5. Optimal Filtering – Fixed and Adaptive

# 5.1 Weiner Filter

Weiner filtering is a method through which an estimate of a random process is derived by filtering a random process through a Weiner filter. In terms of system identification the essential idea is to pass a random process through an unknown system, and also through the Weiner filter and then calculate the difference between the two outputs. The Least Mean Square (LMS) error can then be calculated and used to adjust the Weiner filter coefficients to provide a more accurate estimate. When the outputs of the two systems (the unknown and the Weiner filter) are the same then an accurate model of the unknown system has been developed. The Weiner filter is essentially therefore an FIR filter whose coefficients or weights adapt over time to produce the desired output.

An example could be in weather prediction: the random process or system to be modelled is the weather system of the earth, the input is weather information from previous days, the output is tomorrow’s weather and the estimation is the estimation of tomorrow’s weather. The Weiner filter uses past data to predict future weather forecasts, and then is adjusted by finding the LMS error between the prediction of tomorrow’s data and the actual weather results (e.g. rainfall, humidity, cloud height etc.).

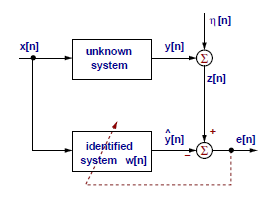
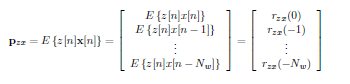
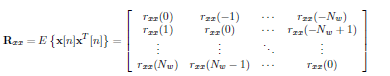


Figure - Diagram demonstrating the block diagram for system identification

## 5.1.1

As discussed in the lab hand out the optimum filter coefficients can be found given the matrix with autocorrelation coefficients of the input signal x, and the vector containing elements from the cross correlation of the signals x and z.

In Equation [1] Rxx is the autocorrelation matrix of x over a lag defined by the order of the target unknown system, and Pzx is a vector of autocorrelation positive lags over the same range.

The autocorrelation matrix was created by finding the autocorrelation of the whole of the input sequence x[n], and then taking up to Nw (the order of the unknown system) positive lags around the current value of n form Rxx using the ‘toeplitz’ command in Matlab. The cross correlation of z and x was found in the same way just without forming the matrix.

The reason we only consider the Nw positive lags is because of the nature of the process. Referring back to diagram 1:

Therefore our process only depends on the previous Nw inputs, so it would be unnecessary to calculate Rxx and Pzx across the whole of the autocorrelation. You might think that this is all very well as long as you know the order of the unknown system, but all that happens if you do calculate wopt for the entire autocorrelation and cross correlation is you end up with, as expected, all weightings that don’t actually exist being approximately zero. Therefore in this exercise only the autocorrelation up to Nw was used as we know the order of the system and it is far more computationally efficient than calculating for N = 1000.

In this exercise the ‘unknown’ filter was known and was the following:

The fundamental goal of this exercise is to consider how effective and what parameters affect the effectiveness, of the Weiner filter estimate of the unknown system.

The Matlab function for generating the optimum filter coefficients given is shown below:

function [ w\_opt ] = wiener\_est(x,z,Nw)

%Function calculates the optimum Weiner coefficients given the input signal

%and output signal of the process to be identified, and the estimated order

%of the unknown system.

temp1 = xcorr(z, x, Nw, 'unbiased');%Calculate the cross correlation of the input with output but only return the values Nw either side of lag 0

Pzx = temp1(Nw+1:2\*Nw+1);% Take only lags ranging from rxx(0) -> rxx(-Nw)

temp2 = xcorr(x,Nw, 'unbiased');%Calculate the autocorrelation of the input but only return the values Nw either side of lag 0

Rxx = toeplitz(temp2(Nw+1:-1:1));% Create Autocorrelation matrix

w\_opt = Rxx\Pzx;%Calculate optimum filter coefficients

end

## 5.1.2

When tested with noise at 0.1 standard deviation the system gave wopt = [0.99105, 1.77905, 3.22295, 1.61848, 0.736225] as opposed to the ideal coefficients of [1, 2, 3, 2, 1]. As can be seen the result is fairly similar, the discrepancy being due to the noise (strictly not true as explained in the next section) and to the inaccuracies of the estimated ACF over a sample size of 1000. By increasing the sample size of x[n] and by reducing the noise n[n] the Weiner estimate can be improved. In this first simulation the noise standard deviation was 0.1 so was reasonably low, in 5.1.3 we shall consider the results with larger noise standard deviations and generally investigate how the Weiner filter performs in varying noise environments.

## 5.1.3

Considering different noise powers by changing the standard deviation of the sequence n[n] allows us to see the performance of the Weiner filter in different noise environments. From the table in figure 2 it is clear that the Weiner estimate still performs well even under very noisy conditions. For example with a standard deviation of 8.35 (a noise power of 69.7225) the estimated filter coefficients are still appreciably close to the actual coefficients.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| SNR | Noise std. | w[1] | w[2] | w[3] | w[4] | w[5] |
| 32.90144 | 0.1 | 0.998003 | 2.008218 | 3.001679 | 2.001463 | 0.999875 |
| 8.576189 | 1.75 | 1.077196 | 2.029884 | 3.015601 | 2.011355 | 0.918543 |
| 4.286414 | 3.4 | 1.054139 | 2.02843 | 3.02961 | 1.968091 | 1.068665 |
| 2.653028 | 5.05 | 1.051319 | 1.937182 | 3.219109 | 2.185805 | 0.774133 |
| 1.556842 | 6.7 | 1.038399 | 2.124987 | 2.939866 | 2.207258 | 0.840974 |
| 0.971303 | 8.35 | 1.164391 | 1.573483 | 3.198484 | 1.862727 | 1.015586 |

Figure - Table detailing the coefficient estimates at different noise powers

The estimation for the coefficients seems to have a consistent level of accuracy despite varying noise levels, for example the estimate at an SNR of 0.97 is essentially as good as the one at an SNR of 33.The reason for the Weiner filters resistance to noise is due to the fact that it is computed through the cross correlation of z[n] and x[n]. The noise only effects signal z[n] (see figure 1), as z[n] =y[n] + n[n]. The noise signal n[n] is random and uncorrelated and independent of x[n] which leads to a useful result:

The autocorrelation of z[n] and x[n] is given that they are both zero mean processes:

Theoretically the noise and input x are independent, i.e. the bearing on 1 has no impact on the output of the other. As a result and because the noise is a Gaussian distribution with an expectation of 0:

Therefore the autocorrelation of z and x depends solely (theoretically) on the autocorrelation of x and y. As a result increasing the noise power should have no effect on the estimate Weiner filter weights as these are dependent on the autocorrelation of z and n which is independent of the noise. However this is naturally true only in a theoretical sense and in the real world none of these processes behave exactly as they should. Therefore x, y, and n are not really zero mean processes, especially at low sample sizes and as a result the coefficient estimations are not quite accurate. However this accuracy is completely independent of noise, it depends only on the sample size (i.e. how well each signal represents the process it was derived from). This can be best illustrated increasing the sample size to 200,000, increasing the standard deviation to 10 (noise power of 100), under these conditions the Weiner filter gave a near perfect estimate of the filter.

Also the effect of having a larger Nw was analysed. By increasing Nw to 10 the following Weiner filter coefficients were generated: w[n] = [0.99681, 1.99984, 3.00880, 2.00047, 0.99809, 0.00296, -0.00120, 0.00123, 0.00595, 0.00037, 0.00964]. As is clearly obvious filter weightings outside the ones that exist in reality are approximately zero and this makes sense as y[n] and therefore z[n] in an ideal world are only correlated to the current and previous four samples of x[n]. As a result the cross correlation of x and z is:

Therefore the cross correlation depends only on the lags up to the order of the known process. Theoretically all other Weiner filter coefficients should be zero, but due to the limited sample size providing a non-perfect representation of each process there is deviation. As we increase the sample size the signals better obey their theoretical statistics and the coefficients outside the actual order of the filter converge quickly to 0 while the actual coefficients converge to their true values. This can be easily observed by running a sample size in the order of a 100,000.

## 5.1.4

In terms of calculating the computational complexity of the Weiner solution:

* A cross correlation or auto correlation for a single lag takes for an N sample long sequence takes N multiplications and N+ 1 addition. To form Rxx and Pzx therefore takes (Nw +1) correlations at a single lag (as there are (Nw +1) correlations that need to be calculated). Therefore the total computational complexity to create Rxx and Pzx is 2Nw (2N +1) or O(NNw):
* To solve the Weiner solution equation:

Rxx is a Nw by Nw matrix, therefore inverting it requires O(Nw3). Multiplying a row of Rxx-1 by Pzx takes Nw multiplications followed by Nw - 1 additions, therefore to calculate the entire matrix vector product takes 2Nw2-Nw or O(Nw2). Therefore to Calculate the Weiner solution requires:

Total complexity is just the sum of these two complexities.

# 5.2 The Least Mean Square (LMS) Algorithm

## 5.2.1

The issue with the current method of deriving the Weiner filter coefficients is that it assumes stationarity. This means that in many real world systems which are typically non-stationary the method in proposed in 5.1 is not a feasible solution, and instead we must turn to adaptive filtering that can estimate the Wiener solution on a sample by sample basis. The Least Mean Square Algorithm adjusts the coefficients of the Weiner filter with each sample, and from sample to sample aims to reduce the LMS error. Mathematically:

Equation [1]

In this way the future coefficient value is the old value plus a ‘new’ term, effected by the error and the current input. The Matlab function that implements this LMS algorithm is shown below:

function [y\_est, e, weights\_mat] = lms(x, z, u, Nw)

%Summary: Function implements the Least Mean Square Algorithm

%Takes as arguments the input signal, the output signal corrupted by noise, the step size u,

%and the order of the unknown process (If unknown just set N = Nw).

%Calculates the estimate of the output by running the input signal through

%the weiner signal and then adjusting the parameters to reduce the MS

%error. Returns the the error vector across time, the estimate evolution

%across time, and the coefficients as they evolve in time (weights\_mat).

N = length(x);

numb\_coeffs = Nw + 1;

%Initialise variables

weights\_mat = zeros(numb\_coeffs, N-numb\_coeffs);

y\_est = zeros(N-numb\_coeffs, 1);

e = zeros(N-numb\_coeffs, 1);

for n = numb\_coeffs:N%Process input through Weiner filter. Note to take make an estimate calculation need first Nw+1 samples

x\_Nw = x(n:-1:n-Nw);%Take previous Nw samples to calculate the estimate

y\_est(n) = (weights\_mat(:,n-numb\_coeffs+1).')\*x\_Nw;%Calculate the output estimate

e(n) = z(n) - y\_est(n);%Calculate error

weights\_mat(:,n-numb\_coeffs+2) = weights\_mat(:,n-numb\_coeffs+1) + (u\*e(n)).\*x\_Nw;%Update coefficients estimate

end

end

## 5.2.2

The parameter ‘ is the step size or learning rate of the LMS algorithm. This determines how fast and accurately the error converges to its minimum value. If is too large then the error will overstep and oscillate around never converging properly to the point of minimum error. If is too small then the system responds very slowly and it will take a very long time for the system to converge to the minimum error. Therefore the choice of is a trade-off.

The evolution of coefficients in time for is shown below in figure 3:



Figure - Wiener Coefficient evolution in time

We can see that figure 3 converges as the mean square error decreases, and finally hits the actual value of the coefficient when the mean square error falls to zero.



Figure - Mean Square Error evolution in time

As can be observed from figure 3 all coefficients converge to their predicted value after around 300 samples. This is a reasonably slow convergence time; changing the value can speed this convergence time up at the cost of introducing inaccuracy and oscillation. Considering just coefficient 3 for:



Figure - Coefficient 3's evolution in time for different values of step size: 0.001 - blue, 0.005-green, 0.01-red, 0.05-yellow, 0.1 - pink

As can clearly be inferred from figure 5 the larger the value of the step size the faster the estimated coefficient converges to the true value, but also the more inaccurate. Looking at the pink trace corresponding to a step size of 0.1 we can see that it never truly settles on the true value but instead oscillates around it. If we make then the system for the coefficients becomes unstable and the estimated coefficients diverge. This can clearly be seen by taking the case of a step size of 0.5:



Figure - Graph shows the instability of the system for to large a step size

## 5.2.3

In terms of the complexity of the LMS algorithm, i.e. how many multiplications and additions are required per iteration, there are three main stages:

* Calculating the estimate of y:

This involves multiplying the current and 4 previous inputs by their respective coefficients and then summing them together. Therefore in the general case, if Nc is the number of coefficients, i.e. the order of the process plus 1, then this requires 2Nc – 1 as there are Nc multiplications and Nc -1 additions. This gives a result of order Nc, .

* Calculating the error:

The error is simply the current input minus the estimate of the input and therefore is simply a subtraction so 1 operation. This can be ignored in the large scale of things.

* Adjusting the weights:

Every loop of the equation the weights are adjusted according to the Equation[1]. Note the variables are in bold to indicate that they are vectors, and e[n] are scalars. Each of these vectors is Nc long. As a result there are Nc +1 multiplications and Nc additions, meaning an order of Nc.

In total therefore there are operations meaning a complexity of for every iteration of the loop, i.e. the complexity of the LMS algorithm is the same order as the number of coefficients of the filter. This is pretty quick and allows for a rapid data processing.

# 5.3 Gear Shifting

If the environment is statistically stationary then it can be beneficial to use a technique known as ‘gear shifting’. Gear shifting involves changing the value of the step size depending on how close to the point of minimum error, i.e. as the system approaches the point of minimum error then the step size decreases so that the error is more accurate around the point of convergence. This allows us to combine the advantages of a fast convergence time from a large step size with the accuracy of the convergence from a small step size. Note that this is only really effective for a stationary environment; if the input model changes then being able to really accurately converge to a particular coefficient value may not be helpful if the convergent value in question changes frequently in time.

We are considering a stationary environment in section 5.2.2 as the model we are trying to estimate is not changing in time, i.e. the coefficients we are trying to estimate are constant in time. By dynamically varying the value of the step size according to the error squared a faster and more accurate response should be able to be achieved.

The approach taken for dynamically varying the step size was to multiply the step size b a factor ‘alpha’ which varies linearly with the error squared and multiplies the step size by a certain percentage. For example if the error is large then alpha will multiply u by a factor of e.g. 5, while if the error is squared is small will multiply by 0.2.

Figure - Diagram illustrating the method by which the step size is altered through dynamic factor alpha

e2 min

e2 max

**Error Squared**

**α**

**α max**

max

**α min**

Diagram showing how the Dynamic Factor Alpha changes with the Error Squared

The limits are all chosen empirically by observing the best result from the output. Making the alpha range large means a faster convergence but more oscillations as the step size becomes more sensitive to any small changes in the error. The range for the error squared should ideally cover the principle range of errors present in the output so that the system is dynamic over the largest range of errors.

Writing out how the LMS algorithm handles the weightings mathematically:

The equation for the weight coefficients then is now:

From experimentation the best results were achieved with an alpha that varied between 0.2 and 5, for error squared values of 5 and 60. The code to implement the gear shifted LMS is shown below:

%LMS GEAR SHIFTING FUNCTION

function [y\_est, e, weights\_mat] = lms\_gearshift(x, z, u, Nw)

%Summary: Function implements the Least Mean Square Algorithm

%Takes as arguments the input signal, the output signal corrupted by noise, the step size u,

%and the order of the unknown process (If unknown just set N = Nw).

%Calculates the estimate of the output by running the input signal through

%the weiner signal and then adjusting the parameters to reduce the MS

%error. Returns the the error vector across time, the estimate evolution

%across time, and the coefficients as they evolve in time (weights\_mat).

%Also adjust the step size dynamically with the error squared so as to

%implement a small step size for small error and a large step size for

%large error.

c = 5;

%Initialise Boundaries

e\_squared\_max = 60;

e\_squared\_min = 5;

alpha\_max = c;

alpha\_min = (1/c);

N = length(x);

numb\_coeffs = Nw + 1;

%Initialise variables

weights\_mat = zeros(numb\_coeffs, N-numb\_coeffs);

y\_est = zeros(N-numb\_coeffs, 1);

e = zeros(N-numb\_coeffs, 1);

for n = numb\_coeffs:N%Process input through Weiner filter. Note to take make an estimate calculation need first Nw+1 samples

x\_Nw = x(n:-1:n-Nw);%Take previous Nw samples to calculate the estimate

y\_est(n) = (weights\_mat(:,n-numb\_coeffs+1).')\*x\_Nw;%Calculate the output estimate

e(n) = z(n) - y\_est(n);%Calculate error

e\_squared = e(n)\*e(n);

%Calculate alpha as a function of the error squared between to limits

if e\_squared > e\_squared\_max

alpha = alpha\_max;

elseif e\_squared\_min <= e\_squared <= e\_squared\_max

alpha = alpha\_max - e\_squared\*((alpha\_max - alpha\_min)/(e\_squared\_max - e\_squared\_min));

else

alpha = alpha\_min;

end

weights\_mat(:,n-numb\_coeffs+2) = weights\_mat(:,n-numb\_coeffs+1) + (alpha\*u\*e(n)).\*x\_Nw;%Update coefficients estimate

end

end

%Calculate alpha as a function of the error squared between to limits

if e\_squared > e\_squared\_max

alpha = alpha\_max;

elseif e\_squared\_min <= e\_squared <= e\_squared\_max

alpha = alpha\_max - e\_squared\*((alpha\_max - alpha\_min)/(e\_squared\_max - e\_squared\_min));

else

alpha = alpha\_min;

end

weights\_mat(:,n-numb\_coeffs+2) = weights\_mat(:,n-numb\_coeffs+1) + (alpha\*u\*e(n)).\*x\_Nw;%Update coefficients estimate

end

end

The improved performance of the gear shifted LMS over the standard LMS can clearly be seen below:



Figure - Comparison of the convergence characteristic between gear shifted and non-gear shifted LMS: red - standard LMS, blue - gear shifted LMS

As can be clearly observed gear shifting rapidly increases the convergence time without the cost of convergence inaccuracy which is present for larger constant step sizes. Note that as already discussed the slight ripple around the true value is because the large range of alpha makes the gear shifted LMS very sensitive to any change in error.

# 5.4 Identification of an AR Process

In this part of the lab an AR process generated values, and then an LMS algorithm is used to identify the coefficients of said AR process.

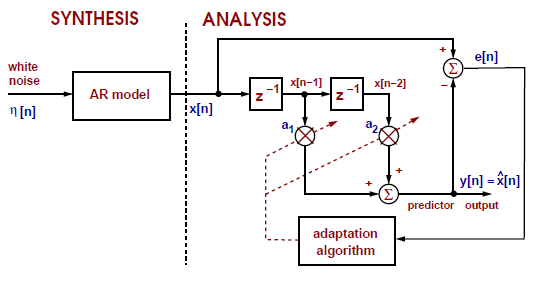


Figure - AR process identification block diagram

Looking at figure 7, the AR model we are trying to estimate acts as a process and produces a sequence of x[n]s. The adaptive filter then takes then takes this output from the AR and compares it with its own estimate. Note that this is estimating an AR process, so the output depends on previous outputs and a noise term. In this instance it is an AR(2) process so can be modelled as:

The adaptive filter uses the two previous outputs in the sequence **x** to calculate an estimate of x[n], y[n], and then checks the accuracy of its coefficients by calculating the error between the estimate of x[n] and x[n]. The adaption algorithm then uses Equation[1] to update its coefficients a1 and a2 so as to reduce the mean square error.

## 5.4.1

The function to carry out LMS to estimate an AR process is shown below:

function [y\_est, e, weights\_mat] = ARestimator(z, u, Nw)

%Summary: Function implements the Least Mean Square Algorithm for AR

%processes

%Takes as arguments the input signal, the step size u,

%and the order of the unknown AR process (If unknown just set N = Nw).

%Calculates the estimate of the output by running the input signal through

%the weiner signal and then adjusting the parameters to reduce the MS

%error. Returns the the error vector across time, the estimate evolution

%across time, and the coefficients as they evolve in time (weights\_mat).

N = length(z);

%Initialise Variables

weights\_mat = zeros(Nw, N-Nw);

y\_est = zeros(N-Nw, 1);

e = zeros(N-Nw, 1);

for n = Nw+1:N%Process input through Weiner filter. Note to take make an estimate calculation need first Nw samples

x\_Nw = z(n-1:-1:n-Nw);%Take previous Nw samples to calculate the estimate, Nw is the order of the AR filter

y\_est(n) = (weights\_mat(:,n-Nw).')\*x\_Nw;%Calculate the output estimate

e(n) = z(n) - y\_est(n);%Calculate error

weights\_mat(:,n-Nw+1) = weights\_mat(:,n-Nw) + (u\*e(n)).\*x\_Nw;%Update coefficients estimate

end

end

The filter was successfully implemented with both coefficients converging to -0.9 and -0.2 respectively. The reason that it converges to these values is because of the way the filter handles the input parameters.

Note that in the function above we use the ‘y =filter(b,a,x)’ command, which generates a difference equation of the following form:

For this particular code implementation, with a =[1, 0.9, 0.2], and b = 1, with x equal to output of the AR process then Matlab creates the following filter:

As a result the estimated filter coefficients naturally converge to -0.9 and -0.2 as the algorithm reduces the mean square error. A plot of the evolution of the estimated coefficients is shown below in figure 8 for a step size of 0.01.

## 5.4.2



Figure - Evolution of coefficient estimates for a step size of 0.01: red = a2, a1 = blue

As figure 8 shows the coefficient estimates converge after around a 1000 samples to their true values. However the convergence is not completely accurate, neither coefficient converges completely to the correct value but instead oscillates around them. This oscillation can be reduced by reducing the step size to 0.01, although this does massively elongate the convergence time as figure 9 demonstrates below:



Figure - Evolution of coefficient estimates for a step size of 0.001: red = a2, a1 = blue

Note that although a settling time of around 50,000 samples sounds very large, if you are sampling at 44.1 KHz then this is still only just over a second to converge to the correct coefficient values.

With a smaller step size the ripple around the true values becomes smoothed out, but naturally there is a cost in the time it takes the system to converge being now at around 50,000 samples. Considering just the first coefficient a1 and then a2 so as to be able to plot and compare multiple evolutions of ai for:



Figure - a1 coefficient estimation evolution in time for different values of step size

As is clear from figure 10 as the step size increases the speed of convergence improves but the error around the point of convergence becomes dramatically worse. With a step size of around 0.06 and above the system loses stability and no longer converges to the true filter coefficient values. The same is true for the evolution of the estimate of coefficient a2 as illustrated by figure 11 on the next page.



Figure – a2 coefficient estimation evolution in time for different values of step size

# 5.5 Speech Recognition

## 5.5.1

In this section we applied the same ideas as in the previous section but instead of applying the learning algorithm to a noisy process passed through a known AR model, samples of my own speech were used as processed. The adaptive algorithm would then try and adjust its weightings to model as accurately as possible the speech using an AR process.

1000 samples were taken from each letter recording, and then the MDL and AIC was calculated up to a model order of 25 for each speech sample to determine the best model order to try and model the speech with.



Figure - MDL and AIC for speech sample 'a': blue - MDL, red - AIC

**

Figure - MDL and AIC for speech sample 'e': blue - MDL, red – AIC



Figure - MDL and AIC for speech sample 's': blue - MDL, red – AIC



Figure - MDL and AIC for speech sample 't': blue - MDL, red – AIC



Figure - MDL and AIC for speech sample 'x': blue - MDL, red – AIC

Figures 12 – 16, suggest the following orders:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Spoken Letter | a | e | s | t | x |
| Best Estimated AR order | 8 | 6 | 3 | 7 | 2 |

Figure - Table showing best order for AR model of each spoken letter

This table details the estimates for the best AR orders for each spoken letter. In actual application we do not know which letter we are receiving, after all the whole point of this exercise is to identify speech. Therefore we need to apply the same order AR model to a continuous input and see how the AR coefficients converge over time to identify different letters. From table 17 we need to make make a compromise between probably over modelling some letters and under modelling others, therefore a compromise of 5-6 seems reasonable. The evolution over a 1000 samples of each vowel is plotted in the next 5 figures.



Figure - Evolution of coefficients of vowel "a" with a u = 0.03



Figure - Evolution of coefficients of vowel "t" with a u = 0.03



Figure - Evolution of coefficients of vowel "s" with a u = 0.03



Figure - Evolution of coefficients of vowel "t" with a u = 0.03



Figure - Evolution of coefficients of vowel "x" with a u = 0.03

As can be seen from figures 20 – 24 we can see that over a 1000 samples each set of vowel data causes the AR(5) model to converge to a particular set of coefficients allowing us to identify different vowels in speech by seeing how the AR model adapts. Note that unsurprisingly similar sounding vowels give similar convergence results, for example “a” and “e” have a very similar pattern, even if they don’t converge to exactly the same values. The issue with taking only 1000 samples of data is that over this range, and given the sampling rate of 44.1KHz, we are essentially only looking at a window of 1/44.1 = 0.0226ths of a second. Over this range at certain points in the speech some of the vowels will be pretty identical, for example taking the last 1000 samples of “e” and “t” will look pretty identical, while taking the end of “e” and the beginning of “t” then the two sets of coefficients will converge to completely different values.

Gear shifting only really helps in stationary environments. The vowel signals themselves are certainly not stationary but over a sample size of 1000 they are possibly locally stationary. Applying the algorithm described in part 3 for gear shifting we can compare the result with the non-gear shifted version and see if there is any improvement. However when attempting this, due to the non –stationary nature of the vowel signal, the error constantly jumped around causing the alpha to remain large and therefore consistently boosts the weights of the coefficients. This meant that the filter coefficients diverge as there is always some error and therefore the dynamic factor alpha is always increasing the weighting. As a result gear shifting was not adopted as a strategy for this section.



Figure - As can be seen, due to the non-stationary nature of the process the error never truly falls to zero, this causes problems for the gear shifting LMS

## 5.5.2

The quality of a predictor can be assessed using a parameter known as the ‘prediction gain’ which is defined below:

The prediction gain was calculated for each of the vowel predictors at 44.1KHz and 16KHz for a constant number of samples. To sample at 16KHz the signal taken at 44.1KHz was simply downsampled. This should improve the reliability of the comparison as the predictive gain is being calculated for the same signal and therefore should have the same error and convergence. The results are shown in the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Spoken Vowel | a | e | s | t | x |
| Predicted Gain for 44.1KHz sampling rate | 6.913774 | 8.79306 | 7.044943 | 7.774735 | 7.351587 |
| Predicted Gain for 16KHz sampling rate | 3.579034 | 7.011617 | 4.00957 | 6.737669 | 4.204872 |

Figure - Table of prediction gain for each vowel at different sampling frequencies

As can be observed from figure 26 the prediction gains are all worse for a lower sampling rate. This can be explained by considering that 1000 samples at 44.1KHz is 0.0226s of the vowel speech sequence, while at 16KHz a 1000 samples covers 0.0625s. Therefore the time over which we calculate the variance for the error and signal is approximately 3 times as large for 16KHz compared to the time for 44.1KHz. Speech is only stationary locally over a short time period, then for lower sample rates the 1000 samples is unlikely to be stationary. This means that the system coefficients may change over time and therefore the error between the system coefficients and the estimated coefficients may not necessarily reduce over time as we would expect with a stationary process.

To conclude; at 44.1KHz the 1000 sample sequence is more stationary than the 1000 sample sequence at 16.1KHz, hence the error power is larger in the sequence taken at 16.1KHz and therefore the predicted gains are worse. In summary over the same sample size the smaller the sampling frequency the lower the predicted gain and therefore the performance of the predictor. Therefore to get accurate convergence processing a very large number of data samples you need a very high sampling rate, if you have a low sampling rate then you can only achieve good convergence if you process a small number of data samples.

## 5.5.6

Signed LMS algorithms are simplified versions of the LMS whose advantage lies in their simplicity, making them very fast and easy to implement in hardware. The three signed LMS algorithms under consideration are described below:

The Matlab code used to implement these functions is pretty much the same as the function for standard LMS just includes the sign adaptations.

The performance of the signed LMS algorithms was compared with the slower but supposedly more accurate standard LMS by comparing the coefficient estimates for 4 different values of step size, for the AR(2) model in part 5.4 and the speech vowel sequences for “a” and “x”.

In terms of the AR(2) model in part 5.4 the following plots show the convergence of coefficients for different LMS models for µ values of 0.005, 0.01, 0.015, 0.02.



Figure - Evolution of coefficients of AR(2) system for u = 0.005



Figure - Evolution of coefficients of AR(2) system for u = 0.01



Figure - Evolution of coefficients of AR(2) model for u = 0.015



Figure - Evolution of coefficients of AR(2) model for u = 0.02

Figures 27 – 30 show that despite their simplicity the signed LMS algorithms converge just as accurately to the correct coefficient values as the standard LMS algorithm, although at a slightly slower rate. Observing figure 27 it is clear that the standard LMS converges faster than its signed counterparts, but considering larger step sizes there seems to be very little difference between standard and signed. In fact looking at the plots for the evolution of the error squared in time the standard LMS has no particular performance advantage over the others and is more complex.

Figure - The Error Squared evolution in time for different LMS algorithms



Figure 31 shows that the signed LMS algorithms give a pretty comparable error squared evolution in time compared to the standard LMS model. This seems strange as the signed algorithms have much cruder update mechanisms; the Signed -Signed LMS for example has a constant gradient which simply has the sign of the gradient reversed under certain conditions.

In terms of measuring the performance of against the spoken vowels “x” and “a” the signed LMS algorithms once again performed remarkably well in comparison to the standard LMS. The following four plots show the convergence for the first coefficient of vowel “a” over a 1000 samples, which from the previous section should converge to a value of 0.25.