a wave file is provided (`OSR_us_000_0030_8k.wav`), which can be imported into your Jupyter notebook/workspace using the command:

```
fs,s = ss.from_wav('OSR_us_000_0030_8k.wav')
```

Other import methods are also possible. The SNR is replaced by a similar measure, denoted signal-to-interference ratio (SIR), defined as

$$\text{SIR} = \frac{\text{var}\{s[n]\}}{(A^2/2)} \tag{26}$$

Modify your FIR ALE to accept the speech signal in place of the sinusoid and the sinusoid in place of the noise. Define the amplitude of the sinusoid to achieve a particular SIR, e.g.,

```
n = arange(0,N+1) # actually length N+1
x = cos(2*pi*1/20*n) # Here A = 1, Fo/Fs = 1/20
s = s[:N+1] # the input speech vector truncated to the length N+1
Ps = var(s) #estimate the AC power in s
x = s + sqrt(2*Ps*10**(-SIR/10))*x
```

a.) Rework the simulation code of Problem 1 to perform adaptive interference cancellation as described above. The function interface should be as follows:

```
def lms_ic(s,SIR,N,M,delta,mu):
    """
    Adaptive interference canceller using LMS and FIR
    n,x,x_hat,e,ao,F,Ao = lms_ic(s,SIR,N,M,delta,mu)

    *******LMS Interference Cancellation Simulation************
         s = Input speech signal
       SIR = Speech signal power to sinusoid interference level in dB
         N = Number of simulation samples
         M = FIR Filter length (order M-1)
     delta = Delay used to generate the reference signal
        mu = LMS step-size

         n = Index vector
         x = Noisy input
     x_hat = Filtered output
         e = Error sequence
        ao = Final value of weight vector
         F = Frequency response axis vector
        Ao = Frequency response of filter
    ***************************************
    Mark Wickert, November 2018
    """
```

b) Using N = 100,000 samples, find an appropriate converge parameter μ, to produce the narrowband (`x_hat`), and broadband (`e`) vectors with SIR = 0 dB. Satisfy yourself that the algorithm is working well by listening to the input speech plus interference vector, `x`,

along with `x_hat` and `e`, by converting the output array back to a wave file and using the `Audio` player in the Jupyter notebook, e.g., to save use:

```
ss.to_wav('e_speech_part_b.wav'.wav,8000,e)
```

`Audio` player in the Jupyter notebook, e.g., for the audio player use:

```
Audio('e_speech_part_b.wav')
```

Also plot the waveforms as a `subplot()` vertical stack of three waveforms, `x`, `x_hat`, and `e`. Finally, so I can verify your results qualitatively, export the three vectors as a `wave` file using the `ss.to_wave()` command. E-mail these files to me on or before the project due date in a ZIP package (six files total including the corresponding files of part (c)).

```
ss.to_wav('x_speech_part_b.wav',8000,x)
ss.to_wav('x_hat_speech_part_b.wav',8000,x_hat)
ss.to_wav('e_speech_part_b.wav',8000,e)
```

c) Repeat part (b) with SIR = -10 dB.

Adaptive Filter Project Hints: Fall 2024

1. The main focus of this project is to actually implement the adaptive filter system for what is known as *adaptive line enhancement* (ALE). On page 4 of the handout there are three steps to the algorithm. There are many ways to implement this. It is important to recognize that because this is a time-varying system you cannot simply process the signal + noise samples as one vector operation. At least one `for` loop will be required to step through the input signal samples one at a time.

   The `lfilter` function in Python is still very useful here. Assume that you initially place all of the input samples (signal + noise) in the array `x`. The signal `y` can be obtained easily as

   ```
   y = lfilter([0, 1],1,x)
   ```

   The `lfilter` function can also be used to filter just one sample at a time allowing you to change filter coefficients in between each sample. This accommodates step 1 of the LMS algorithm. The approach requires that you learn how to use the initial and final states array option of the filter function (recall Project #1), e.g., something like

```
def lms_ale(SNR,N,M,mu,sqwav=False):
    """
    lms_ale  lms ALE adaptation algorithm using an IIR filter.
            n,x,x_hat,e,ao,F,Ao = lms_ale(SNR,N,M,mu)

    *******LMS ALE Simulation************
       SNR = Sinusoid SNR in dB
         N = Number of simulation samples
         M = FIR Filter length (order M-1)
        mu = LMS step-size
      mode = 0 <=> sinusoid, 1 <=> squarewave

         n = Index vector
         x = Noisy input
     x_hat = Filtered output
         e = Error sequence
        ao = Final value of weight vector
         F = Frequency response axis vector
        Ao = Frequency response of filter in dB
    ***************************************
    Mark Wickert, November 2016
    """

    # Sinusoid SNR = (A^2/2)/noise_var
    n = arange(0,N+1) # length N+1
    if not(sqwav):
```

```
        x = 1*cos(2*pi*1/20*n) # Here A = 1, Fo/Fs = 1/20
        x += sqrt(1/2/(10**(SNR/10)))*random.randn(N+1)
    else: # Squarewave case
        #write code here
        pass
    # Normalize mu
    # write code
    # White Noise -> Delta = 1, so delay x by one sample
    y = signal.lfilter([0, 1],1,x)
    # Initialize output vector x_hat to zero
    x_hat = zeros_like(x)
    # Initialize error vector e to zero
    e = zeros_like(x)
    # Initialize weight vector to zero
    ao = zeros(M+1)
    # Initialize filter memory to zero
    zi = signal.lfiltic(ao,1,y=0)
    # ...
    for k,yk in enumerate(y):
       # Filter one sample at a time
       yfilt,zi = signal.lfilter(ao,1,array([yk]),zi=zi)
       x_hat[k] = yfilt[0]
       # Other LMS calculations follow
       # ...
    # Additional calculations in preparation for
    # plotting simulation results from the workspace
    # ...
    return n,x,x_hat,e,ao,F,Ao
```

On each pass through the main simulation loop the filter initial states load into `lfilter` and the final filter states are returned to the same vector for use on the next pass.

You may find it more convenient to just use vector arithmetic throughout, and abandon the use of `lfilter`. If this is the case, you will then need to keep track of the filter states on your own. An array can be set up to hold the past and present inputs to the FIR filter. As each new signal sample is processed you need to update this vector by removing the oldest value from one end, and adding the new input signal sample at the opposite end.

```
y_states = zeros_like(a0)
for k,yk in enumerate(y):
    y_states = hstack((array([yk]), y_states[0:-1]))
    x_hat[k] = dot(a0,y_states) #form inner product to get output x_hat
        ...
```

There are still algorithm design issues you need to figure out. In particular you need to do

step 3, which is where you update the filter coefficient vector before filtering the next signal + noise input sample.

2. Selection of μ is critical to getting good filter convergence. I have asked you to plot all of the waveforms which help you see if things are working correctly. The question you likely have is what do these waveforms look like when all is well. First some comments on setting μ.

   Equation (12) states that

   $$\mu_n < \frac{\mu}{(M+1)\sigma^2}$$

   This is a good way to set μ in your algorithm. Let μ be the value you enter as a parameter into your program. This should be somewhere around 0.001. The actual value used in your code will be μ_n, a much smaller number since M is large. The σ^2 given here is the total signal + noise power level. If you let $A = 1$ the signal power will just be $A^2/2 = 1/2$. As the SNR goes down the noise power contribution will increase, but at 10 dB the total power is dominated by the signal power, so really if you just left σ^2 out of the equation and just used μ and $M+1$ to get μ_n you would be pretty safe for the SNR values you will be using. You can also decrease μ as needed to get good stable convergence. If you are having problems getting the fixed μ to work you may want to try an adaptive μ, where you use equation (13) to adjust the power level used to normalize equation (12)

   When μ is good you will see at the end of 1000 sample a good clean sinewave at SNR = 10 dB. The mean square error (MSE) plot ($e^2[n]$ versus n) will start large and then get small, meaning that the filter has converged to a minimum value of MSE. The frequency response of the converged filter will have a nice clean defined passband centered on the sinusoid frequency, which here is at $\omega = \pi/10$ or in terms of normalized frequency $f = \omega/(2\pi) = 0.05$. When you plot the filter frequency response please plot just the magnitude response in dB versus normalized frequency, $f = \omega/(2\pi)$. You can easily make `freqz` do this by extracting the appropriate outputs and setting the sampling frequency on the input equal to one.

   At SNR = 0 dB the waveforms will not be as clean, but the output sinusoid will still be much cleaner than the composite signal + noise input.

3. For the squarewave investigation first realize that getting a unit amplitude squarewave is easy if you just use the numpy `sign()` function to *hard limit* a sinusoid. Notice also that the SNR definition is slightly different because the power is a squarewave is just A^2.

   With a squarewave input the adaptive filter will form passbands at the significant harmonics of the squarewave. Since the harmonics of a squarewave falloff in amplitude as $1/n$, where n is the harmonic number, the passbands do not have equal gain. Depending upon how μ is

set the recovered or output signal will not be a real square or sharp edge signal. It will still look more like a squarewave than a sinusoid.

4. When implementing the IIR ALE you simply implement the 3 steps found on page 5. You also need to form an update of $\alpha[n]$. You can again use `lfilter` with `zi` to maintain state after you filter each sample, or you can explicitly filter $y[n]$ to get $\hat{x}[n]$ as in the alternate implementation of the FIR ALE. The filter states can be maintained in variables you declare and update as each new signal sample is processed. A single `for` loop which steps `n` from 0 to 1000 should again be your basic code structure.

5. You initialize the algorithm with $w = 0$ and then it will begin to move towards a value corresponding to the sinusoid center frequency as n increases, if all is well. A good trouble shooting technique is to monitor the progress of $w[n]$ over the entire simulation, e.g., start with

   ```
   w = zeros(N+1)
   ```

   Then write new values into this vector each time through your main loop.

Project2 (Final Project) 2024 Point Assignments

Problem	Points per part
1. FIR ALE sine	10 (program) (a) [10 dB]: 4 (MSE) 4 ($\hat{x}/x$), 4 ($\lvert H(e^{j\omega})\rvert_{\text{dB}}$) (b) [0 dB]: 4 (MSE) 4 ($\hat{x}/x$), 4 ($\lvert H(e^{j\omega})\rvert_{\text{dB}}$)
2. FIR ALE sq	4 (program mods) (a) [10 dB]: 4 (MSE) 4 ($\hat{x}/x$), 4 ($\lvert H(e^{j\omega})\rvert_{\text{dB}}$) (b) [0 dB]: 4 (MSE) 4 ($\hat{x}/x$), 4 ($\lvert H(e^{j\omega})\rvert_{\text{dB}}$)
3. **Bonus** IIR ALE	10 (program) (a) [10 dB]: 4 (MSE) 4 ($\hat{x}/x$), 2, (w/γ), 4 ($\lvert H(e^{j\omega})\rvert_{\text{dB}}$)
4. FIR Noise canceler	4 (program mods) (a) 5 (0 dB wavefiles) $x[n]$, $\hat{x}[n]$, $e[n]$ (b) 5 (-10 dB wavefiles) $x[n]$, $\hat{x}[n]$, $e[n]$
Total	76 + 24 Bonus to be combined with the Project 1 grade