LMS and it's variant

Chinmay Sahu

Communications, Signal Processing, Networking Lab
Department of Electrical and Computer Engineering
Clarkson Univeristy,Potsdam,13699

April 9, 2019

1 Introduction

An LTI system is excited with white Gaussian noise of zero mean and unit variance. The simulated output of the system is given in Figure 1. LMS and it's variants are applied to the desired signal to estimate filter coefficients with different step sizes. The algorithms are implemented in MATLAB. The length Monte-Carlo simulation is 50. Then, the errors are averaged to plot the evolution of error.

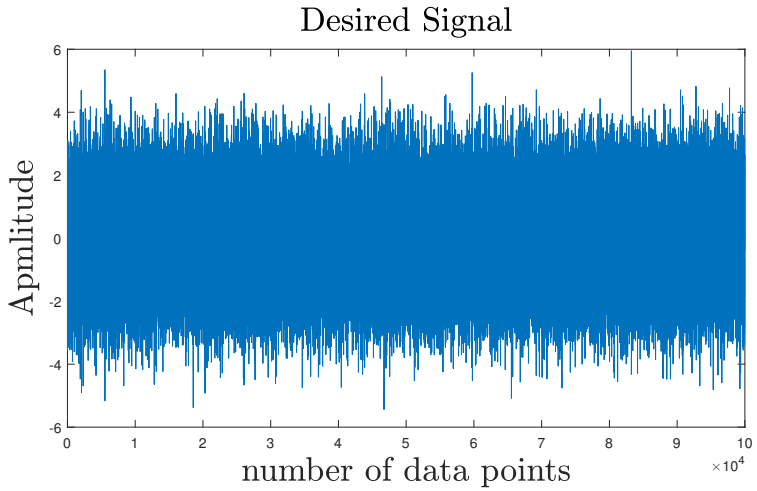

Figure 1: Desired Signal

1.1 LMS

LMS is applied to the given system for different step sizes. The error evolution is plotted for different step sizes for each iteration and is shown in Figure 2 in the next page. We can observe that with a smaller step size of $\mu = 0.001$ the LMS takes longer to converge. While with a step size of $\mu = 0.02$, it converges fast, which is within 1000 data points. With a step size of 0.2, we see that LMS converges to a higher to error value of -500 dB and the error is highly oscillating in and around that error range. This may be due to the use of limited Gaussian data points to simulate desired signal and with this scenario, the strict bound the step size is trace of auto-correlation matrix, which is 0.25. Hence, as the step gets closer to 0.25, the LMS algorithm starts to oscillate and fail to converge.

Similarly, for each step size, the algorithm is run for 50 times and errors are averaged to plot the error evolution in Figure 3. We can see that for a smaller $\mu = 0.001$, LMS takes more time to converge. With step size of $\mu = 0.2$, algorithm converges to minimum error of -600 dB. Due to averaging of errors, oscillations are observed less in the results. Step size of $\mu = 0.02$ works best for this set of simulations. Compared to each iteration shown Figure 2, the errors for a step size $\mu = 0.2$ converges with error as low as -600 dB. During each Monte-Carlo, the white Gaussian is simulated fresh and errors are averaged at the end. This makes the error evolution curve smooth for Monte-Carlo simulation.

The effect of step size and averaging are shown in Figure 4 and 5 respectively. We can see that with bigger step size of $\mu = 0.15$ algorithm seems to converge fast compared to

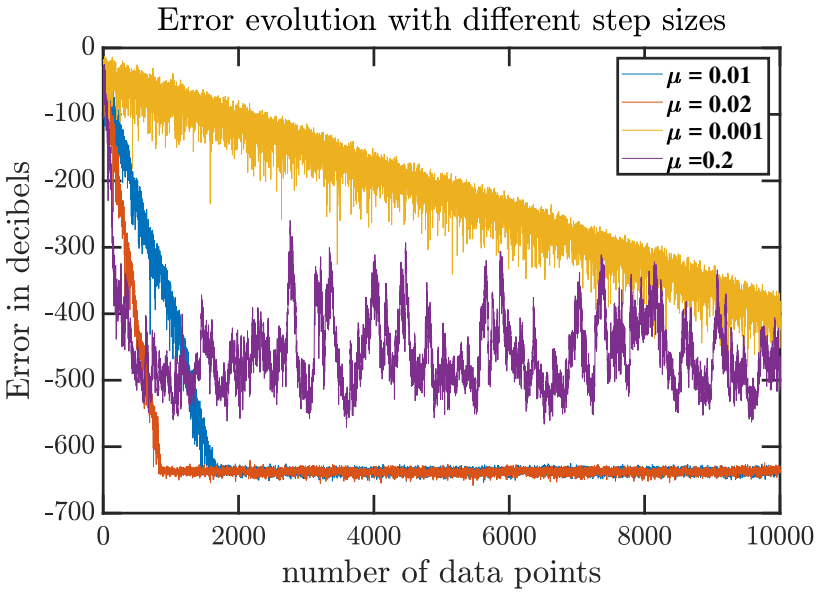

Figure 2: Error evolution for each iteration for LMS for different values of step sizes.

smaller step size of $\mu = 0.02$. However, the oscillations in and around the error are more with higher step size. For a smaller step size, the oscillations are visibly less and Monte-Carlo simulations show the error evolution is even smoother.

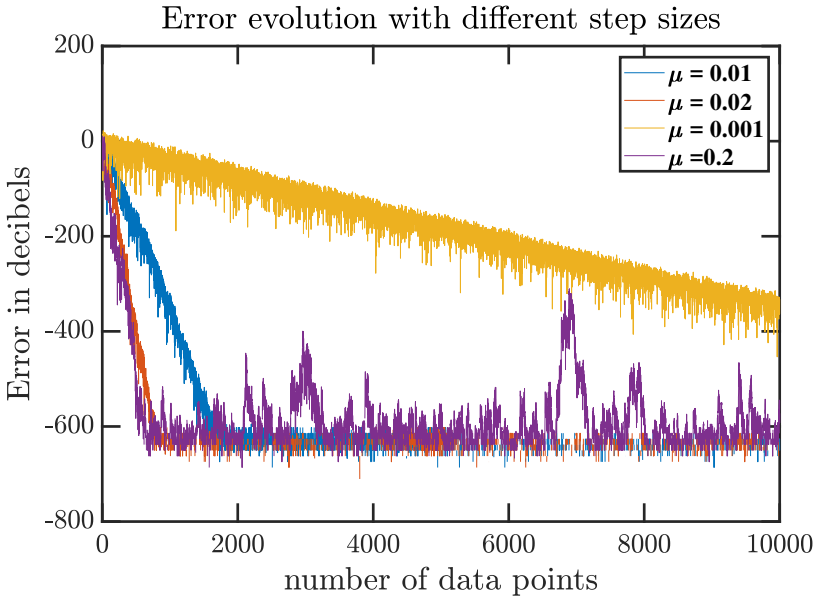

Figure 3: Averaging error evolution in decibels for LMS with a Monte-Carlo size of 50.

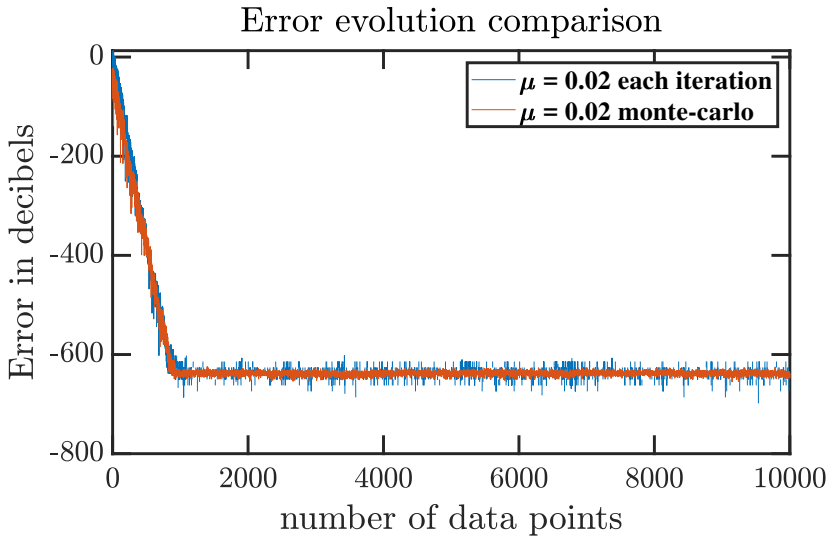

Figure 4: Comparing error evolution in decibels for LMS with each iterations vs monte-carlo for a step small step size of 0.02.

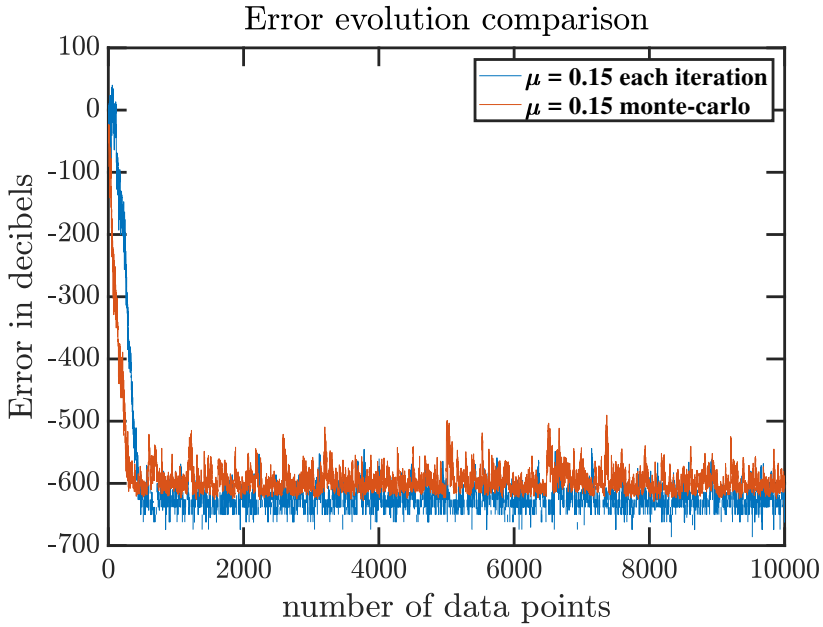

Figure 5: Comparing error evolution in decibels for LMS with each iterations vs monte-carlo for a step step size of 0.15.

1.2 Newton Algorithm

Assuming auto-correlation matrix is estimated perfectly, Newton LMS algorithm is implemented for a step size of $\mu = 0.02$. LMS is compared with Newton algorithm for same step size. The plot is shown in Figure 6. We can see that both plots looks almost identical and on the top of each other. This is because inverse of R matrix is identity matrix and it does not make any change to the existing LMS algorithm.

Using single step newton algorithm does not work in simulations. The errors fail to converge. The simulated white Gaussian noise is not perfect. It is simulated with a limited number of data points. So auto-correlation matrix is not truly ideal and it's inverse is not an Identity matrix. Hence, the algorithm fails to converge.

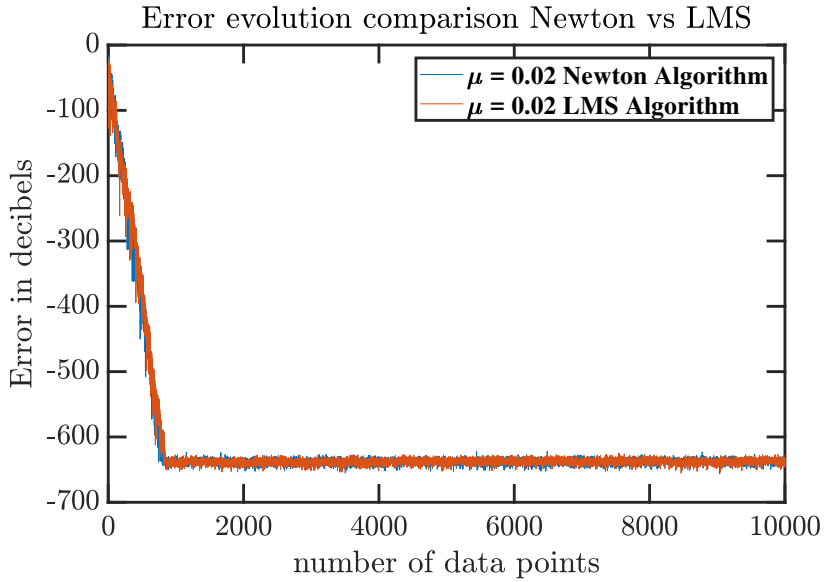

Figure 6: Comparing error evolution in decibels for LMS and Newton Algorithm for a step size of 0.15.

1.3 Normalized LMS

With normalized LMS, the step size is normalized with the filter order and trace of auto-correlation matrix. The simulation of normalized LMS is plotted in Figure 7. The algorithm is run with Monte-Carlo of 50. We can see that it converges really fast and oscillations are around error is also less.

1.4 Block LMS

In block LMS, blocks of samples are processed at the same time. We simulate the block LMS with no overlap and 50 percent overlap. The results are shown in Figure 8. We can see that the error curves are really smooth because of Monte-Carlo. The total number of iterations to simulate the algorithm with 50 percent overlap will be less than no overlap. Blocks with no overlap seems optimal choice from the view point of computational complexity. Block with overlap results in redundant operations in adaptive process as the gradient vector uses more information than the filter itself.

1.5 Frequency Domain Adaptive Filter (FDAF)

When Block LMS is implemented in frequency domain, the resulting algorithm is FDAF. The simulated result of FDAF is shown in Figure 9. We can see that FDAF error curve is

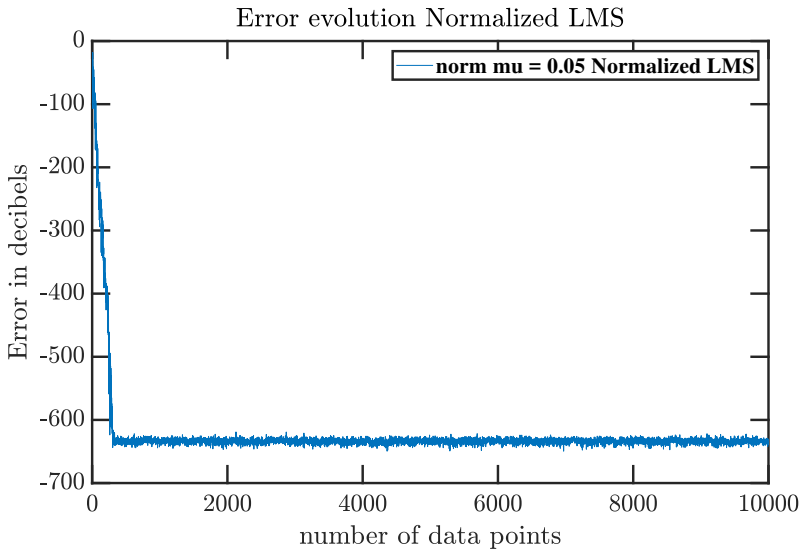

Figure 7: Error evolution in decibels for Normalizes LMS.

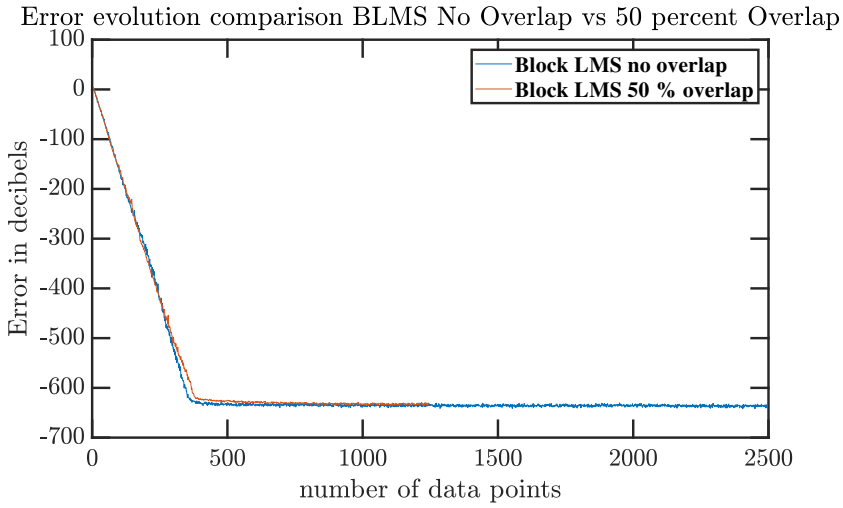

Figure 8: Error evolution in decibels for Block LMS for no overlap vs 50 % percent overlap.

smooth due to Monte-Carlo. The step size used for this is $\mu = 0.05$. The error curve reaches minimum value of -600 dB around 700th data point.

1.6 Comparison and Conclusions

LMS, Newton LMS, Normalized LMS, Block LMS, FDAF are simulated and plotted all together in Figure 10. The step size used is 0.05 and Monte-Carlo length is 50. Apart from FDAF, other algorithms are converging around same time. There seems to be a delay in FDAF algorithm to converge. The Normalized LMS and Block LMS work really well. The error curve for both of them are really smooth. Normalized LMS can perform fast and well if step size is chosen wisely. Otherwise, the step size can be really small and the convergence time will be more. In Block LMS, we have to use block size optimally to use algorithm

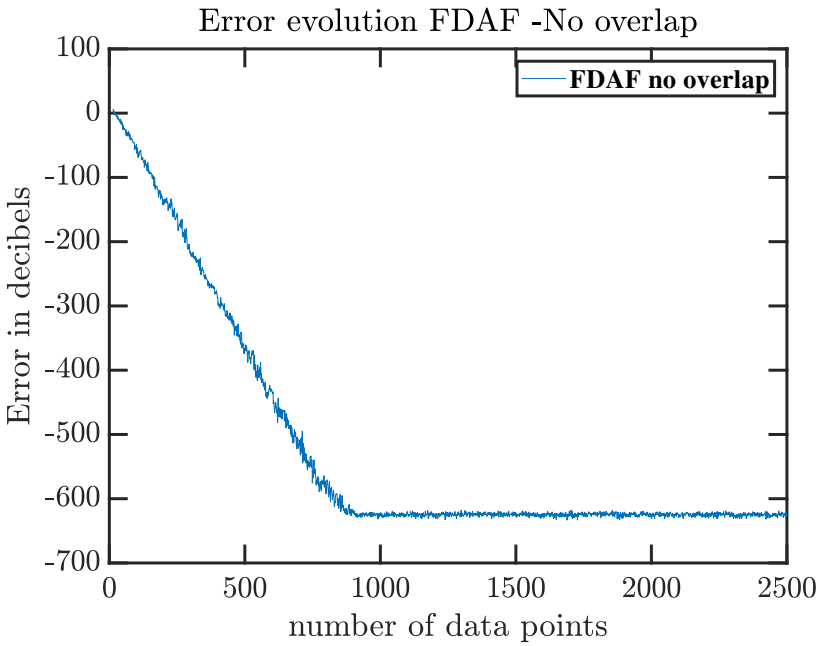

Figure 9: Error evolution in decibels for FDAF for no overlap.

efficiently. So it is trade off between block size and step size for fast convergence between normalized LMS and Block LMS.

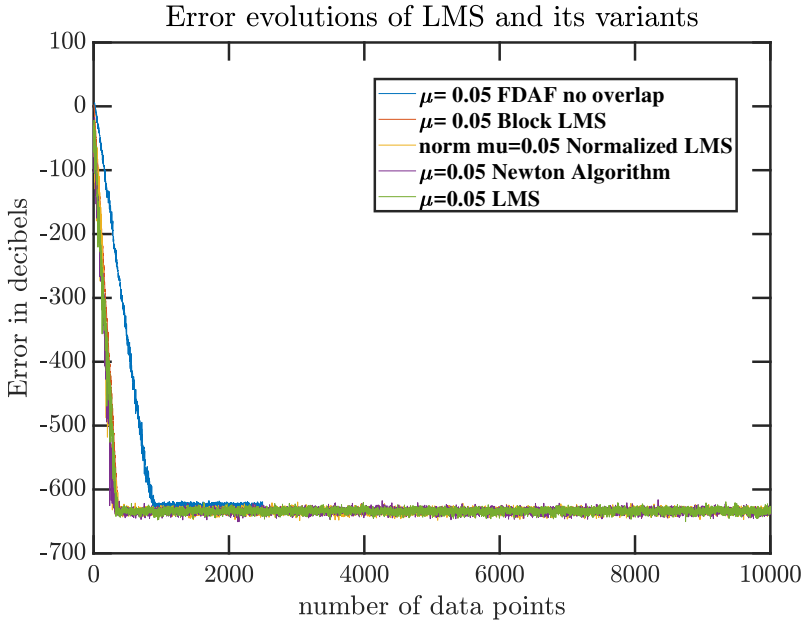

Figure 10: Comparisons of error evolution of LMS and it's variants.

2 Filters for all Pole systems

An LTI system with a given impulse response is simulated with Gaussian noise of zero mean and unit variance. The desired response of the system is shown in Figure 11. LMS, Output-error algorithm, Simple hyper-stable adaptive recursive filter (SHARF), Levinson-Durbin algorithms are simulated for estimating filter coefficients for an all pole system.

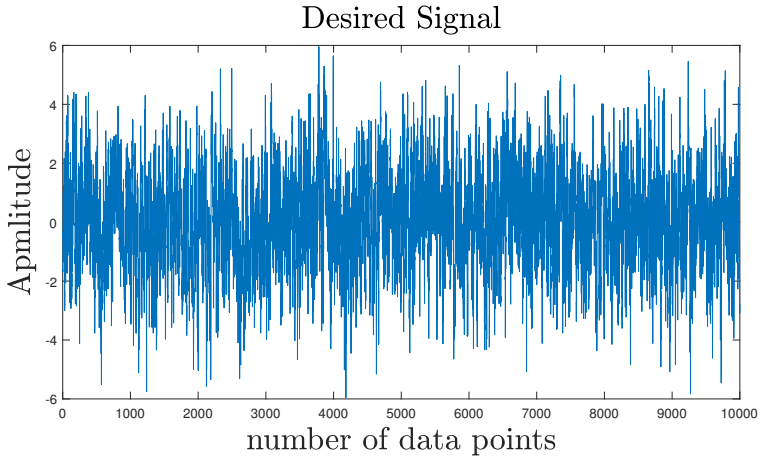

Figure 11: Desired Signal

2.1 LMS

An LMS is algorithm is adapted to all pole systems. Assuming filter order L=2,4,10, the frequency response and impulse response of them are implemented and evaluated. Error evolution for order 2 has been shown in Figure 12. We can see that errors are high and close to zero in decibels. This indicates that LMS is not able to estimate filter coefficients for all pole systems perfectly. The estimates are approximate and not close enough to true value.

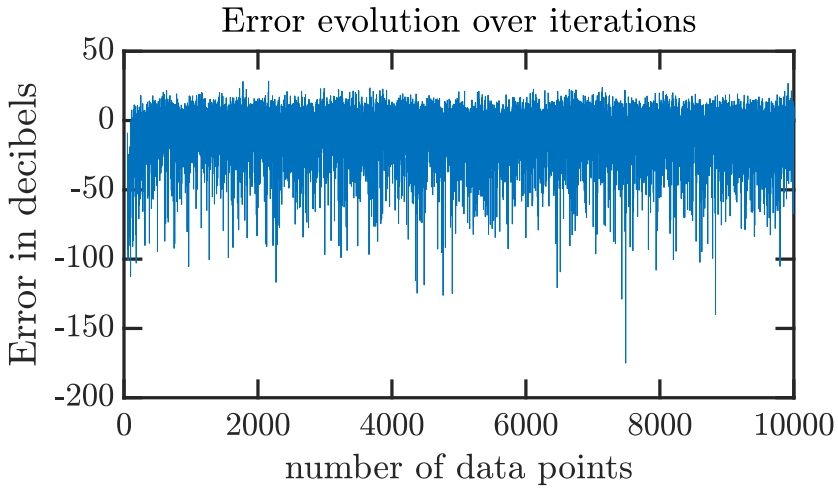

Figure 12: Error evolution over iterations for L=2

2.2 Frequency Response

Frequency of the actual system is plotted in Figure 13. Frequency response for the order 2,4, and 10 are shown in Figure 14,15, and 16 respectively. We observe that as the order increases the magnitude and phase of the response tend to match the desired frequency response as shown in Figure 13. With order 2, the magnitude and phase plots are not close to desired response. With order 4, both magnitude and phase plot tend to desired frequency response. With order 10, the response tend to match the desired response. Hence, by increasing number of order of the FIR filter, we can better match a FIR filter to an IIR Filter.

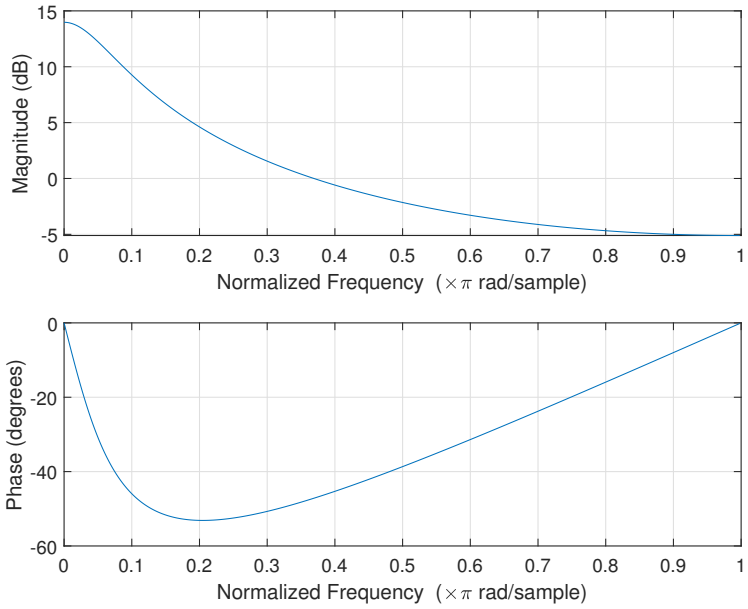

Figure 13: Desired Frequency response

2.3 Impulse Response

Similarly, the desired impulse response of the system is shown in Figure 17. The estimated impulse response for order 2,4, and 10 are plotted in Figure 18,19, and 20 respectively. We see that the number of samples overshooting magnitude 1 are less with order 2. Also, it subsides faster compared to order 4 and 10. As the order increases from 2 to 4 and 10, the number of initial samples overshooting magnitude 1 are more. Also, the time taken to reach initial state is also more for order 4 and 10.

Hence, there is a trade-off for selecting the system order. As we increase the order, the system gets a desired frequency response. However, the system tend to take more time to reach its initial state.

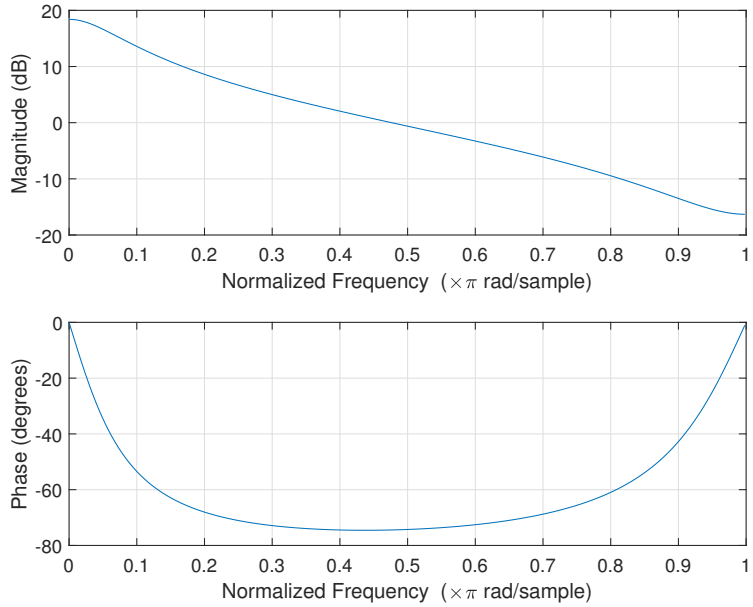

Figure 14: Estimated Frequency response for L=2

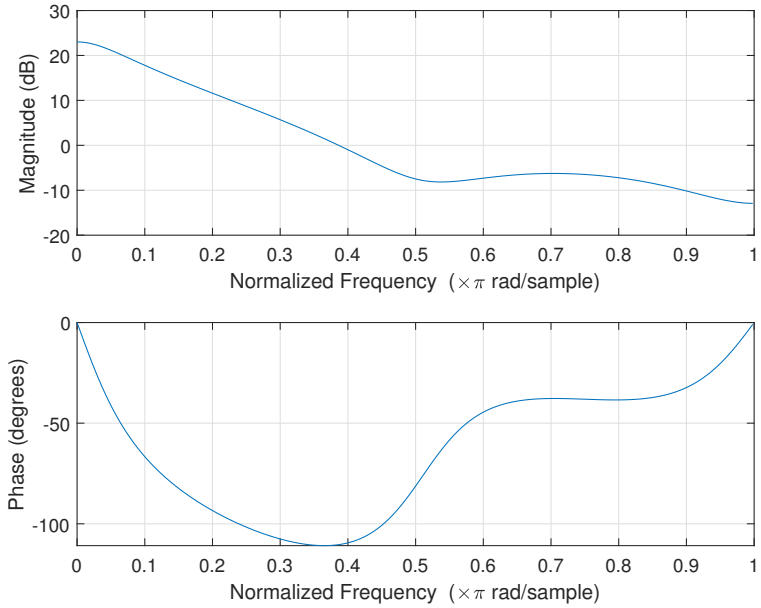

Figure 15: Estimated Frequency response for L=4

2.4 Output-error model

Ideally, the output-error model is not realistically feasible to implement using actual output signal y. Hence, the simulated filtered output in each loop is used and algorithm is adapted to estimate filter coefficients. The adapted model becomes equation error model. Assuming

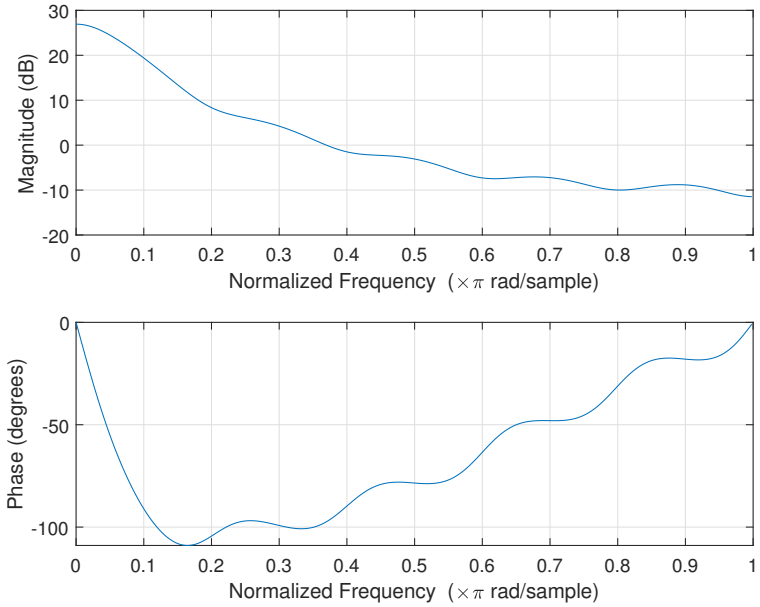

Figure 16: Estimated Frequency response for L=10

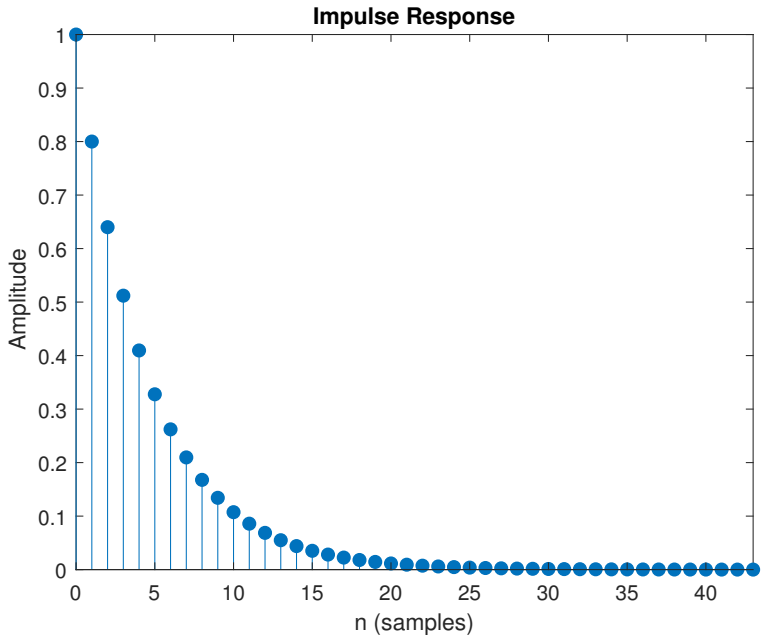

Figure 17: Desired impulse response

no order mismatch, considering first 4 coefficients of the system, the algorithm estimates filter coefficients perfectly for a step size of $\mu = 0.005$. The error evolution curve of the model is shown in Figure 21. We can see that the error converges from the error evolution curve. The error reaches to minimum -600 dB around 2300th Sample.

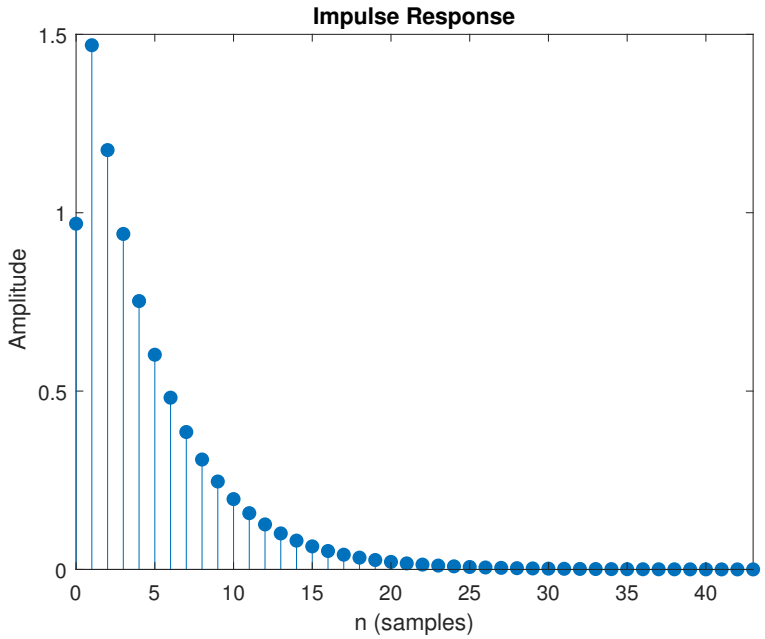

Figure 18: Estimated Impulse response for L=2

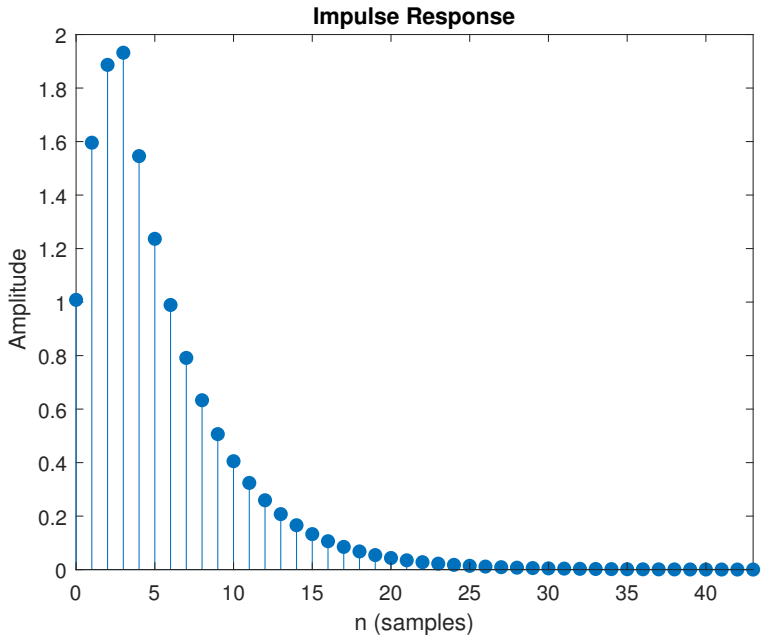

Figure 19: Estimated Impulse response for L=4

2.5 Simple hyper-stable adaptive recursive filter (SHARF)

SHARF is an approximate of equation-error model. It approximates the error gradients of Output error model to implement. The error evolution curve for SHARF is shown in Figure 22. We can see that error evolution curve reach a minimum of -600 dB around 1000th sample.

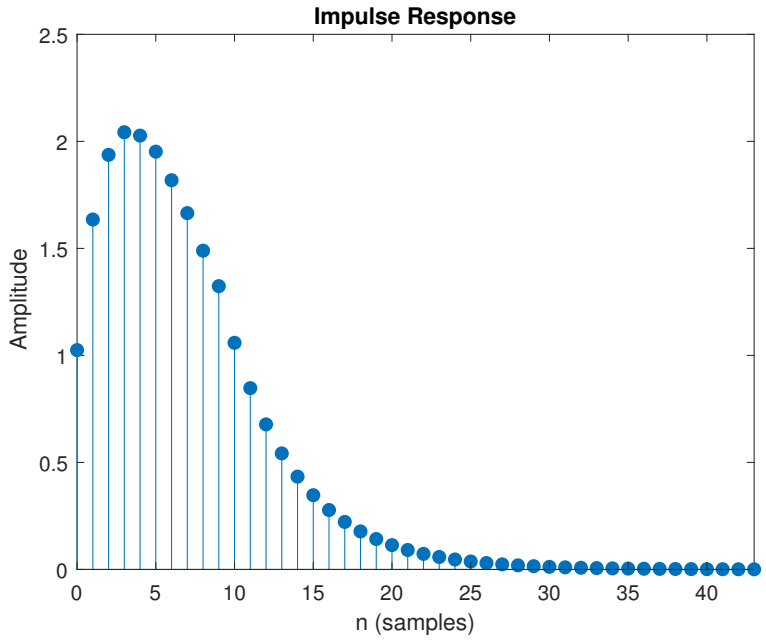

Figure 20: Estimated Impulse response for L=10

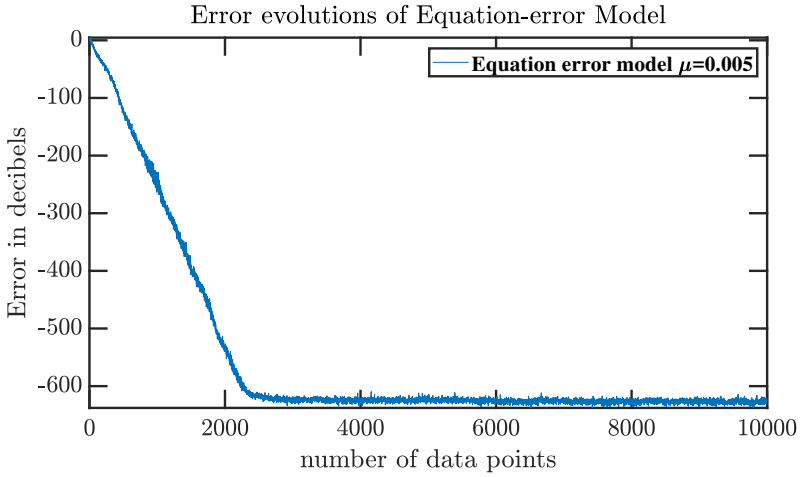

Figure 21: Error evolution of Equation error model

It converges faster compared to equation error model due to higher step size. Monte-Carlo of length 50 is used for averaging the error curves.

2.6 Levinson-Durbin Algorithm

Levinson-Durbin algorithm is used to estimate the filter coefficients. The algorithm estimates near perfect with the estimates. It estimates the filter coefficients in a single monte-carlo iteration. The Matlab code of the algorithm is attached in appendix for details.

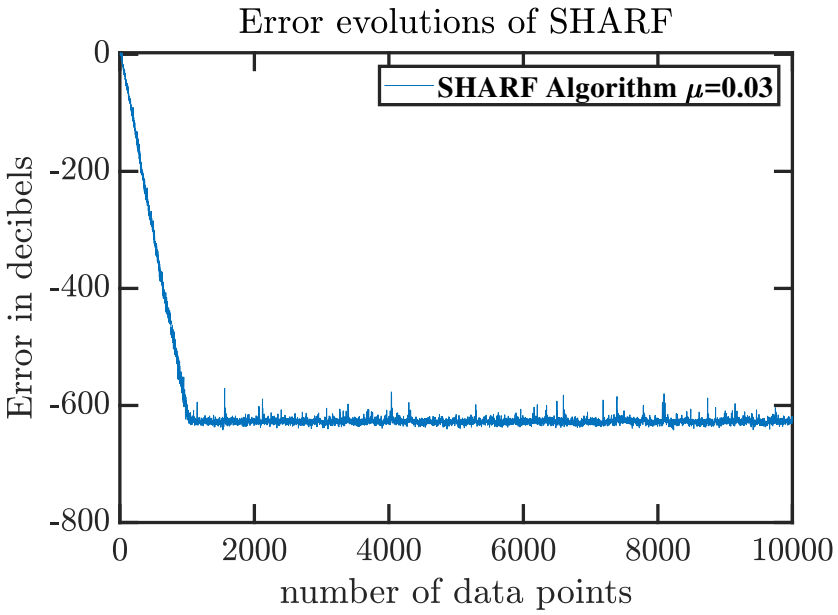

Figure 22: Error evolution of SHARF

3 Conclusions

Filter coefficients for an all pole system are attempted to estimate using LMS [1]. As LMS is not designed to estimate coefficient of IIR filter, the estimated coefficients are approximate estimates. From, frequency and impulse response analysis, we conclude that by adding large number of filter coefficients we can better fit FIR filter to IIR filter. Output Error model, SHARF, and Levinson-Durbin algorithms are designed to adapt FIR to IIR filter. Hence, these three algorithms estimate the filter coefficients accurately. The SHARF seems to be faster than Output error model. Levinson-Durbin seems to be fastest of all. All the codes for the algorithms are attached in appendix section.

References

[1] S. S. Haykin, *Adaptive filter theory*. Pearson Education India, 2005.

Appendix

Matlab Code for LMS

```
% LMS Example

mcN = 50;

N = 10000;

a =1;
b = [1 0.2 0 -0.8]; % true estimates


% bn = [1 -0.8];

b_hat = [0 0 0 0]';
s = length(b_hat);

mu = 0.05;

% Montecarlo
e = zeros(mcN,N);
for mc_loop = 1:mcN
    x = randn(N,1);
    % x = filter(bn,1,w)/100;

    d = filter(b,a,x);

    % LMS Iterations

    for LMS_loop = s:N
        x_e = x(LMS_loop:-1:LMS_loop-s+1);
        e(mc_loop,LMS_loop) = d(LMS_loop) - b_hat'*x_e;
        b_hat = b_hat + 2*mu*x_e*e(mc_loop,LMS_loop);
    end

    b_hat
%     box on
%     hold on
%     plot(s:N,db(e(s:N).^2))

end

MSE = mean(e(:,s:N).^2);
box on
hold on
plot(s:N,db(MSE));
```

Matlab Code For Newton LMS

```
% LMS Example

mcN = 50;

N = 10000;

a =1;
b = [1 0.2 0 -0.8];

% bn = [1 -0.8];

b_hat = [0 0 0 0]';
s = length(b_hat);

mu = 0.05;

% Montecarlo
e = zeros(mcN,N);
for mc_loop = 1:mcN
    x = randn(N,1);
    % x = filter(bn,1,w)/100;

    d = filter(b,a,x);

    % LMS Iterations

    for LMS_loop = s:N
        x_e = x(LMS_loop:-1:LMS_loop-s+1);
        e(mc_loop,LMS_loop) = d(LMS_loop) - b_hat'*x_e;
        b_hat = b_hat + 2*mu*eye(4)*x_e*e(mc_loop,LMS_loop);
    end

    b_hat
%     plot(s:N,db(e(s:N).^2))

end

MSE = mean(e(:,s:N).^2);
box on
hold on
plot(s:N,db(MSE))
```

Matlab Code For Normalized LMS

```
% LMS Example

mcN = 50; % monte carlo experiment  length

N = 10000;

a =1;
b = [1 0.2 0 -0.8]; % true estimates

b_hat = [0 0 0 0]'; %initial b estimates
s = length(b_hat);
Rxx=eye(s);
mu = 1; % step size
norm_mu=mu/((s+1)*trace(Rxx));

% Montecarlo
e = zeros(mcN,N);
for mc_loop = 1:mcN
    x = randn(N,1);
    d = filter(b,a,x);
%     [~,R]=corrmtx(x,10);
%     norm_mu=mu/((s+1)*trace(R));
    % LMS Iterations
    for LMS_loop = s:N
        x_e = x(LMS_loop:-1:LMS_loop-s+1);
        e(mc_loop,LMS_loop) = d(LMS_loop) - b_hat'*x_e;
        b_hat = b_hat + 2*norm_mu*x_e*e(mc_loop,LMS_loop);
    end
    b_hat
%     plot(s:N,db(e(s:N).^2))
end
MSE = mean(e(:,s:N).^2);
box on
hold on
plot(s:N,db(MSE))
```

Matlab Code For Block LMS

```
% Block LMS Example

mcN =50; % monte carlo experiment  length

N = 10000;

a =1;
b = [1 0.2 0 -0.8]; % true estimates
%% Block LMS
b_hat = [0 0 0 0]';
s = length(b_hat); %Number of new samples per iteration
M =4; %Block size
mu = 0.05;
% Montecarlo
e = zeros(M,N,mcN-1);
for mc_loop = 1:mcN
    b_hat = [0 0 0 0]';
    x = randn(N,1);
    d = filter(b,a,x);
    % LMS Iterations
    %for LMS_loop = (1+M/s):N/s-M
    for BLOCK_loop = s:N/M-s
        x_e = x(BLOCK_loop*s:1:BLOCK_loop*s+M-1);
        x_b = zeros(M,length(b_hat));
        for i=1:1:M
            for j=1:1:length(b_hat)
                x_b(i,j)=x(BLOCK_loop*s+i-j);
            end
        end
        e(:,BLOCK_loop,mc_loop) = d(BLOCK_loop*s:1:BLOCK_loop*s+M-1) - ...
            x_b*b_hat;
        b_hat = b_hat + 2*mu/M*x_b.'*e(:,BLOCK_loop,mc_loop);
    end
end

MSE = mean(mean(e(:,:,:).^2,3));
if mcN==1
    MSE = mean(e(:,:,mc_loop).^2);
end
box on
hold on
plot(s:N,db(MSE(s:N)))
```

Matlab Code For FDAF

```
% FDAF LMS Example

mcN =50; % monte carlo experiment  length

N = 10000;

a =1;
b = [1 0.2 0 -0.8 ]; % true estimates
B_hat = fft([0 0 0 0])'; %initial b estimates
s = length(b_hat);
M=4; %block size
padding=2*s;
mu = 0.01; % step size

% Montecarlo
e = zeros(s,N/M,mcN-1);
for mc_loop = 1:mcN
    b_hat = [0 0 0 0]';
    B_hat = fft(b_hat,padding);
    x = randn(N,1);
    d = filter(b,a,x);
    for BLOCK_loop = s:N/M-s
        X_e=diag(fft(x(BLOCK_loop*s-s:1:BLOCK_loop*s+s-1)));
        y = ifft(X_e*B_hat);
        e(:,BLOCK_loop,mc_loop) = ...
            d(BLOCK_loop*s:1:BLOCK_loop*s+s-1)-y(padding-s+1:padding);
        e_padded = zeros(1,padding)';
        e_padded(padding-s+1:padding) = e(:,BLOCK_loop,mc_loop);
        PHI = X_e'*fft(e_padded,padding);
        B_hat = B_hat + mu*PHI;
    end
    b_hat = ifft(B_hat);
    b_hat_array(:,mc_loop) = b_hat(1:s)
end
mean(b_hat_array,2)
MSE = mean(e(:,:,:).^2,3);
if mcN==1
    MSE = mean(e(:,:,mc_loop).^2);
end
plot(s:N/M,db(MSE(s:N/M)))
```

Matlab Code For LMS adapting to IIR

```
% LMS Example

mcN = 50; % monte carlo experiment  length

N = 10000;

b =1;
a = [1 -0.8]; % true estimates
% b_hat = [0 0]'; %initial b estimates L=2
% b_hat = [0 0 0 0]'; %initial b estimates L=4
b_hat = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]'; ...
    %initial b estimates L=10
s = length(b_hat);

mu = 0.01; % step size

% Montecarlo
e = zeros(mcN,N);
for mc_loop = 1:mcN
    x = randn(N,1);
    d = filter(b,a,x);
    % LMS Iterations
    for LMS_loop = s:N
        x_e = x(LMS_loop:-1:LMS_loop-s+1);
        e(mc_loop,LMS_loop) = d(LMS_loop) - b_hat'*x_e;
        b_hat = b_hat + 2*mu*x_e*e(mc_loop,LMS_loop);
    end
    b_hat
%     plot(s:N,db(e(s:N).^2))
end

MSE = mean(e(:,s:N).^2);

%  plot(s:N,db(MSE))

% d_hat=filter(b_hat,a,x);
% figure
% freqz(b,a);
% figure
% freqz(b_hat,a);
% figure
% impz(b,a);
figure
impz(b_hat,a);
```

Matlab Code For Output Error Model

```
%Equation Error Model
mcN = 50;
N = 10000;
a = [1 0.8 0.64 0.5120];
% a = [1 0.4 0.4*.4 .4*.4*.4];
b = 1';
c_hat = [0.1 0.1 0.1 0.1]';
s = length(c_hat);
mu = 0.005;
% Montecarlo
e = zeros(mcN,N);
for mc_loop = 1:mcN
    c_hat = [0.1 0.1 0.1 0.1]';
    x = randn(N,1);
    d = filter(b,a,x);
    beta_alpha=zeros(s,N);
    % LMS Iterations
    for LMS_loop = s+1:N-s-1
        u = zeros(1,s)';
        u(1) = x(LMS_loop);
        u(2:s) = -d(LMS_loop-1:-1:LMS_loop-s+1);
        e(mc_loop,LMS_loop) = d(LMS_loop) - u.'*c_hat;

        beta_alpha(1,LMS_loop) = -x(LMS_loop);
        beta_alpha(2:s,LMS_loop) = d(LMS_loop-1:-1:LMS_loop-s+1);

        for p=1:1:s-1
            beta_alpha(1,LMS_loop) = beta_alpha(1,LMS_loop) - ...
                c_hat(p+1)*beta_alpha(1,LMS_loop-p);
        end
        for z=2:1:s
            for p=1:1:s-1
                beta_alpha(z,LMS_loop) = beta_alpha(z,LMS_loop) - ...
                    c_hat(p+1)*beta_alpha(z,LMS_loop-p);
            end
        end

        del = 2*e(mc_loop,LMS_loop)*beta_alpha(:,LMS_loop);
        c_hat = c_hat - 2*mu*del;
    end

end
MSE = mean(e(:,s:N).^2);
if mcN==1
    MSE = e(:,s:N).^2;
end
plot(s:N,db(MSE))
```

Matlab Code For Levinson-Durbin Model

```
1
2  mcN = 1; % monte carlo experiment  length
3
4  N = 10000;
5
6  b =1;
7  a = [1 -0.8]; % true estimates
8  c_hat = [0 0 0 0]';
9  mu = 0.03;
10
11 s=3;
12 e = zeros(mcN,N);
13 for mc_loop = 1:mcN
14     x = randn(N,1);
15     d = filter(b,a,x);
16     r = autocorr(d);
17     y = zeros(1,s)';
18     y(1) = r(2);
19     beta = 1;
20     c = r(2);
21     g = r(2);
22     for k = 2:s
23         beta = (1-g*g)*beta;
24         g = (r(k+1) - c.'*r(2:k))/beta;
25         c = [g; c - g*c(k-1:-1:1)];
26         y(k) = g;
27     end
28     c_hat_array(:,mc_loop) = [1; -c(s:-1:1)];
29 end
30 mean(c_hat_array,2)
```