Nathan Monson

ECE 529

**A Desktop Implementation of LMS and RLS Adaptive Filter Algorithms**

Digital filtering is the process of performing computational operations on discretized, quantized data for the purpose of extracting, enhancing or shaping signal characteristics. Advances in process technology and the optimization of DSP chips have led to the advent of cheap and fast digital filtering of large streams of data. The availability of cheap DSP has revolutionized wireless communications, image processing, sound processing, and array processing, among other applications. Sometime though we wish to filter signals that are not wide-sense stationary, in other words the autocorrelation of the signal is changing over time. Pre-calculated filters cannot be used on these signals. Adaptive filtering seeks to produce filters that can perform desired filtering behaviors on these changing systems. Adaptive filters are becoming more and more common as the computational power required to implement them becomes more available. Rather than a filter having fixed coefficients (often referred to as tap weights in the literature), adaptive filters dynamically optimize coefficients based on state parameters. Common applications of adaptive filters are in system and channel identification and echo and noise cancellation. For example, you might have an unknown filter or system you wish to characterize. By placing an adaptive filter in parallel with this system and comparing the outputs of the system you can tune your adaptive filter to match the unknown system. When the outputs converge, you have a match.

In the text below we will demonstrate the implementation of a pair of adaptive filter algorithms. These two filter implementations are the least-mean-square adaptive filter algorithm and the recursive least squares algorithm. Theoretical underpinnings of the filters are discussed followed by their implementations and sample results. Interspersed throughout are comments on deficiencies and performance characteristics of the algorithms.

**Adaptive Filter Basics**

An adaptive filter can be broken down into two major components as shown in the following figure (from Wikipedia [8]):

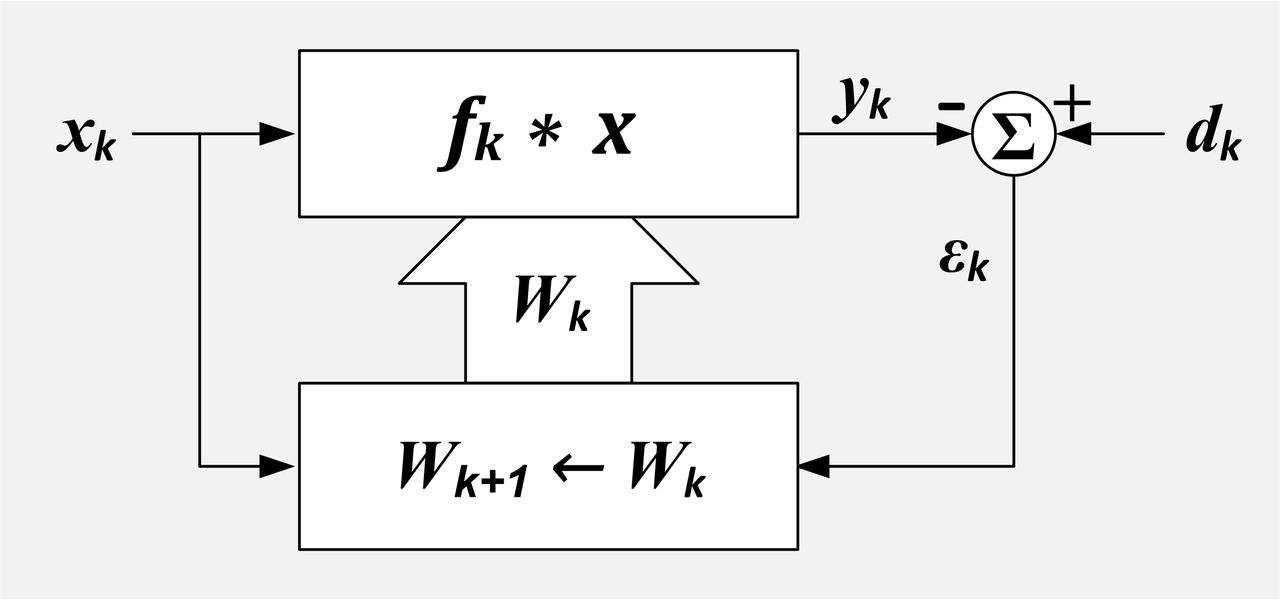


Figure 1. Block diagram for one possible realization of an adaptive filter.

The components are the filtering operation (indicated by the top box) and the optimization algorithm (indicated by the bottom box). As is common to all sorts of filters, not just adaptive filters, the filtering operation can be represented in the time domain as a convolution operation between the filter impulse response and the input delay line from the sampling operation. Adaption comes into play when we include the other component which is the computational block that will update our filter weight coefficients.

In a non-adaptive filter, we determine our desired filter characteristics and design a filter that has a particular transfer function. From the poles and zeroes of this transfer function we are able to determine the various input and feedback terms (for a causal filter) and their associated weights. For applications requiring an adaptive filter, the desired filter characteristics are unknown or are changing and so we calculate the weights dynamically as inputs come in. The specifics of how this dynamic weighting is done will be demonstrated in the computational steps of the adaptive filter algorithms highlighted below.

The distinguishing characteristic of the different types of adaptive filters, as alluded to earlier, is the method by which we calculate filter component weights dynamically. We seek filter weights that will produce an output signal that closely matches a desired output signal. That desired output signal might be the output from an unknown filter we wish to characterize or a noise signal we are measuring that we wish to cancel. In an adaptive filter this the error measurement (or more specifically the mean squared error) between desired and output signal is minimized. Formally, MSE is the average of the error between the estimator (the filter output) and the estimated (the desired signal) and incorporates information about the variance and bias of the filter output. Developing these minimization or optimization operations are part of a broad field of research into mathematical optimization. All adaptive filters will try to minimize the MSE, the various methods of how they do this is the interesting part, and challenging part of their implementation.

A theoretical way to minimize the MSE is to adjust your filter weights by the negative of the gradient of the error. If the gradient is positive, the error is increasing and so the coefficient weights must be adjusted in the opposite direction of the gradient slope. Many algorithms will scale the size of this gradient adjustment, by a step-size (or adaptation) coefficient, commonly seen in the literature as the Greek symbol mu. This coefficient determines how quickly the weights will change between iterations and whether the filter will converge or blow up. The convergence bounds of this coefficient depends statistical characteristics of the input signal.

**LMS Theory**

At its core, the LMS filter works by iteratively adjusting the filter weights by the gradient of the system error. A positive error gradient would indicate that changing the weight in this direction would increase the error of the system. Therefore, we take the negative of the gradient and add that to the tap weights. The mathematical underpinnings of this algorithm are a probabilistic cost optimization function, the mean-square error cost function. The least-mean-square cost function can be thought of as an instantaneous estimate of the mean-square error cost function (MSE).

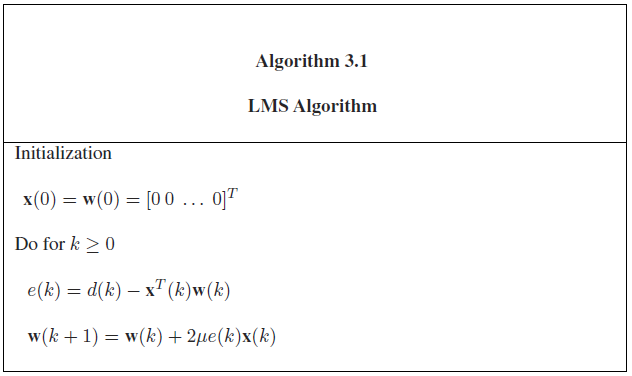


Figure 2. (From Diniz 2013)

The basic steps of the algorithm for a causal filter are highlighted in the above figure. As the algorithm is causal the input array (x(k)) is initialized to zero as is the initial weights (w(k)). If we have some prior knowledge about the system, we can initialize the weights to those starting values. Next, as we process each input index, we will update the error e(k), which is the desired signal d(k) minus the output at that time-point which is the sum of the terms of the transposed input matrix multiplied by the weights for each current (k) or feedback (k-n) term. In the implementation below, I have separated this into two distinct steps. Finally, the weight vector for the next iteration is calculated by adjusting each current weight term by a value proportional to the product of the adaptation constant, the current error term, and the corresponding input value. This is the step where the movement to minimize the error occurs.

We mentioned earlier the adaptation coefficient (mu). Choosing an appropriate mu that produces a convergent filter for a traditional LMS filter can be difficult. For this reason, LMS filter variants have been developed such as the normalized LMS (NLMS), that normalizes the tap weight update sum by the power of the input signal.

**Translating Theory into Practice**

As we saw above, computationally intensive operations such as differentiation are used in optimization algorithms. If we can reduce these operations to multiplication and accumulation operations in filter realization, we can improve computational efficiency.

**Implementation Details**

These adaptive filtering algorithms were implemented in software, in C, on a desktop microprocessor. As the project was coded using Visual Studio 2015, code was compiled using the Microsoft C/C++ compiler. A wrapper main function was used to generate the signals, output files, and call the filtering algorithms. This is decidedly not a real-time implementation and samples are read in and buffered in full before the filtering takes place.

**LMS Algorithm and Filter Results**

/\* LMS adaptive filter

\*

\* Author: Nathan Monson

\*

\* Parameters:

\* x - most recent input sample

\* w - vector of filter term weights

\* d - value of the desired output at this index

\* mu - the step size (also referred to as the adaptation constant

\* return\_array - results of computatation passed in this array - the filter output value in the first

\* index and the error value in the second index

\*

\* This function implements the three main computational steps of an LMS filter:

\* (1) Update the input\_buffer with the most recent sample and calculate filter output (y) by mult-accumulate

\* the input\_buffer by the weights of the filter terms (w)

\* (2) Calculate the error term by subtracting the desired output value (d) from the calculated output value

\* (3) Update the weights vector (w) by adding the update term at each index

\*/

void lms\_filter(double x, double w[], double d, double mu, double return\_array[])

{

double error = 0.0;

double y = 0.0;

//stores the input(x) values that have previously been fed into the filter, most recent at index 0

static double input\_buffer[FILTER\_ORDER];

for (int i = 0; i < FILTER\_ORDER; i++)

{

input\_buffer[i + 1] = input\_buffer[i]; //shift previous input values

}

input\_buffer[0] = x; //push most recent input value into input\_buffer

for (int j = 0; j < FILTER\_ORDER; j++)

{

y += input\_buffer[j] \* w[j]; // calculate filter output

}

return\_array[0] = y;

error = d - y; // calculate error term

return\_array[1] = error;

for (int k = 0; k < FILTER\_ORDER; k++)

{

w[k] = w[k] + (mu \* error \* input\_buffer[k]); //update filter weight vector

}

}

Below are a collection of plots indicating how the algorithm performs with varied parameters, input signals and desired waveforms. Pay particular attention to the stability of the systems and how long it takes the signal to converge on the desired waveform.

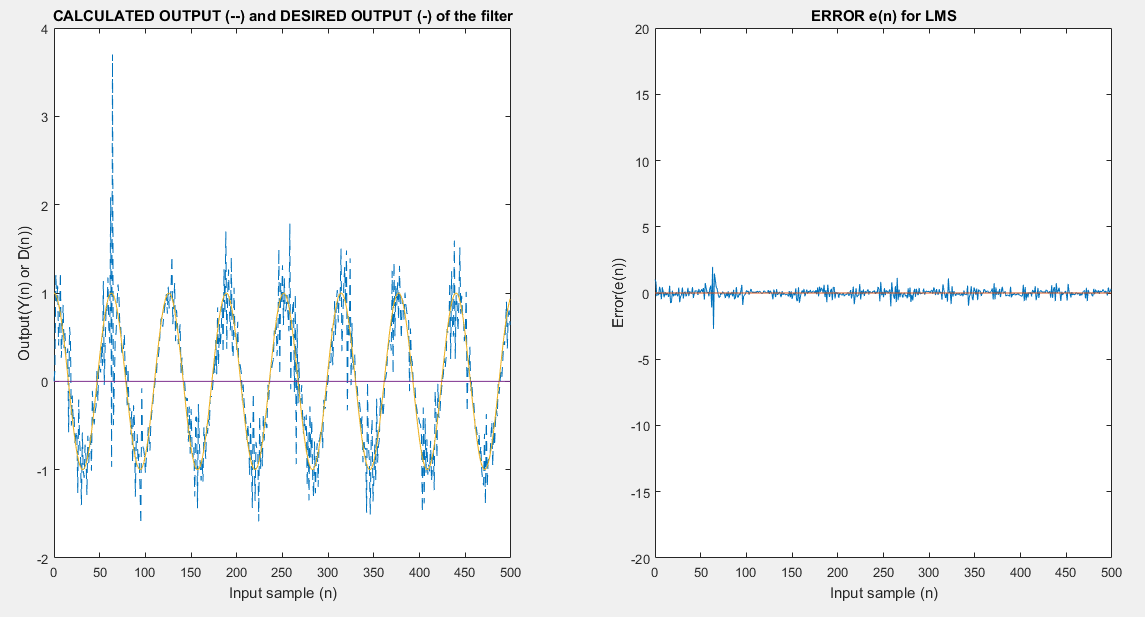


Figure 3. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 10; Desired waveform (d) = cos(0.1n); Step size (mu) = 0.1; Filter order = 10

We have some adaptation and tracking of the desired signal!

*Effects of changes to adaptation coefficient*

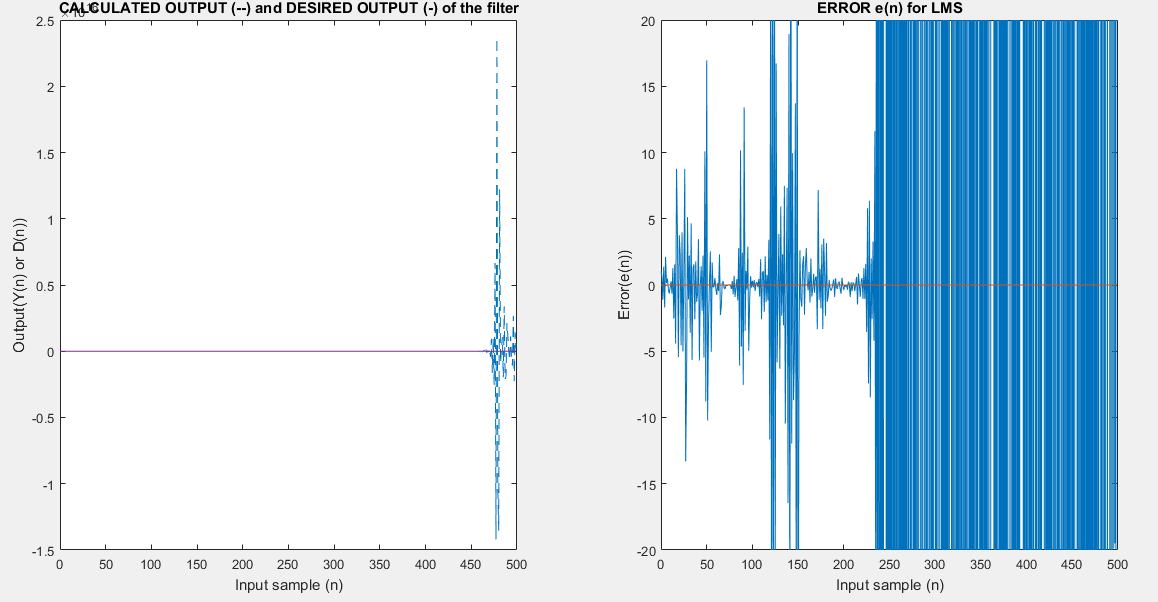


Figure 4. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 10; Desired waveform (d) = cos(0.1n); Step size (mu) = 0.2; Filter order = 10

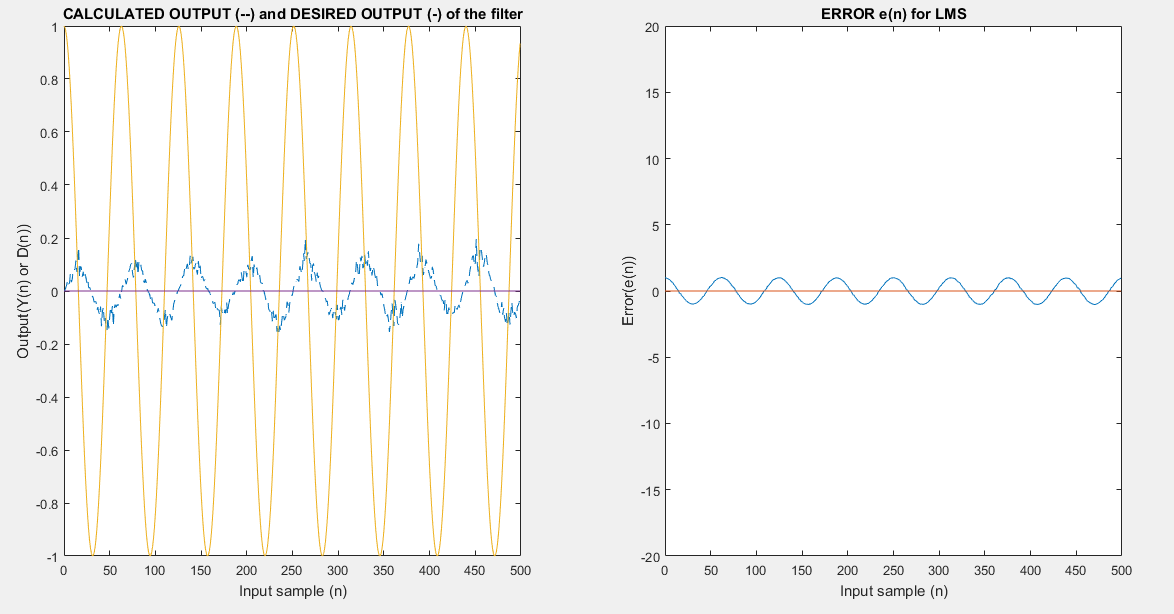


Figure 5. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 10; Desired waveform (d) = cos(0.1n); Step size (mu) = 0.001; Filter order = 10

As you can see too large a mu causes the filter to blow up and not converge, while a small mu can cause the filter to adapt very slowly. Notice in Figure 4 that the output signal lags the desired signal so that by the time the filter output start increasing more rapidly the desired signal is decreasing again and vice-versa.

*Effects of changes to filter order*

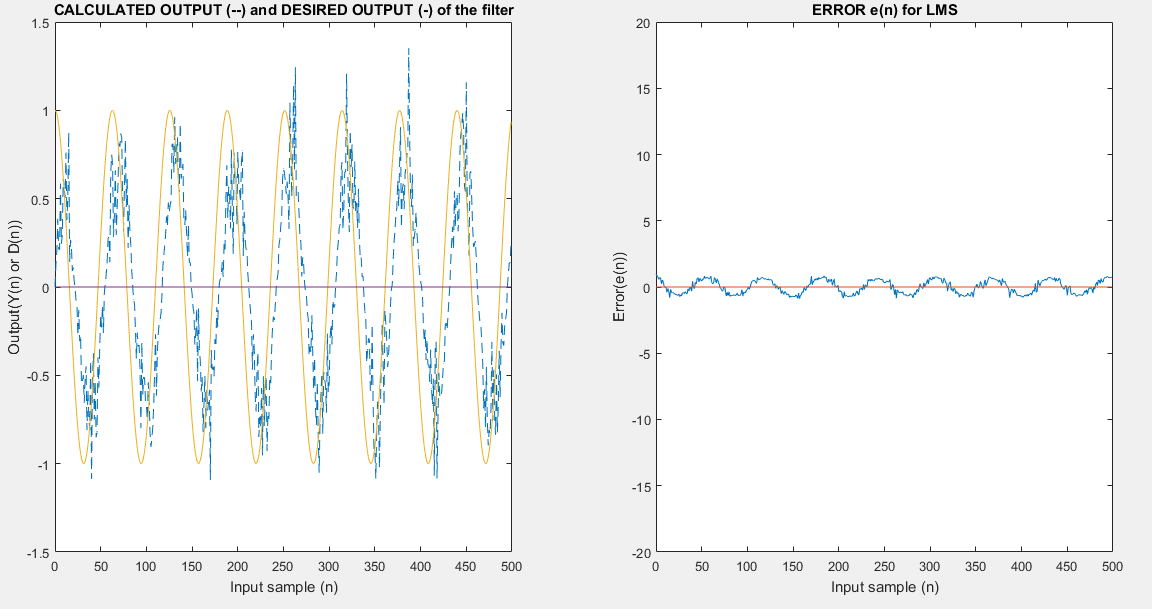


Figure 6. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 10; Desired waveform (d) = cos(0.1n) + 5; Step size (mu) = 0.1; Filter order = 1

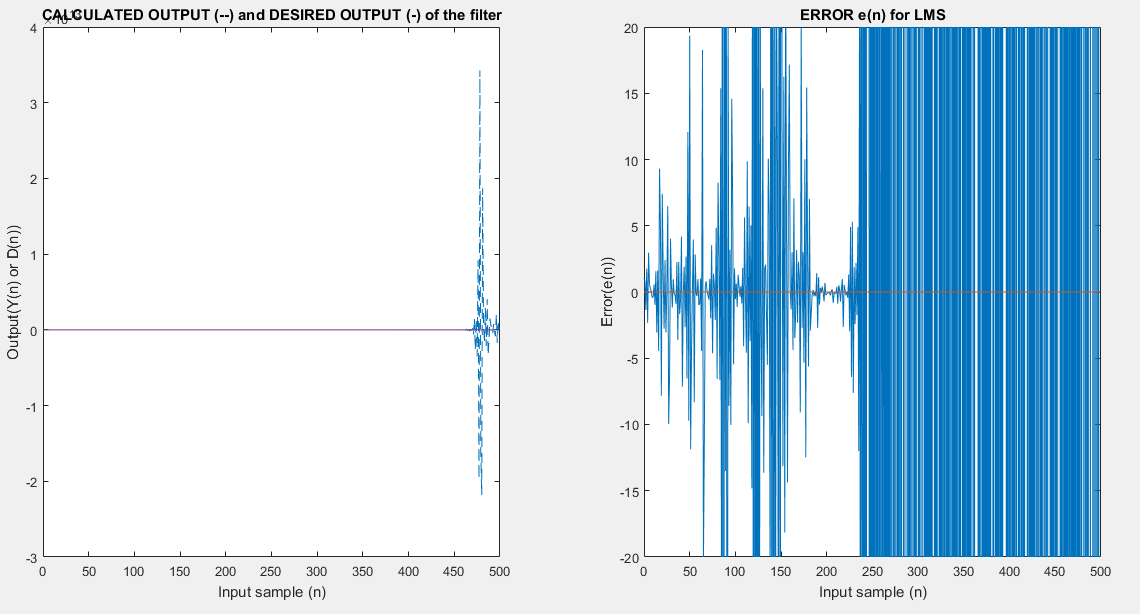


Figure 7. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 10; Desired waveform (d) = cos(0.1n) + 5; Step size (mu) = 0.1; Filter order = 20

A very short filter struggles to track the desired signal rapidly enough, while too many filter terms can cause the filter to blow up at an adaptation coefficient that previously led to convergence.

*Effects of changes to SNR in input signal*

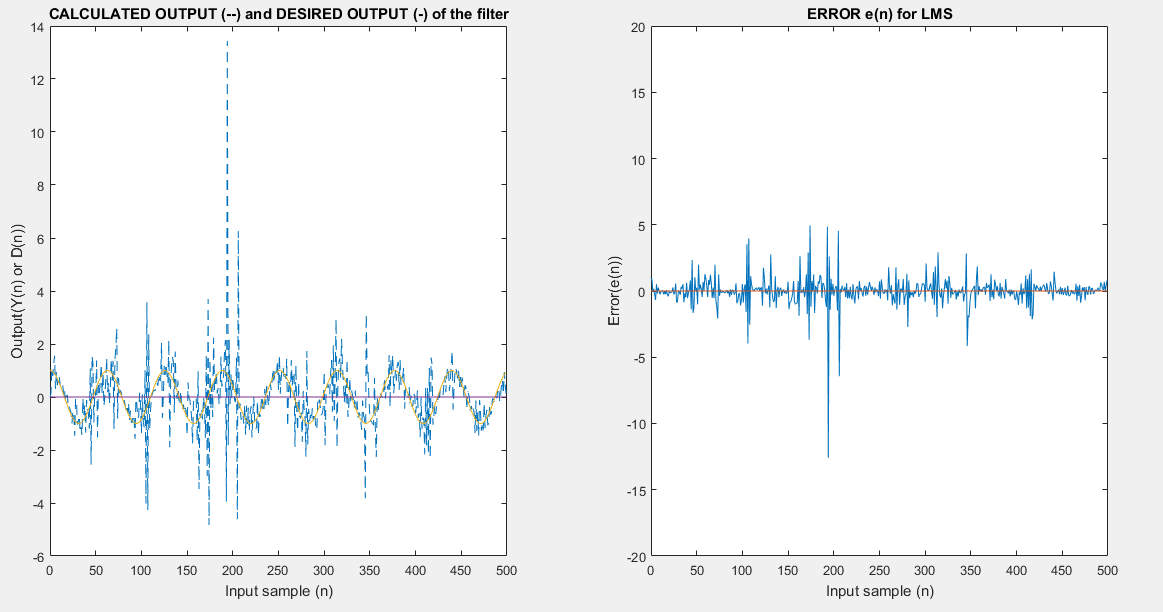


Figure 8. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 5; Desired waveform (d) = cos(0.1n) + 5; Step size (mu) = 0.1; Filter order = 10

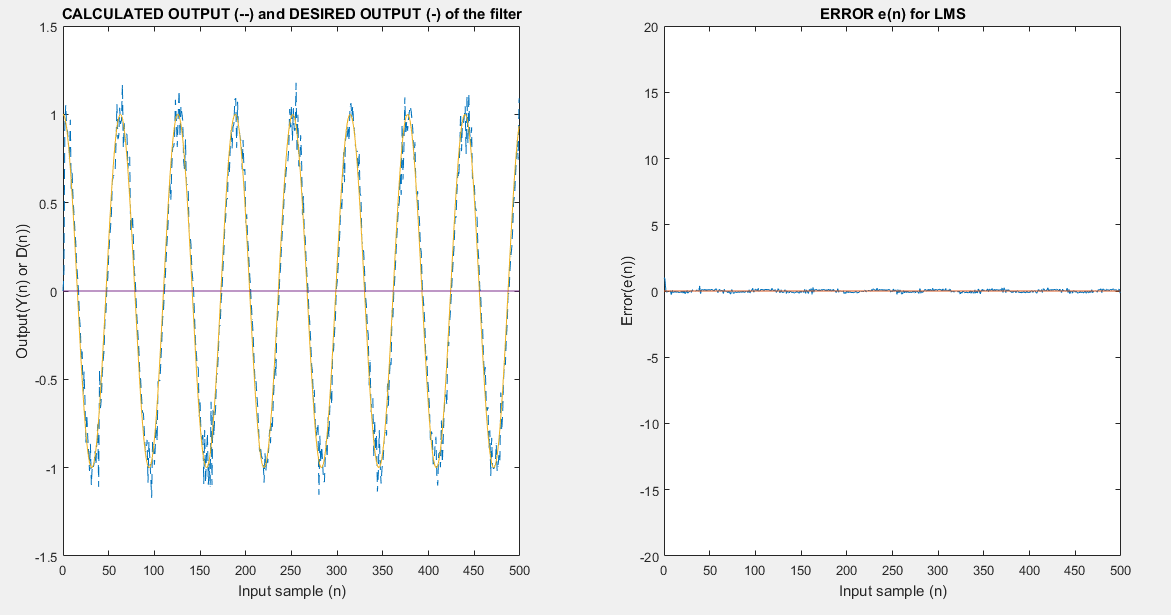


Figure 9. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 20; Desired waveform (d) = cos(0.1n) + 5; Step size (mu) = 0.1; Filter order = 10

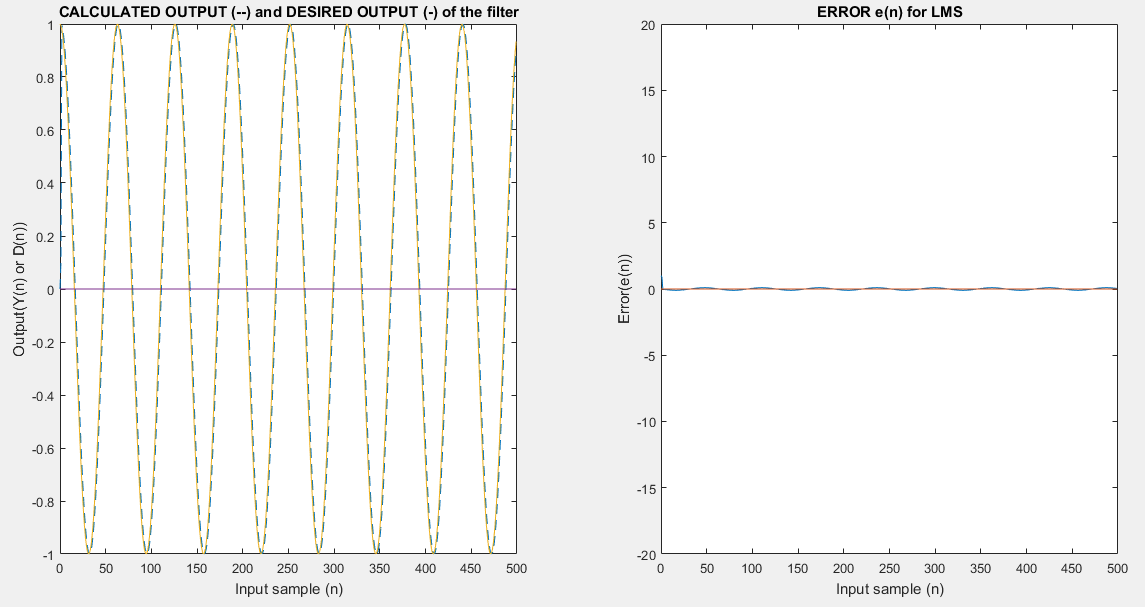


Figure 10. Input waveform (x) = step function with magnitude 1 and Gaussian white noise added at SNR of 100; Desired waveform (d) = cos(0.1n) + 5; Step size (mu) = 0.1; Filter order = 10

Reducing noise in the input signal unsurprisingly makes it easier for the tap weight coefficients to settle on values that minimize the error terms.

**Comparison with RLS Algorithm**

Differences between two of the main choices for optimization algorithms (LMS/statistical and RLS/deterministic)

**Some Notes on Performance**

In order to gauge the effectiveness of the LMS implementation described in this paper, the same signal that was used to generate the above results was used as an input to a MATLAB DSP toolbox LMS Filter with the same parameters as my adaptive filter. The resulting filter output and error signals are shown below in Fig. 11.

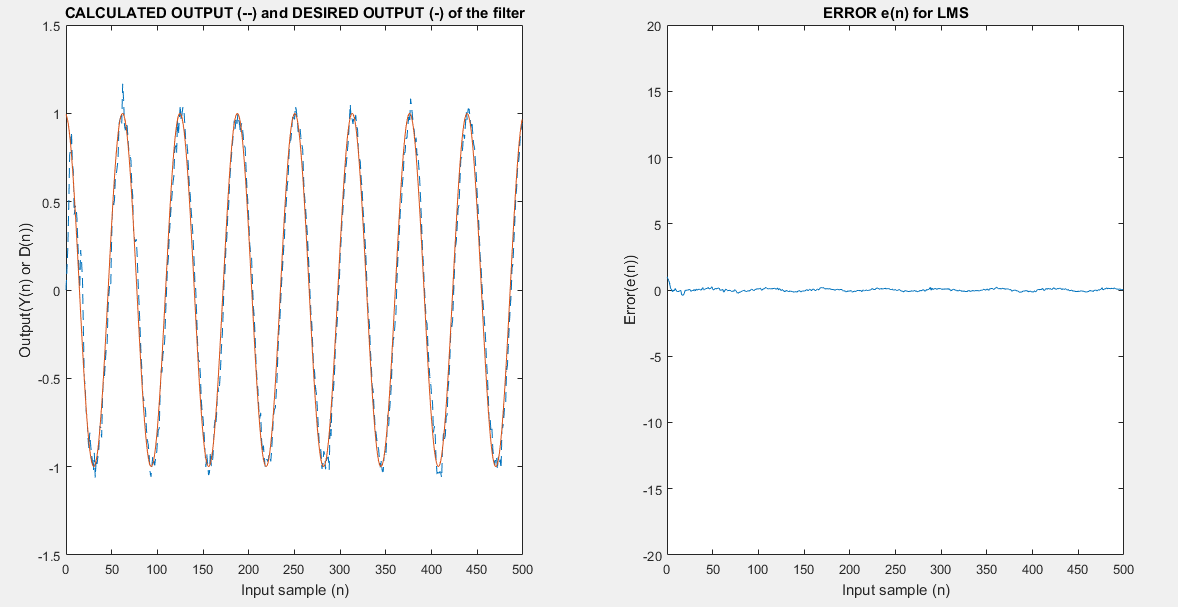


Fig 11. MATLAB LMS Reference Implementation with input of cos(0.1n), filter length of 10 and adaptation coefficient of 0.1

As you can see the reference implementation output signal has a noticeable reduction in the noise compared to my implementation. The output for the reference implementation for an input SNR of 10 looks similar to the output for my implementation for an input SNR of 20. Similarly, the reference implementation is about twice as fast as my implementation (11.3 ms to 21.0 ms). These two results would indicate that perhaps there is a bug somewhere in my implementation, perhaps during the updating of the tap weights.

**Resources**

1. Diniz, P.S. Adaptive Filtering: Algorithms and Practical Application. Springer. 2013.
2. Douglas, S.C. Introduction to Adaptive Filters. Digital Signal Processing Handbook. CRC Press. 1999.
3. Farhang-Boroujeny, B. Adaptive Filters: Theory and Applications. John Wiley and Sons. 2013.
4. Haykin, S. and Widrow, B. Least-mean-square Adaptive Filters. Wiley-Interscience. 2003.
5. Malepati, H. Digital Media Processing. Elsevier. 2010.
6. National Instruments. Least Mean Square Adaptive Filter. <http://www.ni.com/example/31220/en/>. 8/23/2013.
7. Widrow, B. et al. The complex LMS algorithm. Proceedings of the IEEE. 63. 1975. Pp. 719-720.
8. Wikipedia – “Adaptive Filter”. <https://en.wikipedia.org/wiki/Adaptive_filter>

**Wrapper Function Code**

#include <stdio.h>

#include <math.h>

#include <string.h>

#include <cstdlib>

#define SIGNAL\_LENGTH 400 //31

#define FILTER\_ORDER 20

#define DESIRED\_SIGNAL (cos(0.1 \* i)) + 5

void lms\_filter(double x, double w[], double d, double mu, double return\_array[]);

int

main(void)

{

FILE \*output\_fp;

FILE \*error\_fp;

FILE \*desired\_fp;

FILE \*input\_fp;

bool input\_file = false;

double read = 0.0;

char read\_string[100];

output\_fp = fopen("OUTPUT.csv", "w+");

error\_fp = fopen("ERROR.csv", "w+");

desired\_fp = fopen("DESIRED.csv", "w+");

//Comment/uncomment these two lines if you have input data in a csv file

input\_fp = fopen("INPUT.csv", "r+");

input\_file = true;

double x[SIGNAL\_LENGTH] = { 0.0 };

double d[SIGNAL\_LENGTH] = { 0.0 };

double y[SIGNAL\_LENGTH] = { 0.0 };

double w[FILTER\_ORDER] = { 0.0 };

double lms\_return[2] = { 0.0, 0.0 };

double mu = 0.001; //0.35 blows up, 0.3

for (int i = 0; i < SIGNAL\_LENGTH; i++)

{

y[i] = 0.0;

}

for (int i = 0; i < SIGNAL\_LENGTH; i++)

{

if (!input\_file)

{

x[i] = 1.0;

}

else

{

fgets(read\_string, 100, input\_fp);

x[i] = atof(read\_string);

}

}

for (int i = 0; i < FILTER\_ORDER; i++) {

w[i] = 0.0;

}

for (int i = 0; i < SIGNAL\_LENGTH; i++) {

d[i] = DESIRED\_SIGNAL;

if (i < (SIGNAL\_LENGTH - 1)) {

fprintf(desired\_fp, "%lf,\n", d[i]);

}

else

{

fprintf(desired\_fp, "%lf", d[i]);

}

}

for (int i = 0; i < SIGNAL\_LENGTH; i++) {

lms\_filter(x[i], w, d[i], mu, lms\_return);

if (i < (SIGNAL\_LENGTH - 1)) {

fprintf(output\_fp, "%lf,\n", lms\_return[0]);

fprintf(error\_fp, "%lf,\n", lms\_return[1]);

}

else

{

fprintf(output\_fp, "%lf", lms\_return[0]);

fprintf(error\_fp, "%lf", lms\_return[1]);

}

}

fclose(output\_fp);

fclose(error\_fp);

fclose(desired\_fp);

return 0;

}