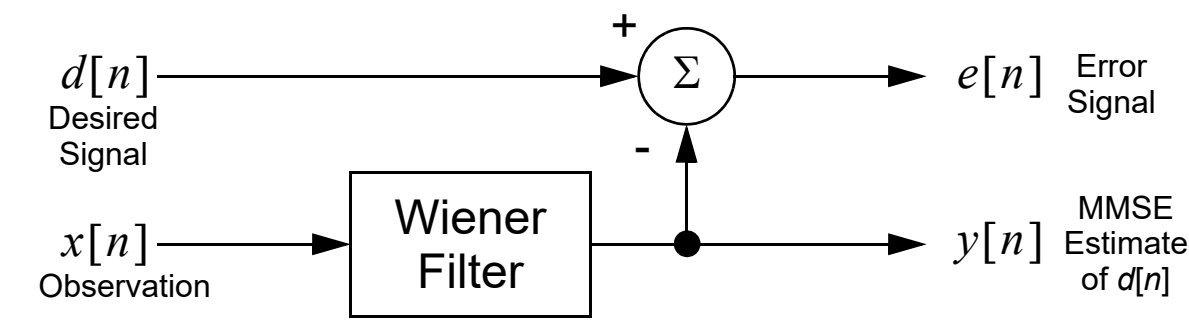# Adaptive Filters

Introduction

The term adaptive filter implies changing the characteristic of a filter in some automated fashion to obtain the *best* possible signal quality in spite of changing signal/system conditions. Adaptive filters are usually associated with the broader topic of statistical signal processing. The operation of signal filtering by definition implies extracting something desired from a signal containing both desired and undesired components. With linear FIR and IIR filters the filter output is obtained as a linear function of the *observation* (signal applied) to the input. An optimum linear filter in the minimum mean square sense can be designed to extract a signal from noise by minimizing the error signal formed by subtracting the filtered signal from the *desired signal*. For noisy signals with time varying statistics, this minimization process is often done using an *adaptive filter*.

For statistically stationary inputs this solution is known as a *Wiener* filter.[1]

1. Simon Haykin, *Adaptive Filter Theory,* fourth edition, Prentice Hall, 2002.

Wiener Filter

- An *M* tap discrete-time Wiener filter is of the form

$$y[n] = \sum_{m=0}^{M-1} w_m x[n-m] \qquad (8.1)$$

where the w_m are referred to as the filter weights

- Note: (8.1) tells us that the Wiener filter is just an *M*-tap FIR filter

- The quality of the filtered or estimated signal $y[n]$ is determined from the error sequence $e[n] = d[n] - y[n]$

- The weights w_m, $m = 0, 1, …, M-1$ are chosen such that

$$E\{e^2[n]\} = E\{(d[n] - y[n])^2\} \qquad (8.2)$$

is minimized, that is we obtain the minimum mean-squared error (MMSE)

- The optimal weights are found by setting

$$\frac{\partial}{\partial w_m} E\{e^2[n]\} = 0, m = 0, 1, …, M-1 \qquad (8.3)$$

- From the orthogonality principle[1] we choose the weights such that the error $e[n]$ is orthogonal to the observations (data), i.e.,

---

1. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, third edition, McGraw-Hill, 1991.

$$E\{x[n-k](d[n]-y[n])\} = 0, k = 0, 1, \ldots, M-1 \qquad (8.4)$$

- This results in a filter that is optimum in the sense of minimum mean-square error

- The resulting system of equations

$$\sum_{m=0}^{M-1} w_m E\{x[n-k]x[n-m]\} = E\{x[n-k](d[n])\} \qquad (8.5)$$

or

$$\sum_{m=0}^{M-1} w_m \phi_{xx}[m-k] = \phi_{xd}[-k] \qquad (8.6)$$

for $k = 0, 1, \ldots, M-1$ are known as the Wiener-Hopf or normal equations

- Note: $\phi_{xx}[k]$ is the autocorrelation function of $x[n]$ and $\phi_{xd}[k]$ is the cross-correlation function between $x[n]$ and $d[n]$

- In matrix form we can write

$$\mathbf{R}_{xx}\mathbf{w}_o = \mathbf{p}_{xd} \qquad (8.7)$$

where $\mathbf{R}_{xx}$ is the $M \times M$ correlation matrix associated with $x[n]$

$$\mathbf{R}_{xx} = \begin{bmatrix} \phi_{xx}[0] & \cdots & \phi_{xx}[M-1] \\ \vdots & \ddots & \vdots \\ \phi_{xx}[-M+1] & \cdots & \phi_{xx}[0] \end{bmatrix} \tag{8.8}$$

$\mathbf{w}_o$ is the optimum weight vector given by

$$\mathbf{w}_o = \begin{bmatrix} w_{o0} & w_{o1} & \cdots & w_{oM-1} \end{bmatrix}^T \tag{8.9}$$

and $\mathbf{p}_{xd}$ is the cross-correlation vector given by

$$\mathbf{p}_{xd} = \begin{bmatrix} \phi_{xd}[0] & \phi_{xd}[-1] & \cdots & \phi_{xd}[1-M] \end{bmatrix}^T \tag{8.10}$$

- The optimal weight vector is given by

$$\mathbf{w}_o = \mathbf{R}_{xx}^{-1}\mathbf{p}_{xd} \tag{8.11}$$

- As a matter of practice (8.11) can be solved using sample statistics, that is we replace the true statistical auto- and cross-correlation functions with time averages of the form

$$\phi_{xx}[k] \cong \frac{1}{N}\sum_{n=0}^{N-1} x[n+k]x[n] \tag{8.12}$$

$$\phi_{xd}[k] \cong \frac{1}{N}\sum_{n=0}^{N-1} x[n+k]d[n] \tag{8.13}$$

where N is the sample block size

Adaptive Wiener Filter

- 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*

- With respect to the weight vector $\mathbf{w}$, the error $e[n]$ can be viewed as an M dimensional error surface, that due to the squared error criterion, is convex cup (a bowl shape)

Optimum at [1,1]
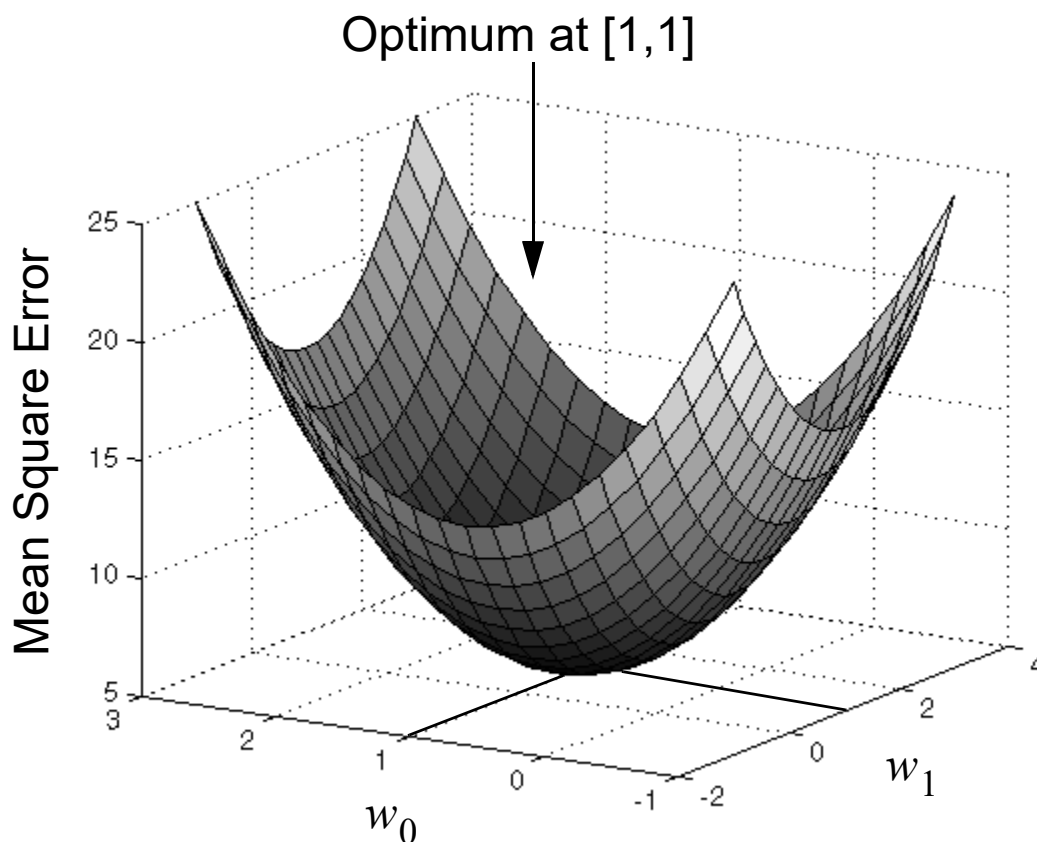
Error Surface for M = 2

- The filter decorrelates the output error $e[n]$ so that signals in common to both $d[n]$ and $x[n]$ in a correlation sense are canceled

---

- A block diagram of this adaptive Wiener (FIR) filter is shown below

Adaptive Wiener Filter
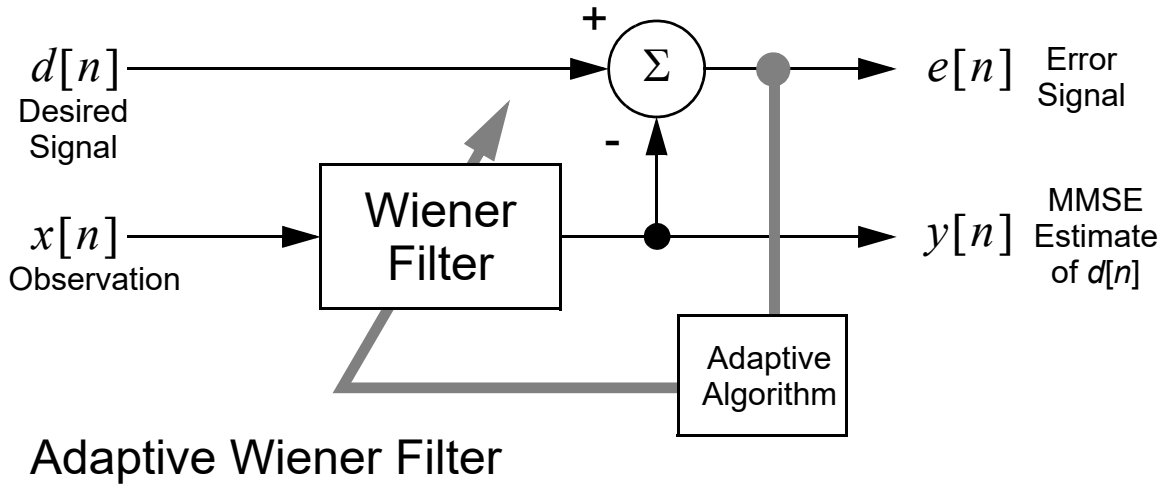
Least-Mean-Square Adaptation

- Ideally the optimal weight solution can be obtained by applying the steepest descent method to the error surface, but since the true gradient cannot be determined, we use a *stochastic gradient*, which is simply the instantaneous estimate of $\mathbf{R}_{xx}$ and $\mathbf{p}_{xd}$ from the available data, e.g.,

$$\hat{\mathbf{R}}_{xx} = \mathbf{x}[n]\mathbf{x}^T[n] \tag{8.14}$$

$$\hat{\mathbf{p}}_{xd} = \mathbf{x}[n]d[n] \tag{8.15}$$

where

$$\mathbf{x}[n] = \left[x[n]\ x[n-1]\ \dots\ x[n-M+1]\right]^T \tag{8.16}$$

- A practical implementation involves estimating the gradient from the available data using the *least-mean-square* (LMS) *algorithm*

- The steps to the LMS algorithm, for each new sample at time n, are:

  - Filter $x[n]$ to produce:

$$y[n] = \sum_{m=0}^{M-1} \hat{w}_m[n]x[n-m] = \hat{\mathbf{w}}^T[n]\mathbf{x}[n] \qquad (8.17)$$

  - Form the estimation error:

$$e[n] = d[n] - y[n] \qquad (8.18)$$

  - Update the weight vector using step-size parameter μ:

$$\hat{\mathbf{w}}[n+1] = \hat{\mathbf{w}}[n] + \mu\mathbf{x}[n]e[n] \qquad (8.19)$$

- For algorithm stability, the step-size μ must be chosen such that

$$0 < \mu < \frac{2}{\text{tap-input power}} \qquad (8.20)$$

  where

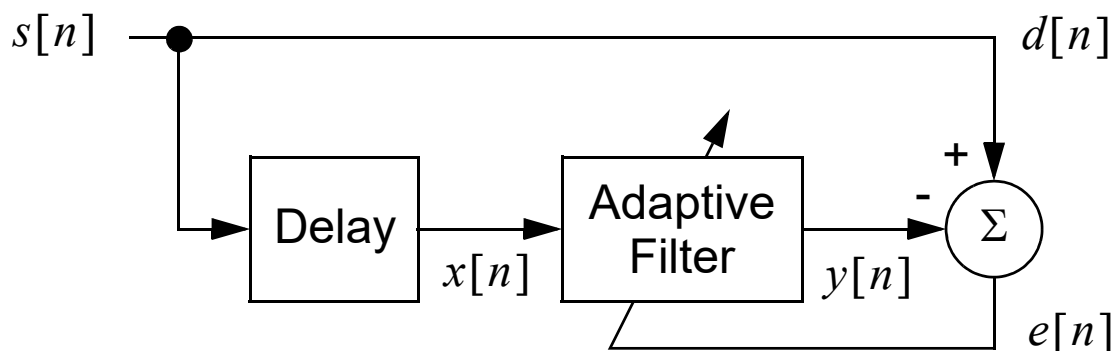$$\text{tap-input power} = \sum_{k=0}^{M-1} E\{|x[n-k]|^2\} \qquad (8.21)$$

- In theory, (8.20) is equivalent to saying
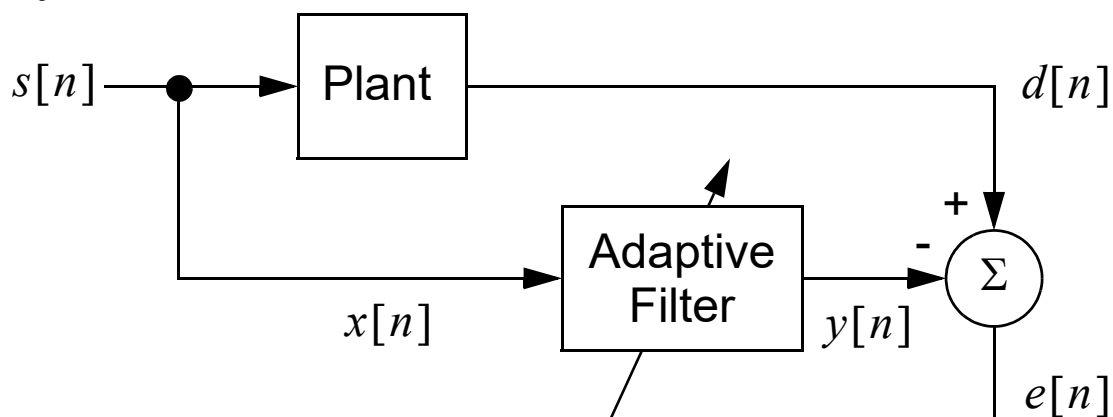
$$0 < \mu < \frac{2}{\lambda_{max}} \qquad (8.22)$$

  where λ_{max} is the maximum eigenvalue of $\mathbf{R}_{xx}$

Adaptive Filter Variations[1]

- **Prediction**

$s[n]$ $d[n]$

Delay → $x[n]$ → Adaptive Filter → $y[n]$

$+$ $-$ Σ

$e[n]$

- **System Identification**

$s[n]$ → Plant → $d[n]$

$x[n]$ → Adaptive Filter → $y[n]$

$+$ $-$ Σ

$e[n]$

- **Equalization**

$s[n]$ → Delay → Training Pattern $d[n]$

Plant/Channel → $+$ Σ $+$ → $x[n]$ → Adaptive Filter → $y[n]$

Noise

$+$ $-$ Σ

$e[n]$
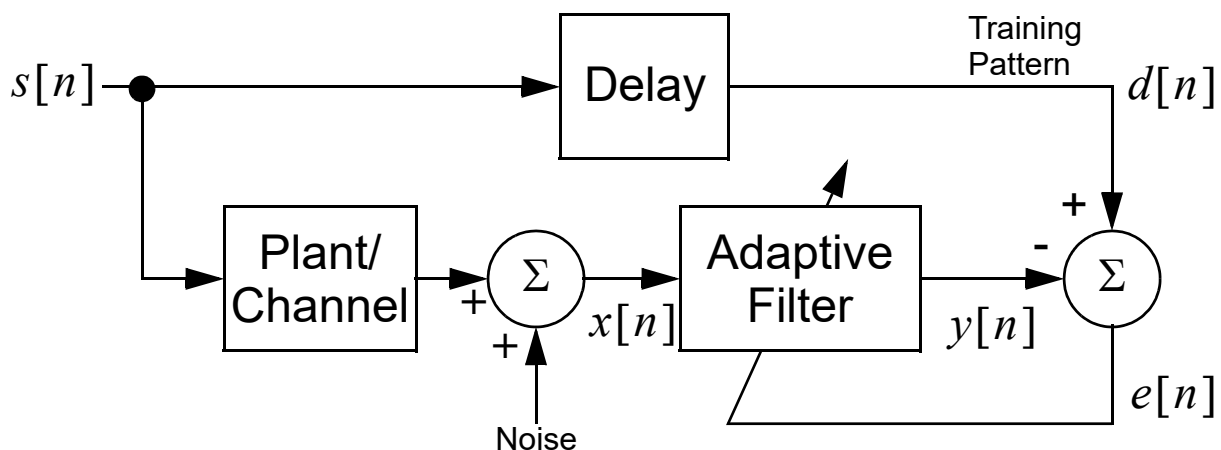
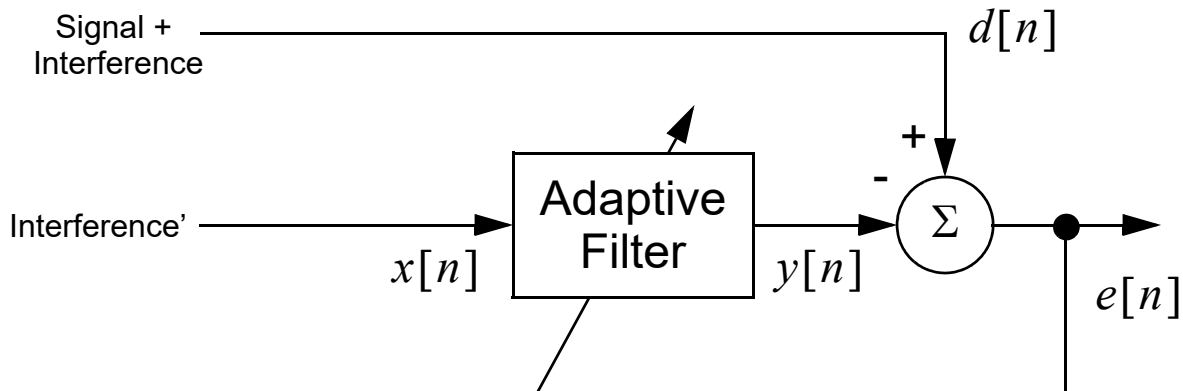
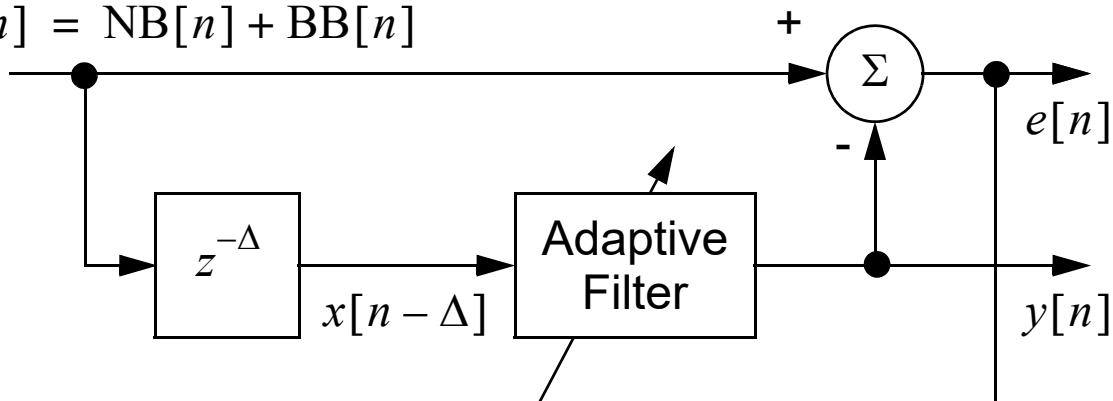1. B. Widrow and S. Stearns, *Adaptive Signal Processing*, Prentice Hall, New Jersey, 1985.

- **Interference Canceling**

Adaptive Line Enhancement

- A relative of the interference canceling scheme shown above, is the *adaptive line enhancer* (ALE)

- Here we assume we have a narrowband signal (say a sinusoid) buried in broadband additive noise

- The filter adapts in such a way that a narrow passband forms around the sinusoid frequency, thereby suppressing much of the noise and improving the signal-to-noise ratio (SNR) in $y[n]$

Example: Python ALE Simulation

- A simple Python simulation is constructed using a single sinusoid at normalized frequency $f_o = 1/20$ plus additive white Gaussian noise

$$x[n] = A\cos[2\pi f_o n] + w[n] \tag{8.23}$$

- The SNR is defined as

$$\text{SNR} = \frac{A^2}{2\sigma_w^2} \tag{8.24}$$

```python
def lms_ale(SNR,N,M,mu,sqwav=False,Nfft=1024):
    """
    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 2014
    """

    # 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)))*randn(N+1)
    else: # Squarewave case
        x = 1*sign(cos(2*pi*1/20*n)); # Here A = 1, Fo/Fs = 1/20
```

```python
        x += sqrt(1/1/(10**(SNR/10)))*randn(N+1)

    # Normalize mu
    mu /= M + 1
    # 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)
    # Initialize a vector for holding ym of length M+1
    ym = zeros_like(ao)
    for k,yk in enumerate(y):
        # Filter one sample at a time
        x_hat[k],zi = signal.lfilter(ao,1,[yk],zi=zi)
        # Form the error sequence
        e[k] = x[k] - x_hat[k]
        # Update the weight vector
        ao = ao + 2*mu*e[k]*ym
        # Update vector used for correlation with e[k]
        ym = hstack((array(yk), ym[0:-1]))
    # Create filter frequency response
    F = arange(0,0.5,1/Nfft)
    w,Ao = signal.freqz(ao,1,2*pi*F)
    Ao = 20*log10(abs(Ao))
    return n,x,x_hat,e,ao,F,Ao
```

- A simulation is run in a Jupyter notebook using 1000 samples, SNR $= 10$ dB, $M = 64$, and $\mu = 0.01/64$

```python
Ntaps = 128
n,x,x_hat,e,ao,F,Ao = lms_ale(10,1000,Ntaps,0.01,sqwav=False)
```
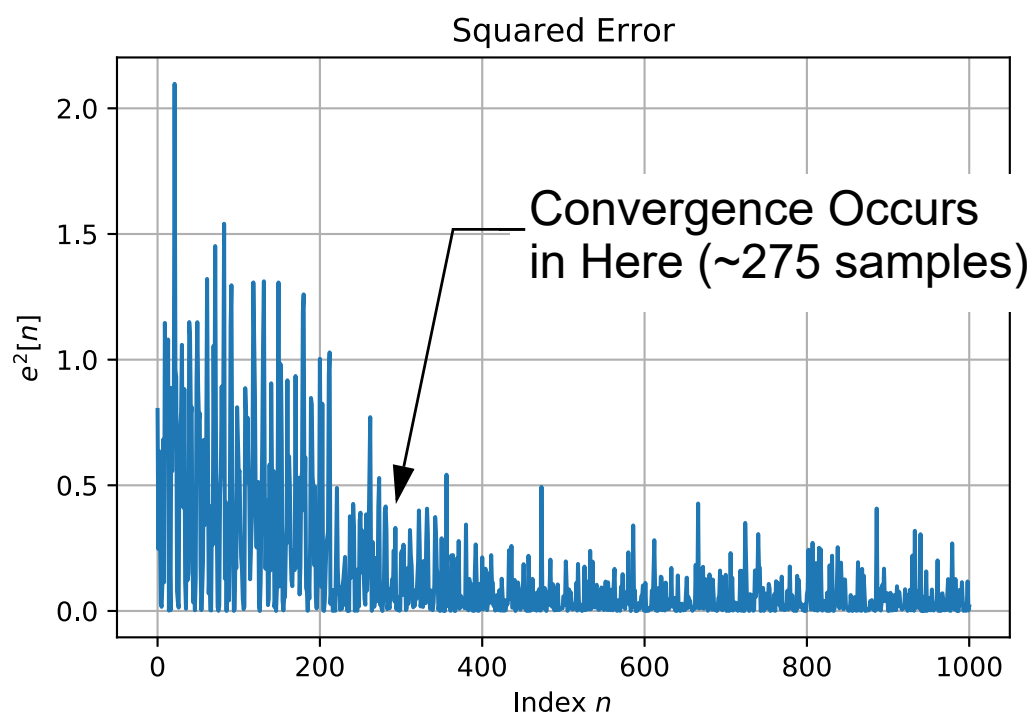
- Plot the squared error

```python
plot(n,e**2)
ylabel(r'$e^2[n]$')
xlabel(r'Index $n$')
title(r'Squared Error')
grid();
savefig('ALE_mse.pdf')
```

Squared Error

- Plot the ALE filter noisy input and *clean* output

```
subplot(211)
plot(n[600:],x[600:])
ylabel(r'$x[n]$')
xlabel(r'Index $n$')
title(r'Noisy Input')
grid();
subplot(212)
plot(n[600:],x_hat[600:])
ylabel(r'$\hat{x}[n]$')
xlabel(r'Index $n$')
title(r'Filtered Output')
grid();
tight_layout()
savefig('ALE_io.pdf')
```
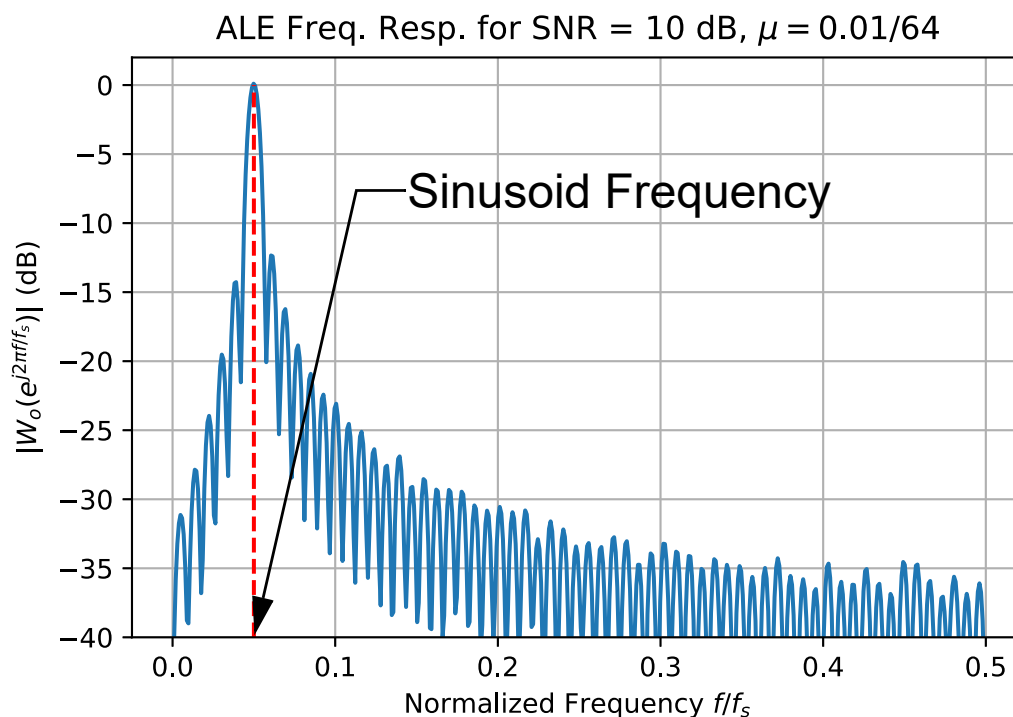
Noisy Input

Filtered Output
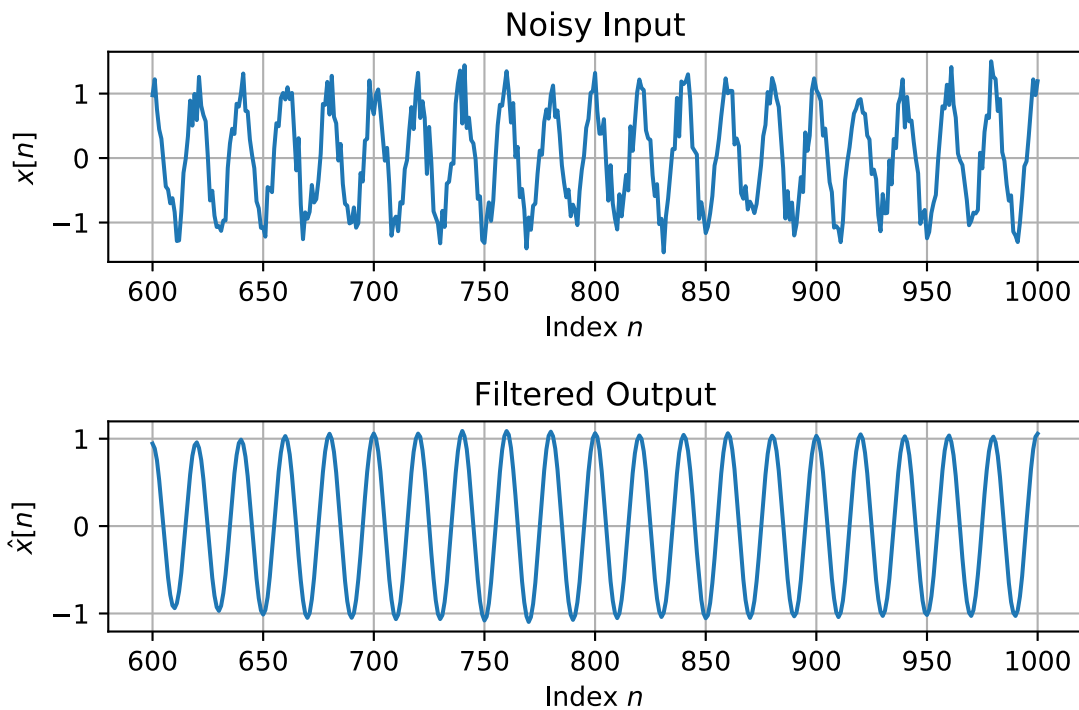
- Plot the frequency response of the approximately optimum filter, $\mathbf{w}_o$, at 1000 samples in to the simulation

```
plot(F,Ao)
ylim([-40,2])
plot([0.05, 0.05],[-40,0],'r--')
xlabel(r'Normalized Frequency $f/f_s$')
ylabel(r'$|W_o(e^{j2\pi f/f_s})|$ (dB)')
title(r'ALE Freq. Resp. for SNR = 10 dB, $\mu = 0.01/64$')
grid();
savefig('ALE_fresp.pdf')
```

ALE Freq. Resp. for SNR = 10 dB, $\mu = 0.01/64$

- A C version of the above Python code would be very similar except all of the vector operations would have to be replaced by `for` loops

- With CMSIS-DSP available, we will choose this route in an upcoming example

Example: Adaptive Interference Canceling

- Adaptive interference canceling is implemented in the scikit-dsp-comm module sigsys.py

```python
def lms_ic(r,M,mu,delta=1):
    """
    Least mean square (LMS) interference canceller adaptive filter.

    A complete LMS adaptive filter simulation function for the case of
    interference cancellation. Used in the digital filtering case study.

    Parameters
    ----------
```

```
M : FIR Filter length (order M-1)
delta : Delay used to generate the reference signal
mu : LMS step-size
delta : decorrelation delay between input and FIR filter input

Returns
-------
n : ndarray Index vector
r : ndarray noisy (with interference) input signal
r_hat : ndarray filtered output (NB_hat[n])
e : ndarray error sequence (WB_hat[n])
ao : ndarray final value of weight vector
F : ndarray frequency response axis vector
Ao : ndarray frequency response of filter

Examples
----------
>>> # import a speech signal
>>> fs,s = from_wav('OSR_us_000_0030_8k.wav')
>>> # add interference at 1kHz and 1.5 kHz and
>>> # truncate to 5 seconds
>>> r = soi_snoi_gen(s,10,5*8000,[1000, 1500])
>>> # simulate with a 64 tap FIR and mu = 0.005
>>> n,r,r_hat,e,ao,F,Ao = lms_ic(r,64,0.005)
"""

N = len(r)-1;
# Form the reference signal y via delay delta
y = signal.lfilter(np.hstack((np.zeros(delta), np.array([1]))),1,r)
# Initialize output vector x_hat to zero
r_hat = np.zeros(N+1)
# Initialize error vector e to zero
e = np.zeros(N+1)
# Initialize weight vector to zero
ao = np.zeros(M+1)
# Initialize filter memory to zero
z = np.zeros(M)
# Initialize a vector for holding ym of length M+1
ym = np.zeros(M+1)
for k in range(N+1):
    # Filter one sample at a time
    r_hat[k],z = signal.lfilter(ao,np.array([1]),np.array([y[k]]),zi=z)
    # Form the error sequence
    e[k] = r[k] - r_hat[k]
    # Update the weight vector
    ao = ao + 2*mu*e[k]*ym
    # Update vector used for correlation with e(k)
    ym = np.hstack((np.array([y[k]]), ym[:-1]))
```

```python
# Create filter frequency response
F, Ao = signal.freqz(ao,1,1024)
F/= (2*np.pi)
Ao = 20*np.log10(abs(Ao))
return np.arange(0,N+1), r, r_hat, e, ao, F, Ao
```

- Here we consider testing with an $f_s = 8\,\text{kHz}$ speech file with sinusoidal interference, using a 64 tap filter

- Tone at 1 kHz and 1.5 kHz are added to the speech

- The overall signal-to-interference ratio (SIR) is set to 10dB

```python
import sk_dsp_comm.sigsys as ss

# import a speech signal
fs,s = ss.from_wav('OSR_us_000_0030_8k.wav')
# add interference at 1kHz
# truncate to 5 seconds and include one or more tone frequencies
r_in = ss.soi_snoi_gen(s[15000:],10,5*8000,[1000, 1500])
# simulate with a 64 tap FIR and mu = 0.005
n,r,r_hat,e,ao,F,Ao = ss.lms_ic(r_in,64,0.01)
```
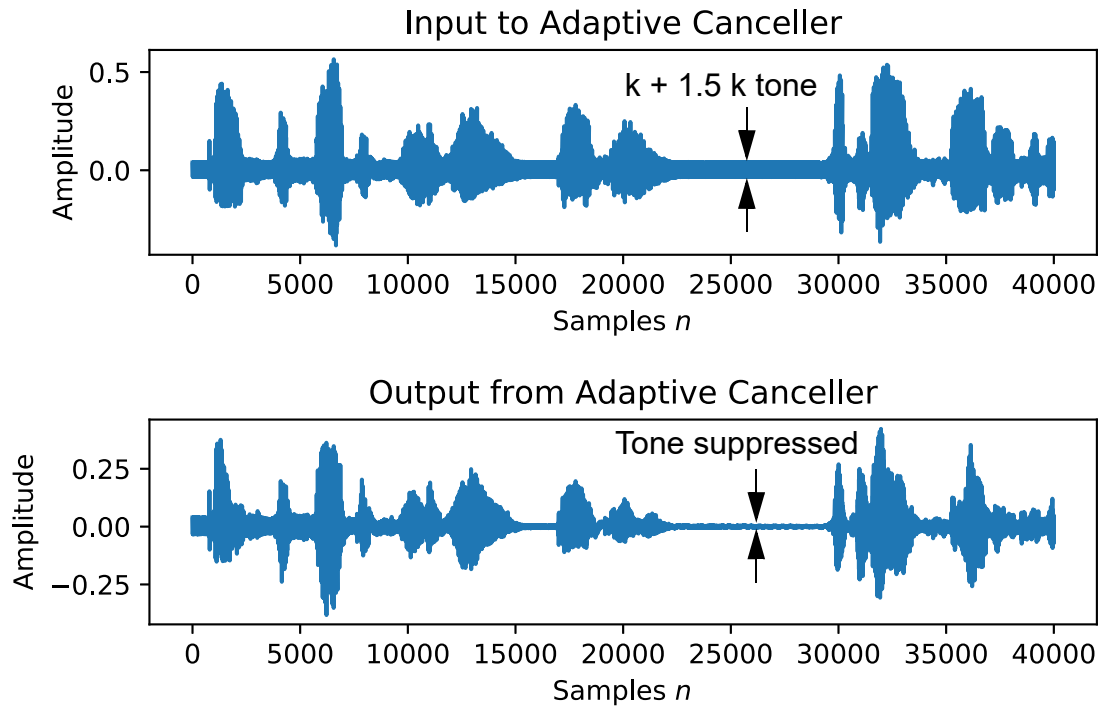
- The before and after waveforms are plotted below:

```python
subplot(211)
plot(r)
title(r'Input to Adaptive Canceller')
ylabel(r'Amplitude')
xlabel(r'Samples $n$')
subplot(212)
#plot(r_hat)
plot(e)
title(r'Output from Adaptive Canceller')
ylabel(r'Amplitude')
xlabel(r'Samples $n$')
tight_layout()
savefig('IC_io.pdf')
```

Input to Adaptive Canceller

Output from Adaptive Canceller

- Listening make the results more convincing

```
# The starting point
ss.to_wav('OSR_us_30_8k_in.wav',8000,r)
Audio('OSR_us_30_8k_in.wav')
```

▶ 0:00 ———●————— -0:04

```
#ss.to_wav('OSR_us_30_8k_out.wav',8000,r_hat)
ss.to_wav('OSR_us_30_8k_out.wav',8000,e)
Audio('OSR_us_30_8k_out.wav')
```

▶ 0:05 —————————●— -0:00

Playback using the Jupyter notebook Audio control

Cortex-M4 Implementation of ALE & IC

- The CMSIS-DSP library contains adaptive LMS filters

Least Mean Square (LMS) Filters

Filtering Functions

Functions

void **arm_lms_f32** (const **arm_lms_instance_f32** *S, **float32_t** *pSrc, **float32_t** *pRef, **float32_t** *pOut, **float32_t** *pErr, uint32_t **blockSize**)
Processing function for floating-point LMS filter.

void **arm_lms_init_f32** (**arm_lms_instance_f32** *S, uint16_t numTaps, **float32_t** *pCoeffs, **float32_t** *pState, **float32_t** mu, uint32_t **blockSize**)
Initialization function for floating-point LMS filter.

void **arm_lms_q15** (const **arm_lms_instance_q15** *S, **q15_t** *pSrc, **q15_t** *pRef, **q15_t** *pOut, **q15_t** *pErr, uint32_t **blockSize**)
Processing function for Q15 LMS filter.

void **arm_lms_q31** (const **arm_lms_instance_q31** *S, **q31_t** *pSrc, **q31_t** *pRef, **q31_t** *pOut, **q31_t** *pErr, uint32_t **blockSize**)
Processing function for Q31 LMS filter.

Description

LMS filters are a class of adaptive filters that are able to "learn" an unknown transfer functions. LMS filters use a gradient descent method in which the filter coefficients are updated based on the instantaneous error signal. Adaptive filters are often used in communication systems, equalizers, and noise removal. The CMSIS DSP Library contains LMS filter functions that operate on Q15, Q31, and floating-point data types. The library also contains normalized LMS filters in which the filter coefficient adaptation is indepedent of the level of the input signal.

An LMS filter consists of two components as shown below. The first component is a standard transversal or FIR filter. The second component is a coefficient update mechanism. The LMS filter has two input signals. The "input" feeds the FIR filter while the "reference input" corresponds to the desired output of the FIR filter. That is, the FIR filter coefficients are updated so that the output of the FIR filter matches the reference input. The filter coefficient update mechanism is based on the difference between the FIR filter output and the reference input. This "error signal" tends towards zero as the filter adapts. The LMS processing functions accept the input and reference input signals and generate the filter output and error signal.
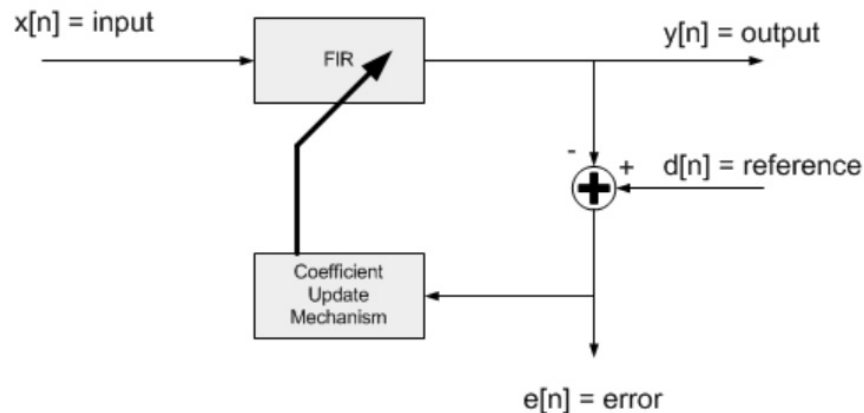
arm_lms_instance_f32 Struct Reference

Instance structure for the floating-point LMS filter.

Data Fields

uint16_t	**numTaps**
float32_t *	**pState**
float32_t *	**pCoeffs**
float32_t	**mu**

Later change this field using the GUI slider control

Internal structure of the Least Mean Square filter

The functions operate on blocks of data and each call to the function processes `blockSize` samples through the filter. `pSrc` points to input signal, `pRef` points to reference signal, `pOut` points to output signal and `pErr` points to error signal. All arrays contain `blockSize` values.

The functions operate on a block-by-block basis. Internally, the filter coefficients `b[n]` are updated on a sample-by-sample basis. The convergence of the LMS filter is slower compared to the normalized LMS algorithm.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter and coefficient and state arrays cannot be shared among instances. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, mu, postShift (not for f32), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 3 different data type filter instance structures

```
arm_lms_instance_f32 S = {numTaps, pState, pCoeffs, mu};
arm_lms_instance_q31 S = {numTaps, pState, pCoeffs, mu, postShift};
arm_lms_instance_q15 S = {numTaps, pState, pCoeffs, mu, postShift};
```

where `numTaps` is the number of filter coefficients in the filter; `pState` is the address of the state buffer; `pCoeffs` is the address of the coefficient buffer; `mu` is the step size parameter; and `postShift` is the shift applied to coefficients.

- The next steps are to develop some examples using custom code and using the CMSIS-DSP library for comparison

Adaptive Line Enhancer and Interference Canceler on the FM4

- A combination adaptive line enhancer and interference cancellor example is available inside `Lab_6_UART_Adaptive.zip`

- The source file `FM4_Adaptive_intr.c` uses the GUI slider control to allow a single application file to implement different adaptive filter implementations:

  - Interference canceling
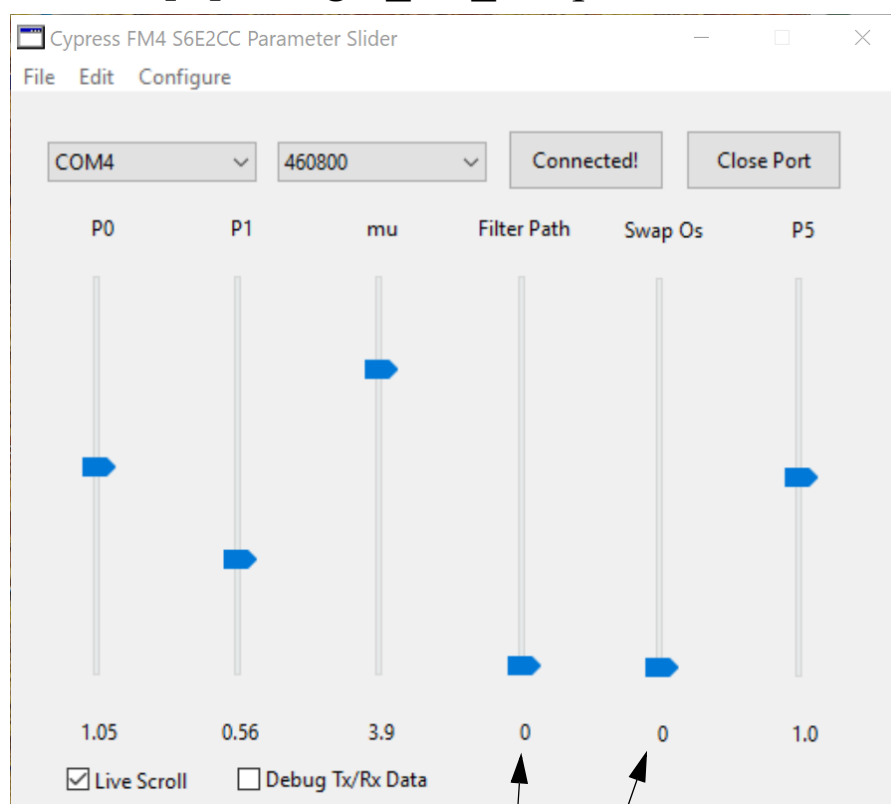
  - Adaptive line enhancement (ALE)
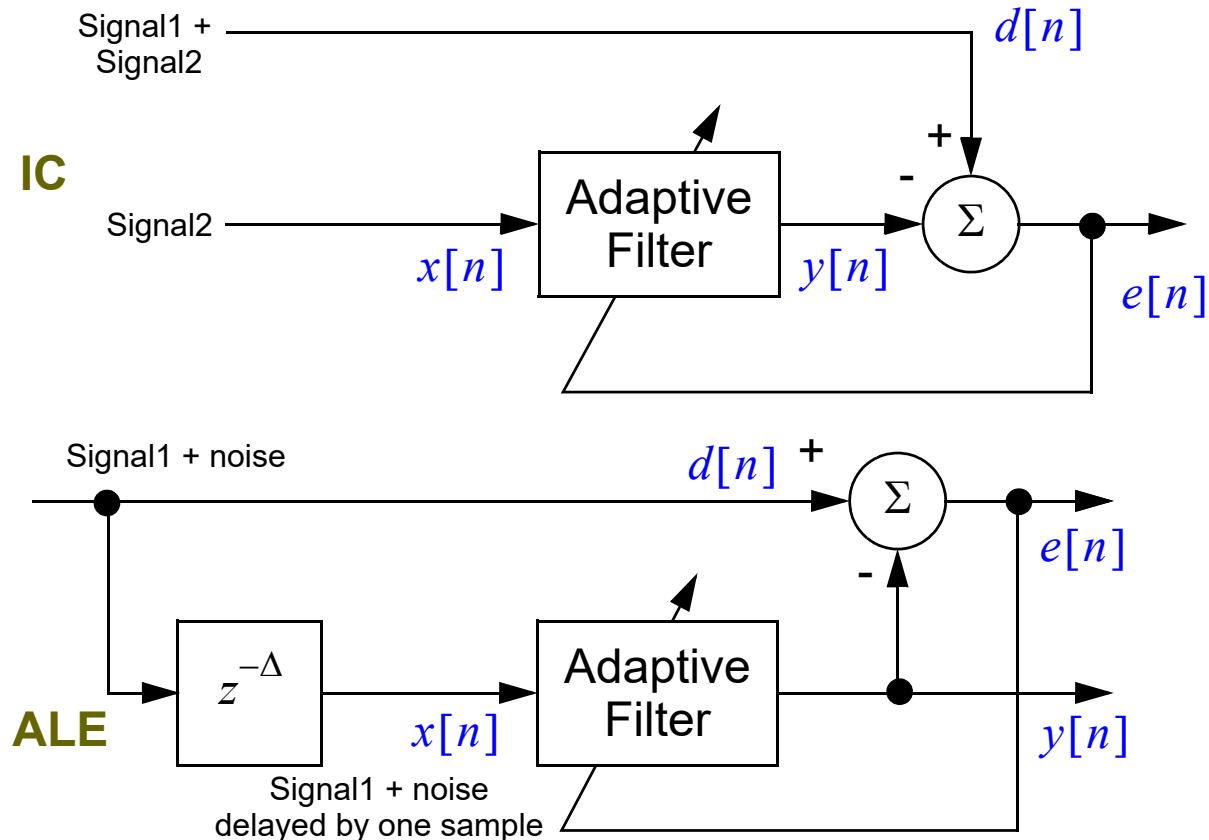
- The CMSIS-DSP function `arm_lms_f32()` is used as the adaptive filter kernel in both cases:

$$d[n] \Leftarrow \begin{cases} \text{left\_in\_sample} \cdot \text{P\_vals}[0] \\ \quad + \text{right\_in\_sample} \cdot \text{P\_vals}[1], \\ \text{left\_in\_sample} \cdot \text{P\_vals}[0] \\ \quad + \text{noise} \cdot \text{P\_vals}[1], \end{cases} \tag{8.25}$$

$$x[n] \Leftarrow \begin{cases} \text{left\_in\_sample}, & \text{P\_vals}[4] = 0 \\ d[n-1], & \text{P\_vals}[4] = 1 \end{cases} \tag{8.26}$$

$$y[n] \Rightarrow \text{left\_output\_sample} \tag{8.27}$$

$$e[n] \Rightarrow \text{right\_out\_sample} \tag{8.28}$$

Change app with Filter Path = P_vals[3]
Swap L/R outputs with = P_vals[4]

- For the interference canceling scenario the input $x[n]$ is a version of just interference portion of $d[n]$ which is composed of both the desired signal plus the interference

- For the adaptive line enhancer the input $x[n]$ is a copy of $d[n]$ delayed by one sample

- Delaying $d[n]$ by one sample uncorrelates the broadband component of $d[n]$ from itself, meaning that the adaptive filter will be ignoring this input

- The outputs for both algorithm need some interpretation too:

  – With the interference canceler the error signal $e[n]$ should contain the desired signal with the interference removed

  – For the ALE the broadband component, e.g., noise or speech, is returned on $e[n]$ and the narrowband component is recovered on $y[n]$

- Main module code listing:

```
// fm4_adaptive_intr_GUI.c

#include "fm4_wm8731_init.h"
#include "FM4_slider_interface.h"

int32_t rand_int32(void); // prototype for random number generator

// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

//CMSIS-DSP adaptive filter structure
arm_lms_instance_f32 LMS1;
//arm_lms_norm_instance_f32 LMS1;
uint16_t Ntaps = 100;
float32_t a_coeff[100];
float32_t states[100];
float32_t mu = 1e-13;

float32_t x_inter, x_noise, x_speech;
```

```c
float32_t y, y_hat, error;
float32_t y_del = 0.0f;

void PRGCRC_I2S_IRQHandler(void)
{
  union WM8731_data sample;
    //int16_t xL, xR;

  gpio_set(DIAGNOSTIC_PIN,HIGH);
    // Get L/R codec sample
  sample.uint32bit = i2s_rx();

    // Map L & R inputs to adaptive filter variable names
    x_noise = (float32_t)(((short)rand_int32())>>2);
    x_inter = (float32_t)sample.uint16bit[LEFT];
    x_speech = (float32_t)sample.uint16bit[RIGHT];
    if (FM4_GUI.P_vals[3] < 1)
    {
        // Used for interference canceller
        y = FM4_GUI.P_vals[0]*x_inter + FM4_GUI.P_vals[1]*x_speech;
    }
    else
    {
        // Used for ALE (note noise is generated internally)
        y = FM4_GUI.P_vals[0]*x_inter + FM4_GUI.P_vals[1]*x_noise;
    }
    if (FM4_GUI.P_vals[3] < 1) // For interference cancelling
    {
        //              input/src  ref  output  error  blk size
        arm_lms_f32(&LMS1, &x_inter, &y, &y_hat, &error, 1);
    }
    else // ALE where input is tone + noise
    {
        //              input/src  ref  output  error  blk size
        arm_lms_f32(&LMS1, &y_del, &y, &y_hat, &error, 1);
    }
    //arm_lms_norm_f32(&LMS1, &y_del, &y, &y_hat, &error, 1);
    y_del = y; // one sample delayed version of the reference input

    // Return L/R samples to codec via C union
    if (FM4_GUI.P_vals[4] < 1) // Normal channel outputs
    {
        sample.uint16bit[LEFT] = (int16_t) y_hat;
        sample.uint16bit[RIGHT] = (int16_t) error;
    }
    else // Swap channel outputs
    {
        sample.uint16bit[LEFT] = (int16_t) error;
```

```c
        sample.uint16bit[RIGHT] = (int16_t) y_hat;
    }
  i2s_tx(sample.uint32bit);

  NVIC_ClearPendingIRQ(PRGCRC_I2S_IRQn);

    gpio_set(DIAGNOSTIC_PIN,LOW);
}

int main(void)
{

    // Initialize the slider interface by setting the baud rate (460800 or
921600)
    // and initial float values for each of the 6 slider parameters

    init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0);

    // Send a string to the PC terminal
    write_uart0("Hello FM4 World!\r\n");

    // Initialize LMS
    arm_lms_init_f32(&LMS1,Ntaps,a_coeff,states,mu,1);
    //arm_lms_norm_init_f32(&LMS1,Ntaps,a_coeff,states,mu,1);

// Some #define options for initializing the audio codec interface:
    // FS_8000_HZ, FS_16000_HZ, FS_24000_HZ, FS_32000_HZ, FS_48000_HZ,
FS_96000_HZ
    // IO_METHOD_INTR, IO_METHOD_DMA
    // WM8731_MIC_IN, WM8731_MIC_IN_BOOST, WM8731_LINE_IN
  fm4_wm8731_init (FS_48000_HZ,                    // Sampling rate (sps)
                   WM8731_LINE_IN,           // Audio input port
                   IO_METHOD_INTR,           // Audio samples handler
                   WM8731_HP_OUT_GAIN_0_DB,   // Output headphone jack Gain
(dB)
                   WM8731_LINE_IN_GAIN_0_DB); // Line-in input gain (dB)
  while(1){
        // Update slider parameters
        update_slider_parameters(&FM4_GUI);
        // Update LMS mu of the LMS1 struct if slider parameter changes
        if(FM4_GUI.P_idx == 2)
        {
            mu = FM4_GUI.P_vals[2]*1e-13f;
            LMS1.mu = mu;
        }
    }
}
```
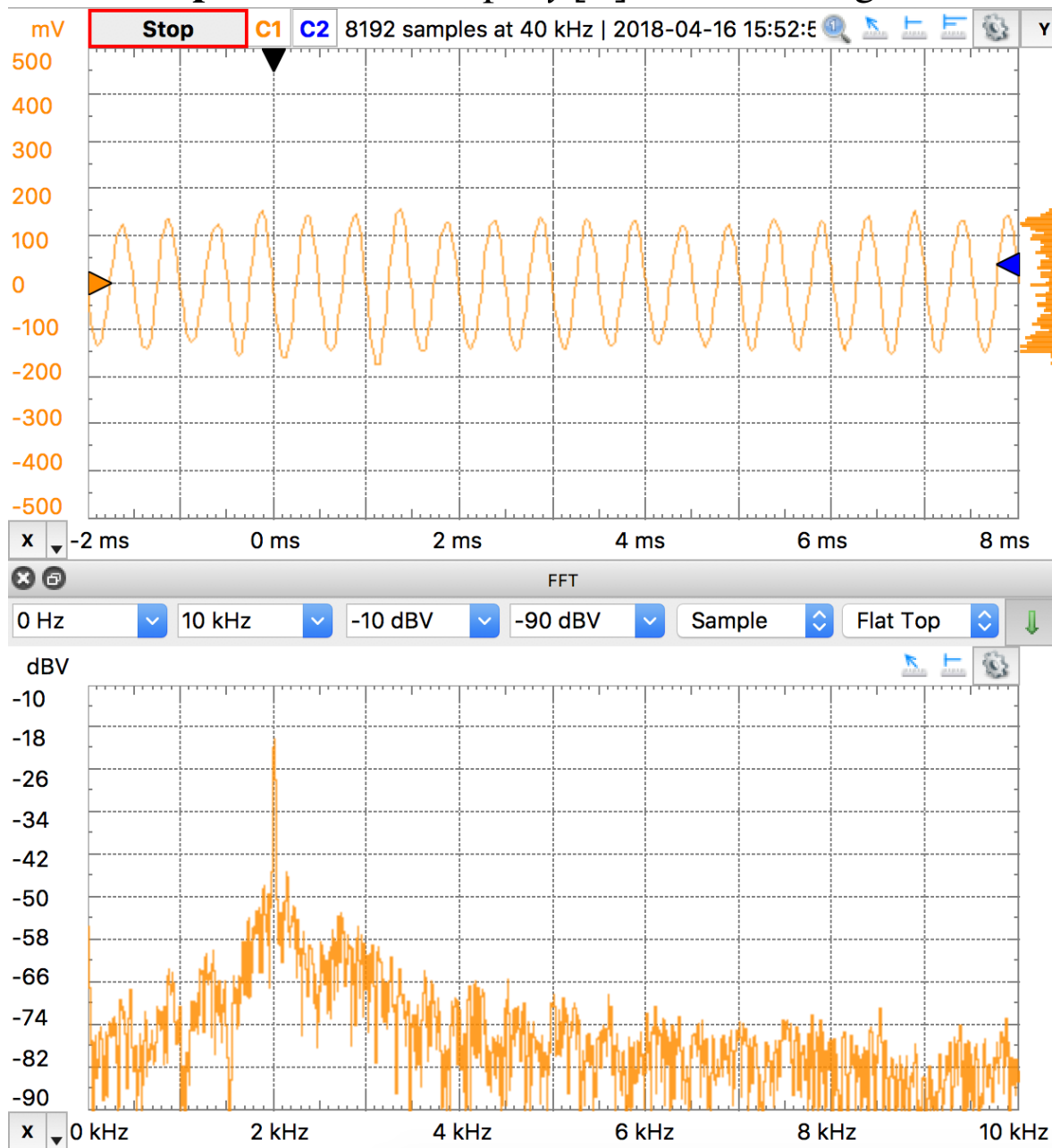
```c
int32_t rand_int32(void)
{
    static int32_t a_start = 100001;

    a_start = (a_start*125) % 2796203;
    return a_start;
}
```
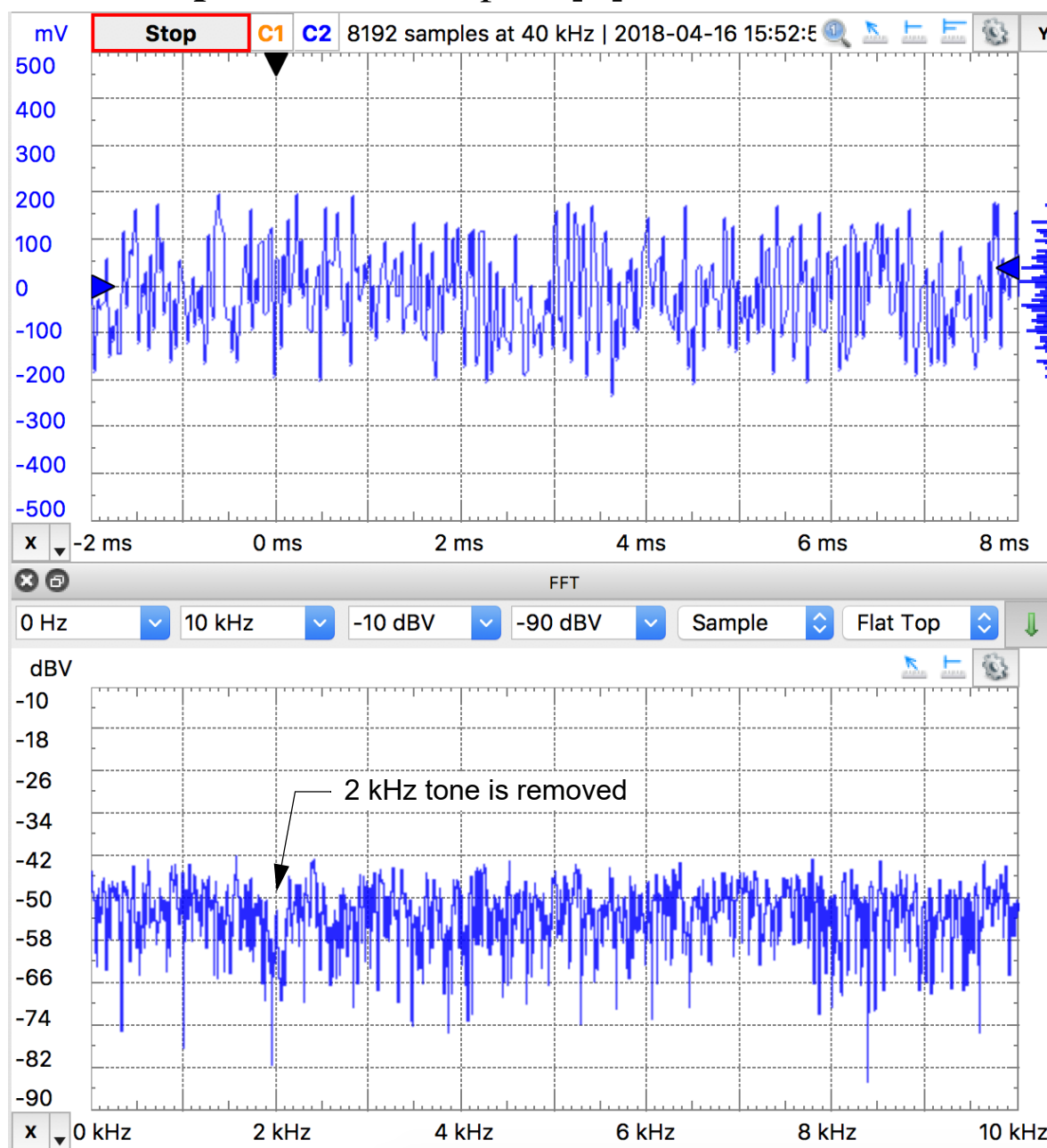
- **ALE Capture**: Filter Output $y[n]$ for 2 kHz Signal

- **ALE Capture**: Error Output $e[n]$

FFT

2 kHz tone is removed

- **IC Capture**: