

## Laboratory Exercise #7

*(This lab can be replaced by a project chosen by the student)*

### Nonlinear and Adaptive Processing

#### Introduction

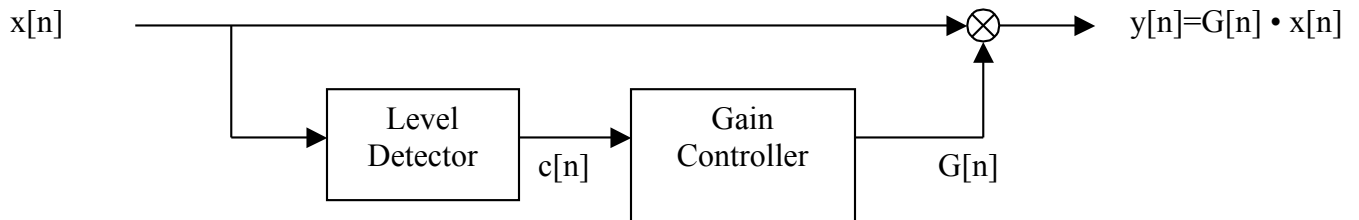
An important area of real time DSP involves nonlinear and adaptive algorithms. Nonlinear systems include modulation/demodulation in communications, automatic gain control, noise and interference suppression, and various types of pulse forming and wave shaping schemes.

In this experiment you will investigate a few processing tasks with data-dependent behavior, and systems in which signals are multiplied together. While for linear systems we have a great deal of analytical muscle in the form of transforms and decompositions based on the superposition property, for nonlinear systems we often must use “linearizing” assumptions like small signal models and perturbation analysis.

#### Dynamic Range Compression

There are a variety of situations in which it is desirable to modify the level of a signal using some sort of automatic adjustment. For example, we may have a detection algorithm that works best if its input signal is maintained at a relatively constant amplitude even if the signal itself varies greatly with time. Another example is compensation for channel characteristics. We may find that the *dynamic range* of a signal is too great to fit within the dynamic range of a channel due to the presence of low-level noise or high-level distortion. In this case we would like to adjust automatically the level of the input signal so that it stays within the allowable dynamic range. This sort of automatic level adjustment is known as an *automatic gain control (AGC)*, a dynamic range compressor/expander, or as a signal *limiter*.

The basic gain control concept is depicted in Figure 1. The input signal,  $x[n]$ , is sent through a *side chain* consisting of a level detection block and a gain control block. The level detector computes a useful metric of some kind that indicates the current amplitude, signal power, or loudness envelope of the input signal. The level measurement,  $c[n]$ , is then fed into a block that computes the proper gain multiplier signal,  $G[n]$ , to apply to the input signal.



**Figure 1:** Automatic gain control structure.

There are many ways to determine  $c[n]$  and  $G[n]$ , depending on the needs of the application. One method is to have the level depend on the absolute value of the input signal, with some “memory” of the previous values to smooth the response. This can be achieved with the expression

$$c[n] = \alpha c[n-1] + (1-\alpha)|x[n]|$$

where  $\alpha$  controls the balance between quickly following the instantaneous amplitude ( $\alpha \ll 1$ ) and more gradually tracking the ups and downs of the signal ( $\alpha \approx 1$ ). It is also possible to make the attack time of the level detector (rate at which  $c[n]$  increases) different from the decay time (rate at which  $c[n]$  decreases). For audio signal processing, the choice of attack and decay time constants is usually a tradeoff between responding promptly to level changes in the input signal, while avoiding abrupt gain changes that would result in clicks or noticeable signal distortion.

Once  $c[n]$  is calculated, its value is processed to determine the actual gain control value,  $G[n]$ , for the gain adjustment. If it is desirable to limit the maximum value of the input signal, we need to design a *compressor* function that reduces the gain when  $c[n]$  exceeds a threshold value. This gain reduction is sometimes called the *compression ratio*, which is expressed as the amount of dB change in the input signal that causes a unit dB change in the output signal, such as 2:1 or 3:1. For example, if the compression ratio is 3:1, a 3dB change in the input signal level causes just a 1dB change in the output level.

### ⇒ Exercise A: Implement and test an automatic gain control

Write a real time DSP program that implements the non-linear gain control block diagram shown in Figure 1.

The signal level detector should be a simple envelope follower or “leaky” peak detector. One possibility would be:

$$\begin{aligned} &\text{if } (|x[n]| > c[n-1]) \\ &\quad c[n] = \alpha c[n-1] \\ &\text{else } c[n] = \beta c[n-1] \end{aligned}$$

where  $\beta > 1$  controls the attack rate and  $\alpha < 1$  controls the decay rate. Feel free to consider an envelope detector of your own design!

Your gain compressor must provide a gain reduction when the signal peak level exceeds 50% of full-scale, and the gain control must have attack time and release time constants that can be varied independently over the range from about 1ms to 30ms. The gain reduction should be chosen so that when the input signal level is near full-scale, the output signal level is 70%.

Using the test file `leveltst.wav` that will be posted to the course web site, play the signal and observe the output using the oscilloscope, or better yet, figure out a way to record portions of the output signal—perhaps using another computer to do the recording. Make sure that you adjust the input signal so that the signal exceeds the 50% threshold, but not so high that it clips the A/D converter. Carefully sketch the output signal for several values of attack and decay time constants. Also try your compressor with some speech and music signals. Include in your report a description of the results.

#### **Additional exercise for Graduate Students:**

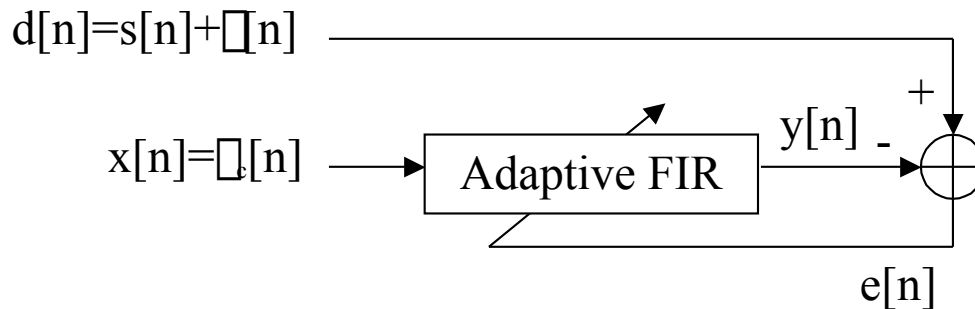
Modify your program to be a gain *expander*: reduce the gain when the signal level goes below 50% of full scale. Adjust the expansion factor so that the output signal level is near zero when the input signal is below 10% of full scale. Carefully verify that your code works properly and include your results and tests in your report.

## Adaptive Interference Cancellation

The most common type of adaptive filter is based on the familiar FIR structure. The concept of an adaptive filter is to adjust automatically the filter coefficients so that the average squared-error between the filter output signal and a reference signal is minimized. The adaptation process occurs incrementally: the filter coefficients are updated gradually at each step in the process, and the output error is used to determine the amount of change to be made at each step. As long as the input and reference signals are correlated and the statistics of the signals does not change too rapidly, the adaptive filter will reach a stable state and provide

the desired filtered output signal. It is important to keep in mind that the adaptive process generally takes some time to converge, so this must be taken into account when designing the signal processing system.

A typical application of adaptive filtering is depicted in Figure 2. Note that the filter structure has two inputs,  $x[n]$  and  $d[n]$ , and the “error” signal is the difference between the filter output,  $y[n]$ , and the reference signal  $d[n]$ .



**Figure 2:** Adaptive noise canceller diagram.

In this example, the reference signal is a “measured” signal consisting of the desired signal,  $s[n]$ , and an unknown additive noise,  $v[n]$ . We also have a separately measured signal  $v[n]$  that is correlated with the noise in the signal. This would be the example if we had a microphone recording someone talking in a noisy room ( $s[n]$  is speech,  $v[n]$  is background noise) while we also had a separate microphone located some distance away that was recording mostly the background noise ( $v[n]$ ). We would like to adjust the adaptive FIR filter so that we minimize the power of the signal  $e[n]$ . This would occur if the adaptive filter was able to make its output  $y[n]$  as close as possible to the unknown noise signal  $v[n]$ , so that  $e[n]$  would become approximately  $s[n]$ , the desired speech-only signal.

So how is this possible? Without looking at the theory in detail, the concept is to alter the coefficients of the FIR filter in such a way that the expected value of  $e^2[n]$  tends toward a minimum. Consider the mathematical expression

$$\begin{aligned} e[n] &= d[n] - y[n] \\ &= d[n] - B^T[n] X[n], \end{aligned}$$

where  $B^T[n]$  is the vector (transpose) of FIR filter coefficients, and  $X[n]$  is the vector of current and past input samples of  $x[n]$ . The squared error is therefore given by

$$e^2[n] = d^2[n] - 2d[n]X^T[n]B[n] + B^T[n]X[n]X^T[n]B[n]$$

and the expected value  $E\{ \}$  of  $e^2[n]$  is

$$E\{ e^2[n] \} = E\{ d^2[n] \} - 2R(x,d)B[n] + B^T[n]R(x,x)B[n],$$

where  $R(\cdot)$  indicates a correlation matrix. Note that this expression involves at most the square of the filter parameters, so it should be possible (for stationary signals) to find the adjustment of each filter coefficient to cause a decrease in the mean-square error. In other words, we should find the gradient of the mean-square error with respect to each of the coefficients, then update each coefficient to follow the gradient's steepest decrease.

In theory, it is possible to do the minimization perfectly if we know the exact autocorrelation and cross correlation information for  $x[n]$  and  $d[n]$ , and everything is stationary. In practice, however, we usually do not have truly stationary signals and we also don't typically know the correlation matrices, or they may drift with time. Thus, we need to use an approximation to the theoretical method.

## The LMS Algorithm

The LMS (least-mean-squares) algorithm is a popular method to obtain an approximate solution to the error minimization problem. The algorithm is expressed as an update equation: the *next* set of filter coefficients are calculated from the *current* set of coefficients and an estimate of the error gradient.

$$B[n+1] = B[n] - 0.5K \hat{E} \{ e^2[n] \}$$

where  $K$  is the *loop gain* parameter (positive number less than 1) which controls the rate of convergence. The estimate of the gradient of the mean-square error for the LMS algorithm is usually just taken as the gradient of the current error sample, which turns out to be simply  $-2e[n]X[n]$ . This makes the LMS expression become

$$B[n+1] = B[n] + K e[n] X[n]$$

So, for each FIR coefficient  $b_i[n]$  we can write the individual update expression as

$$b_i[n+1] = b_i[n] + K e[n] x[n-i]$$

The steps to the LMS algorithm can be summarized as follows.

1. Obtain next samples of  $d[n]$  and  $x[n]$
2. Run FIR filter on  $x[n]$  to obtain  $y[n]$
3. Subtract  $y[n]$  from  $d[n]$  to get  $e[n]$
4. Apply the update expression to get each  $b_i[n+1]$  for next time
5. Repeat

The choice of  $K$  can be a bit arbitrary in practice, but it is best to start with a relatively small number, say, 0.25. Increasing the value of  $K$  towards 1 will speed up the adaptation process, but this may make the filter coefficients hop around and fail to converge. Decreasing the value of  $K$  towards zero helps to stabilize the adaptation, but it also slows down the rate of convergence and this may be undesirable if the filter needs to track changes in the input signals.

### ⇒ Exercise B: Adaptive noise reduction

Write an assembly language program for the 56307 EVM that implements the LMS algorithm. An example framework is given below. Start with a filter of length 8, and then consider trying some other filter lengths. Your code should assume the  $d[n]$  signal is in the left channel and the correlated noise signal  $x[n]$  is in the right channel. Note that the desired output signal is  $e[n]$ . The code below does not contain any deliberate errors, but it HAS NOT BEEN FULLY TESTED: BE SURE to understand what the code is doing and fix any bugs!!

```
adapt_fir    macro ntaps,kval
; On entry the following must be already set:
; Accumulator A has d[n]
; Accumulator B has x[n]
; r0 points to filter state (x memory)
; r4 points to filter coefficients (y memory)
; m0, m4 set for ntaps-1

; Do FIR filter on contents of a
    move b,x0
    clr b                x0,x:(r0)+  y:(r4)+,y0
    rep #ntaps-1
```

```

        mac x0,y0,b          x:(r0)+,x0  y:(r4)+,y0
        macr x0,y0,b
; Filter output is in b
; Now calculate e[n]
        sub b,a

; NOTE: you need to do something here to save contents of a (it is the output signal)

; Calculate kval * e[n]
        move #kval,y1
        move a,x1
        mpy x1,y1,a
        move a,y1

; Now update the coefficients
        do    #ntaps,update_loop
        move y:(r4),a          x:(r0)+,x0
        mac x0,y1,a
        move a,y:(r4)+
update_loop
; reset filter state pointer
        lua (r0)-,r0
        nop

; NOTE: you need to arrange for the output signal to be in the right registers

endm

```

A few test files will be placed on the course web site. `buzzsine.wav` has a noisy sine wave in one channel and correlated noise in the other channel. `buzztalk.wav` contains a speech signal in one channel that is contaminated by a “noise” signal, and then a correlated noise sound is in the other channel. Make sure the desired signal ends up in the accumulator you expect, e.g., `d[n]` in A and `x[n]` in B !

Play the test files, observe the “desired” output signal, and comment on the results. Why isn’t the cancellation perfect? Try some different values of filter length (`ntaps`) and loop gain, `K`. Describe the system behavior.

## **Report and Grading Checklist**

### **A: Automatic gain control**

Commented assembly code and description of your approach.  
Results for provided test signals, and validation of attack/decay and level behavior.  
Comments on results with speech and music signals.  
Grad student exercise, if applicable.

### **B: Adaptive interference suppression**

Source code and comments for your noise reduction algorithm.  
Description of debugging and verification.  
Results for provided test signals, and comments on the details.

*Grading Guidelines (for each grade, you must also satisfy the requirements of all lower grades):*

- F Anything less than what is necessary for a D.
- D Exercise A code with meager evaluation.
- C Exercise A validation and comments. Results for several time constants.
- B Exercise B source code and comments, little evaluation or verification.
- A Exercise B with complete results and description.

*Grad student grades also require the additional exercise (part A).*