Parallel implementations of adaptive filters

Heerkens, J.T.I.

Award date:
1994

7073

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Circuit Design Group (EEB)

Parallel Implementations of Adaptive Filters

Master Thesis of:
J.T.I. Heerkens

Master Thesis Report
Supervision: Prof. Dr. Ir. W.M.G. van Bokhoven
Dr. Ir. P.C.W. Sommen

Eindhoven, august 1993

Summary

In this thesis parallel implementations are investigated for a number of adaptive filters. By implementing the calculations parallel, faster calculation times can be achieved. However the main disadvantage of parallel implementations is that more hardware is needed.

The first algorithm that is investigated is the Least Mean Squares (LMS) algorithm. Because this algorithm is not suited to be implemented parallel, two algorithms have been developed. The first algorithm is the Delayed LMS (DLMS) algorithm. This algorithm has almost the same properties as the LMS algorithm, but introduces a small amount of processing delay. The second algorithm is the LMS with Delayed Weights (LMSDW). This algorithm has for a large adaptive constant α, a worse final misadjustment than the LMS algorithm. The advantage of this filter is that it does not introduce any processing delay.

Secondly, three types of the Recursive Least Squares (RLS) algorithm are investigated. These are: the Fast RLS (FRLS) or Fast Kalman implementation, the Givens based triangular systolic array, and the Least Squares Lattice (LSL) implementation. All these algorithms minimize the same squared error, but do this with different algorithms.

The third algorithm that is investigated is the Frequency Domain Adaptive Filter (FDAF) algorithm. Because this filter is not suited to be implemented parallel, a new algorithm is developed. First the Discrete Fourier Transformation (DFT) is changed to the Sliding window DFT (SDFT) and secondly the same technique, to overcome the summation problem, as in the DLMS algorithm is used. The new algorithm Delayed FDAF (DFDAF) is then investigated and implemented parallel.

Contents

1 Introduction

In many applications where filters are used, the external characteristics, such as echo path or transmission line characteristics, vary in time. If these characteristics change, also the optimal filter characteristics change. To overcome the problem of calculating the optimal filter every sample, adaptive filters are developed.

Adaptive filters can be applied to many different situations such as:

- Echo cancelation (e.g. acoustic echo's in a room),
- Noise cancelation (e.g. cancelling noise from the pilot's speech signal in the cockpit of an aircraft),
- Adaptive arrays (e.g. adaptive null steering),
- Signal correction (e.g. linear equalizer),
- Signal prediction (e.g. Linear Predictive Coding (LPC) of speech).

The behaviour of the adaptive filter depends on the update algorithm and the statistical properties of the input signal. If simple update algorithms are used (such as LMS) the behaviour of the filter is largely influenced by the statistical properties of the input signal. To overcome this problem more complex algorithms are developed. The disadvantage of these more complex algorithms is that the computational costs are higher.

When creating adaptive filters with a large number of coefficients the calculation time needed for the update algorithm can become too large in order to meet the required samplerate. One of the solutions to this problem is block processing. Block processing reduces the computational costs of the algorithm, so that less multiplications are necessary to calculate the filter weights and output signal.

In this report it is investigated if it is possible to do the necessary calculation parallel. Doing the calculation parallel does not decrease the number of needed multiplications. But, by doing multiplications parallel, it reduces the time needed to calculate the algorithm. To do as much as possible calculations parallel, it can be necessary to change the algorithm. If the algorithm is changed also the convergence properties of the adaptive filter change and have to be investigated.

2 Adaptive filters

In this report the signal estimator is used to describe and test the different types of adaptive filters. A short description of an example of such a system, namely an echo canceller, is given in figure 1. There are two different elements: the echo path (not part of the filter) and the adaptive filter. It is assumed that the echo path can be modelled with a Finite Impulse Response (FIR) filter. The adaptive filter consists of a filter part and an update part. The filter part is a FIR filter. The update part changes the filter weights each step towards the optimal filter response.

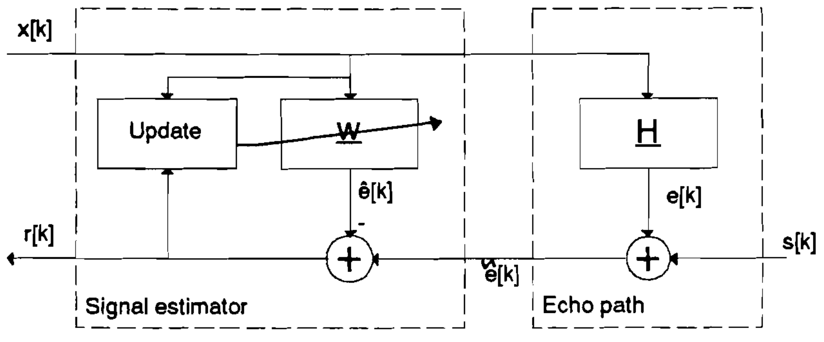


Figure 1: Signal estimator

The signals in figure 1 have the following meaning. Signal x is the input signal (e.g. acoustic signal of the speaker of a telephone) which through the (unknown) echo path H forms the error signal e. Signal e together with signal s (e.g. the speech of the user of the telephone) form the measurable signal ẽ. With these signals the adaptive filter makes an estimation ê of the echo signal e. When this estimation is calculated, the filter output r = ẽ-ê can be calculated. Notice that ideally ê = e, and thus r = s, which implies that the echo is completely cancelled.

In this report the following notations will be used:

x[k] input signal at time k
$\underline{x}^T[k]$ = (x[k], x[k-1], ..., x[k-N+1]) vector with last N samples
$w_i[k]$ ith weight coefficient of the adaptive filter at time k
$\underline{w}^T[k]$ = ($w_0[k]$, $w_1[k]$, ..., $w_{N-1}[k]$) vector with all weight coefficients
ê[k] estimated error signal
ẽ[k] combination of desired signal s[k] and error signal e[k]
e[k] error signal
s[k] desired signal
r[k] filter output (ideally r[k] = s[k])

2.1 Implementations

To implement adaptive filters there are basicly two possibilities. First adaptive filters can be implemented directly into hardware (VLSI structure). Because of the expensive design process this implementation is best suited, when a large number of filters have to be made. The second way to implement adaptive filters is to use Digital Signal Processors (DSP). DSP's are processors especially designed for the use in signal processing. Because the DSP's have to be programmed, DSP based systems form also a good platform to develop and test algorithm's, with low cost, because changes can be easily made by changing the program.

Because developing hardware (VLSI) is very expensive, it is desirable to have as much as possible redundancy in the design. This can be achieved if the implementations of all filter coefficients (or groups of successive coefficients) are equal so that only one filter coefficient (or one group of filter coefficients), and the interconnections between the coefficients (or groups of coefficients), have to be designed. A well known solution fore this is the systolic array. A systolic array consists of a number of Processing Entities (PE's) which are interconnected in a chain. Each PE only communicates with it preceding and succeeding PE. Therefore also the interconnections are very regular. Although not all filters can be implemented as systolic arrays, it is desirable to implement the filter with as much redundancy as possible.

When implementing a filter on DSP's, it is important that the work load can be equally divided over the processors. Furthermore the efficiency loss due to communication between the DSP's has to be minimized. If the algorithm is implemented as a systolic array this can be achieved by letting each processor do the calculations of a number of successive coefficients (PE's). In this way the communication between the processors is also minimized. In figure 2 a systolic array and an equivalent multiprocessor implementation is shown. Each processor does the calculations of two PE's.

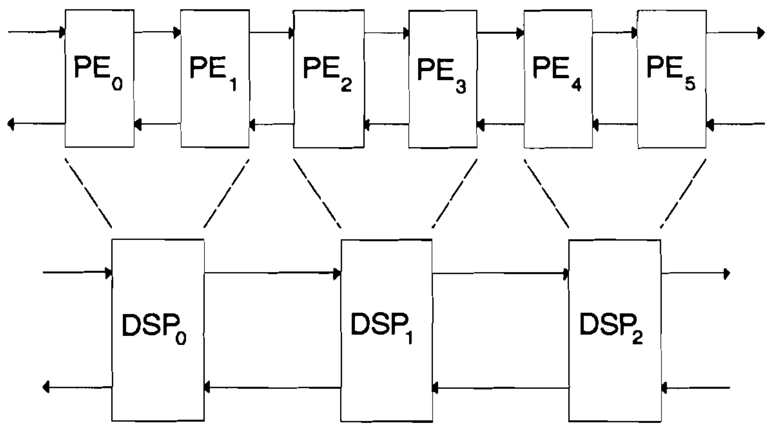


Figure 2: Systolic array and multiprocessor implementation

2.2 Characterizing quantities

To compare the different filtertypes and implementations, a number of quantities will be investigated. These quantities can be divided into two categories: filter behaviour and computational complexity.

The filter behaviour is described by two quantities: $J[\infty]$ and ν_{20}. $J[\infty]$ is the final misadjustment defined as: $J[\infty] = \frac{E\{(e[\infty]-\hat{e}[\infty])^2\}}{E\{s[\infty]^2\}}$ and ν_{20} (Rate of Convergence) is the number of samples necessary to decrease the initial misadjustment with 20 dB. This implies that a small $J[\infty]$ results in an accurate filterresponse while a small ν_{20} gives a fast adaptation of the filter.

The computational complexity of a filter can be described with many quantities. The first important quantity is the complexity order of the algorithm (denoted as C). This quantity gives the order of the number of multiplication needed to calculate one output sample. (Notice that parallel implementation of an algorithm does not change the order of the algorithm.) The second quantity is the processing delay (denoted as L). The processing delay of a filter gives the number of samples the output signal is delayed. Another important quantity is the minimum time that is needed between every sample to calculate the filterresponse. This quantity is denoted as ΔT and is expressed in the number of needed multiplication.

2.3 Signal types

To be able to compare the different filters and implementations, 4 types of signals will be used in the experiments which will be applied to the adaptive filters. These 4 signals are: White Noise, MA(1) (Moving Average), AR(1) (Auto Regressive) and a "stationary female unvoiced signal" (created in a synthetic way and denoted as Speech). Furthermore the powers of the signals are the same. In the next definitions of the used signal models n[k] is a white noise signal with $E\{n^2[k]\}=1/3$ and the Eigenvalue Ratio of the MA(1) and AR(1) signal equals 100. The E.R. of the SP signal equals 4,08. In figure 3 the power spectra of the signals are given.

White Noise signal model

$x[k] = n[k]$

Moving Average signal model of order 1

$x[k] = 0{,}7740 \cdot n[k] + 0{,}6332 \cdot n[k-1]$

Auto Regressive signal model of order 1

$x[k] = 0{,}5750 \cdot n[k] + 0{,}8182 \cdot x[k-1]$

Female unvoiced stationary signal model

$$\begin{aligned} x[k] = n[k] &+ 0{,}3190 \cdot x[k-1] - 0{,}02510 \cdot x[k-2] + 0{,}0271 \cdot x[k-3] \\ &+ 0{,}0046 \cdot x[k-4] - 0{,}0012 \cdot x[k-5] + 0{,}0001 \cdot x[k-6] \end{aligned}$$

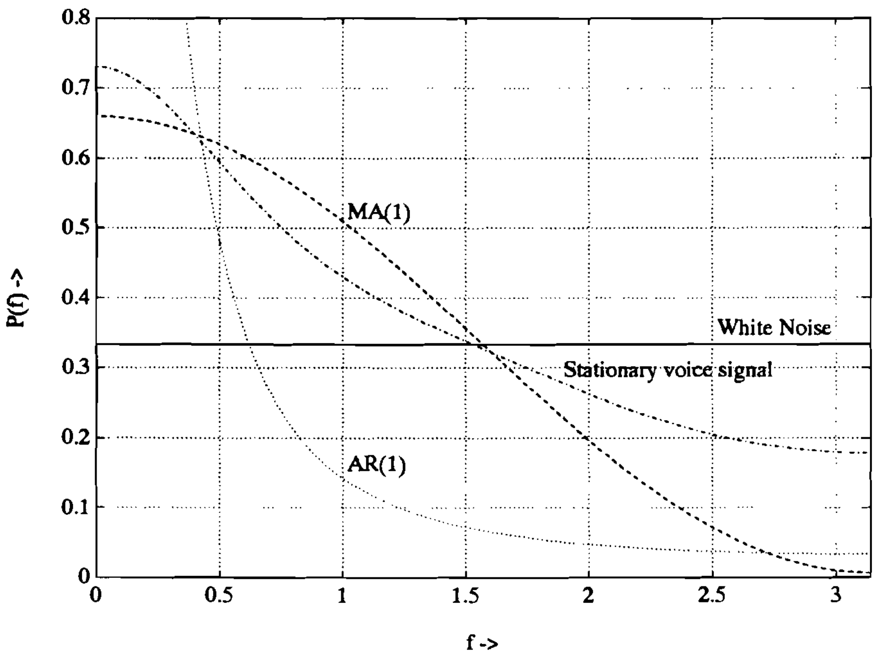


Figure 3: Spectra of different signals

3 Least Mean Squares Algorithm

One of the simplest update algorithms is the Least Mean Squares (LMS) algorithm. The LMS algorithm is popular because its simplicity of its computational complexity and the relative ease with which it can be analyzed. The LMS update algorithm is based on the steepest descent method, which is described by the following update expression:

$$\underline{w}[k+1] = \underline{w}[k] \overset{+}{-} 2\cdot\alpha\cdot E\{\underline{x}[k]\,r[k]\}$$

Because the calculating of $E\{\underline{x}[k]\,r[k]\}$ costs many computations, a common used method is to use the following approximation:

$$\underline{w}[k+1] = \underline{w}[k] + 2\cdot\alpha\cdot\underline{x}[k]\,r[k]$$

Besides the filter coefficients, the estimated signal $\hat{e}[k]$ and the filter output r[k] have also to be calculated. This is done with the following expressions:

$$\hat{e}[k] = \underline{x}^T[k]\cdot\underline{w}[k]$$
$$r[k] = \tilde{e}[k] - \hat{e}[k]$$

An outline of a filter using the LMS algorithm for the update of 4 coefficients is given in figure 4. In figure 5 the update part of the filter is given. Notice that r[k] is multiplied with 2α before it is used in the update part.

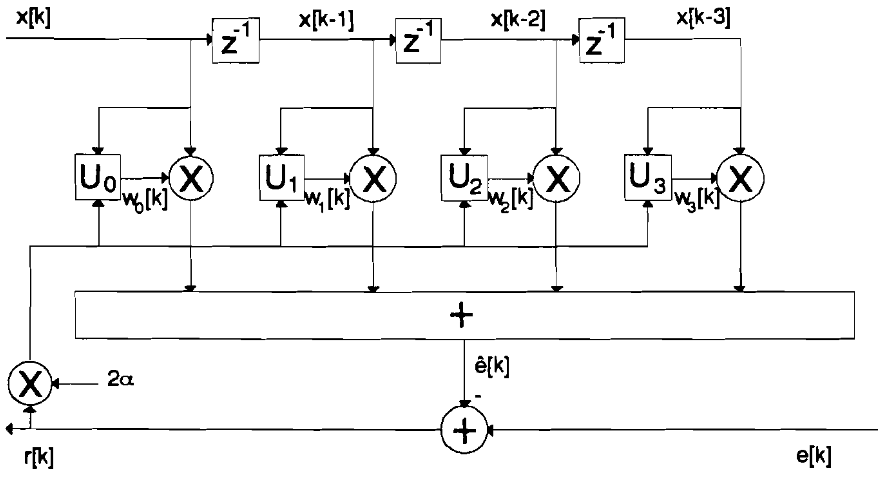


Figure 4: LMS adaptive filter

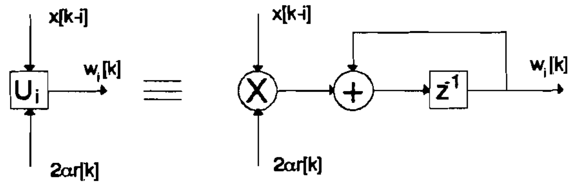


Figure 5: LMS update part

From literature (see Haykin[1]) it follows that, for white noise, the convergence properties of the LMS algorithm are given by the following equations:

$$J[\infty] = \frac{\alpha N \sigma_x^2}{1 - \alpha N \sigma_x^2}$$

$$\nu_{20} = \frac{-2}{{}^{10}\log(\,1 - 4\alpha\sigma_x^2\,)} \approx \frac{1.15}{\alpha\sigma_x^2}$$

Because of the simplicity of the LMS algorithm the computational costs are very low. Furthermore because r[k] has to be known to calculate the next filter coefficients, the processing delay is 0.

$C_{\text{LMS}} = 2N+1$
$L_{\text{LMS}} = 0$

The problem with the implementation of the LMS algorithm is the summation used in calculating ê[k]. If N is very large, the calculation time of this summation cannot be neglected in respect to the time needed for one multiplication. To overcome this problem, adapted LMS algorithms are investigated.

3.1 Delayed LMS algorithm

A possible solution of the summation problem is proposed by Meyer et.al.[2] and is generally referred to as the Delayed LMS (DLMS) algorithm. In the DLMS algorithm the calculation of the summation is delayed, so there is more time to calculate the signal ê[k]. Because of this delay in the calculation of r[k], it is necessary to approximate E{$\underline{x}$[k]r[k]} by a delayed version of $\underline{x}$[k]r[k] ($\underline{x}$[k-D]r[k-D]). By doing this the behaviour of the algorithm changes. The result of this change is that the output and the update of the weights of the filter is delayed (with D samples). The new DLMS algorithm is described by the following formulas:

$$\hat{e}[k-D] = \underline{x}^T[k-D]\cdot\underline{w}[k-D]$$
$$r[k-D] = \tilde{e}[k-D] - \hat{e}[k-D]$$
$$\underline{w}[k+1] = \underline{w}[k] + 2\cdot\alpha\cdot\underline{x}[k-D]\cdot r[k-D]$$

An outline of a filter using the DLMS algorithm for the update of 4 coefficients is given in figure 6. The update part is the same as in the LMS implementation (see figure 5).

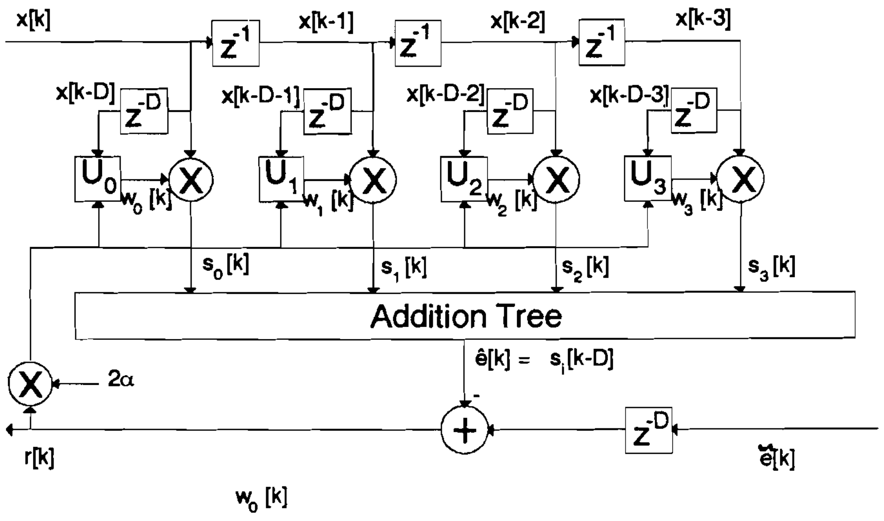
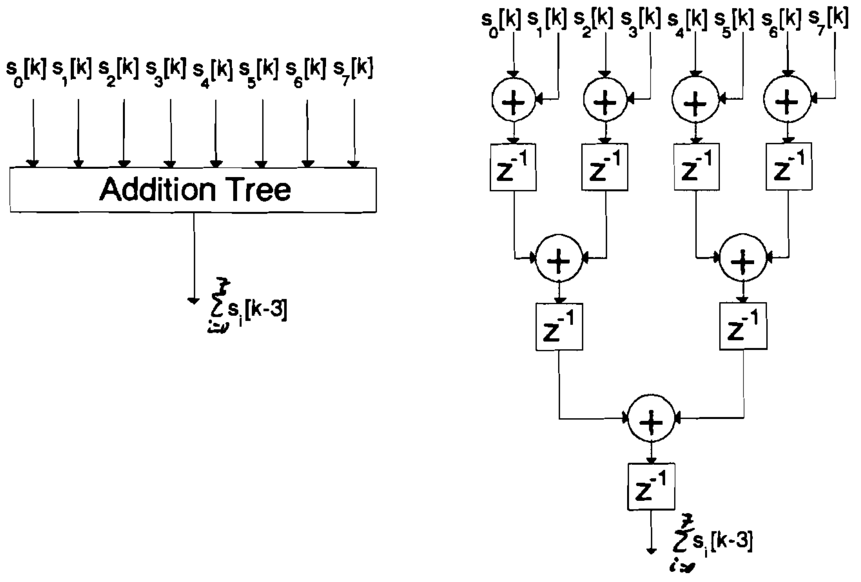


Figure 6: Delayed LMS filter

Because of the changes in the algorithm in respect to the LMS algorithm, different convergence properties apply for white noise (see Long et.al.[3]):

$$J[\infty] = \frac{\alpha N \sigma_x^2}{1 - \alpha(N+2D)\sigma_x^2}$$
$$v_{20} = \frac{-2}{{}^{10}\log(1 - 4\alpha\sigma_x^2)} \approx \frac{1.15}{\alpha\sigma_x^2} \quad (\textit{valid after } k>D)$$

Looking at these quantities it can be seen that if 2D«N the response of the Delayed LMS algorithm is almost equal to the response of the LMS algorithm. If N is large this can be achieved with an addition tree which delays the subresults. In figure 7 the outline of an addition tree is given. In this example a binary addition tree to add 8 signals is used as proposed by Long et.al.[3]. The delay introduced by this binary addition tree is ${}^{2}\log(N)$. And if N is large it follows that $2\cdot D = 2\cdot{}^{2}\log(N) \ll N$.



$s_0[k]$ $s_1[k]$ $s_2[k]$ $s_3[k]$ $s_4[k]$ $s_5[k]$ $s_6[k]$ $s_7[k]$
Addition Tree
$\sum_{i=0}^{7} s_i[k-3]$
z^{-1}

Figure 7: Addition tree

In this addition tree the time to add M signals can be up to the time for one multiplication. Although in the above example 2 signals are added at each addition, it is often possible to add more than 2 signals in this time. If it is possible to add M signals in this time, the introduced delay by the addition tree is ${}^{M}\log(N)$. To keep the delay as small as possible M has to be chosen as large as possible.

When implementing the filter parallel it is seen that all multiplications in the convolution part of the filter and the multiplication of $2\alpha\tau[k]$ can be done parallel. Furthermore all multiplications in the update parts can be done parallel. Because the update parts need the signal $2\alpha\tau[k]$ the multiplications in the update parts can not be calculated at the same time as $2\alpha\tau[k]$. This results in the following quantities:

$$C_{\text{DLMS}} = 2N+1$$
$$L_{\text{DLMS}} = D = {}^{M}\log(N)$$
$$\Delta T_{\text{DLMS}} = 2$$

Comparing these quantities with the LMS algorithm it can be seen that the same number of multiplications is needed but that, because most of them can be done parallel, the calculation time becomes very small and independent of the number of coefficients. A disadvantage of the DLMS algorithm is that it introduces a processing delay which grows logarithmical with the number of coefficients.

3.2 LMS with Delayed Weights

Another possible solution of the summation problem is proposed by Chester et.al.[4] and is referred to as LMS with Delayed Weights (LMSDW). In this solution it is supposed that the weights vary very slowly in time, so over a small number of samples the weights can be treated as constants. By doing so the convolution of $\underline{x}^T[k]\cdot\underline{w}[k]$ can be implemented as in figure 8. By implementing the convolution in this way no summation problem exists any more.

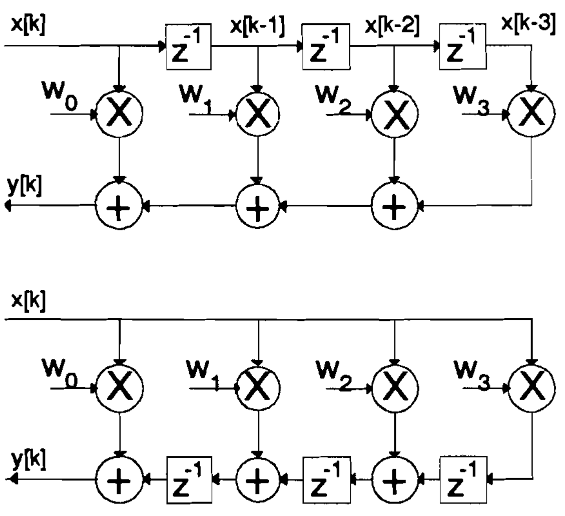


Figure 8: Filter convolution part

Because the weights are not constant in time this results in the following formulas describing the LMSDW algorithm:

$$\hat{e}[k] = \sum_{i=0}^{N-1} x[k-i] \cdot w_i[k-i]$$
$$r[k] = \tilde{e}[k] - \hat{e}[k]$$
$$\underline{w}[k+1] = \underline{w}[k] + 2\alpha \underline{x}[k] r[k]$$

An outline of a filter using the LMSDW algorithm for the update of 4 coefficients is given in figure 9. The update part is the same as in the LMS implementation (see figure 5).

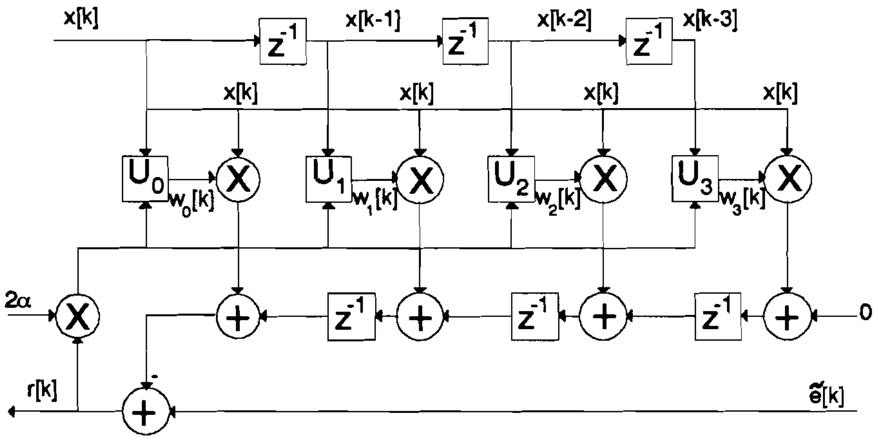


Figure 9: LMS with Delayed Weights filter

Because of the changes in the algorithm in respect to the LMS algorithm, different final misadjustment and rate of convergence apply for white noise:

$$J[\infty] = \frac{\alpha N\sigma_x^2}{1 - 3\alpha N\sigma_x^2}$$

$$\nu_{20} = \frac{-2}{{}^{10}\log(\,1 - 4\alpha\sigma_x^2\,)} \approx \frac{1.15}{\alpha\sigma_x^2}$$

Looking at these quantities it can be seen that the final misadjustment of the LMSDW filter is always worse in respect to the LMS algorithm. But for small α the difference between this filter and the LMS filter becomes smaller.

An efficient systolic array implementation of the LMS with delayed weight coefficients algorithm is proposed by Chester et.al.[4]. An outline of this implementation is given in figure 9. The implementation exists of N Processing Entities (PE's). Each PE does the calculations necessary for one weight coefficient. Because all PE's are equal, it is very easy to make a filter with more coefficients just by adding PE's at the end of the array.

When implementing the filter parallel it is seen that all multiplications in the convolution part can be done parallel. Furthermore all multiplications in the update parts can be done parallel. Because the update parts need the signal $2\alpha\text{r}[k]$ the multiplications in the update parts can not be calculated at the same time as $2\alpha\text{r}[k]$. And because the calulation of $2\alpha\text{r}[k]$ must be done after the calculation of the convolution part, these multiplications can not be done parallel. This results in the following quantities:

$C_{\text{LMSDW}} = 2N+1$
$L_{\text{LMSDW}} = 0$
$\Delta T_{\text{LMSDW}} = 3$

The advantage of this implementation is that it has a very fast calculation time which is independent of the number of coefficients and there is no processing delay. The disadvantage is that the final misadjustment, for large α, is worse than that of the LMS algorithm.

3.3 Experiments

In the first experiment the final misadjustment of the DLMS algorithm is investigated with respect to the processing delay of the filter. This is done for white noise ($\sigma_x^2 = 1/3$) and a filter length N=1024. The processing delay of the filter has a maximum of $L < {}^2\log(N) = 10$. In the experiment the echopath is an exponentially decaying impulse response of length N with a factor of 0.75. The results of this experiment can be found in figure 10. As expected the final misadjustment does not differ much if $2D \ll N$.

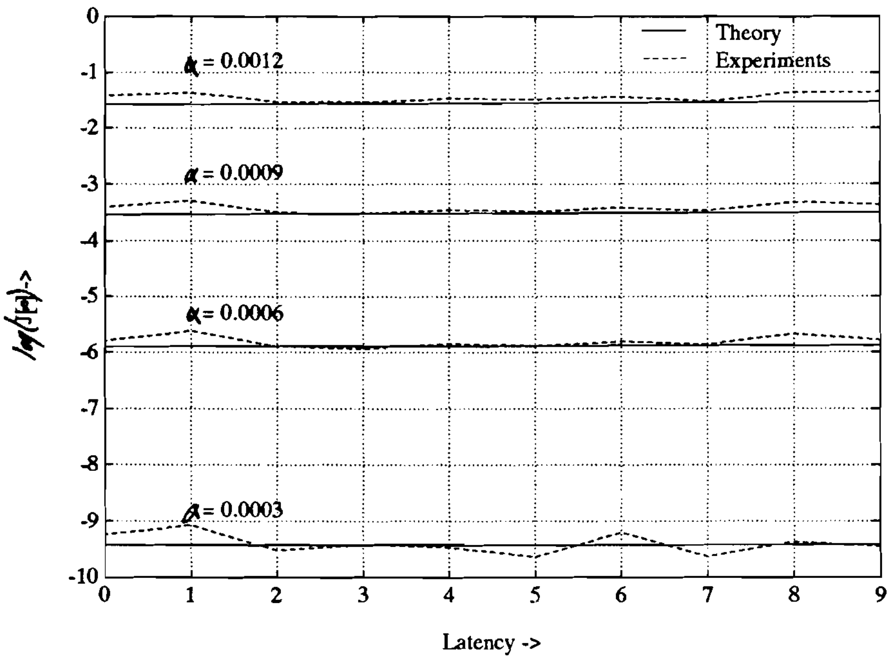


Figure 10: Final misadjustment of DLMS

In the second experiment the final misadjustments of a LMS, a DLMS and a LMSDW filter are investigated for different α. All filters have 1024 coefficients and for the DLMS filter L=10. Furthermore white noise is used as described in §2.3. The result can be found in figure 11. The DLMS algorithm has, as can be seen, almost the same final misadjustment as the LMS algorithm. The LMSDW algorithm has, however, a worse final misadjustment especially for large α.

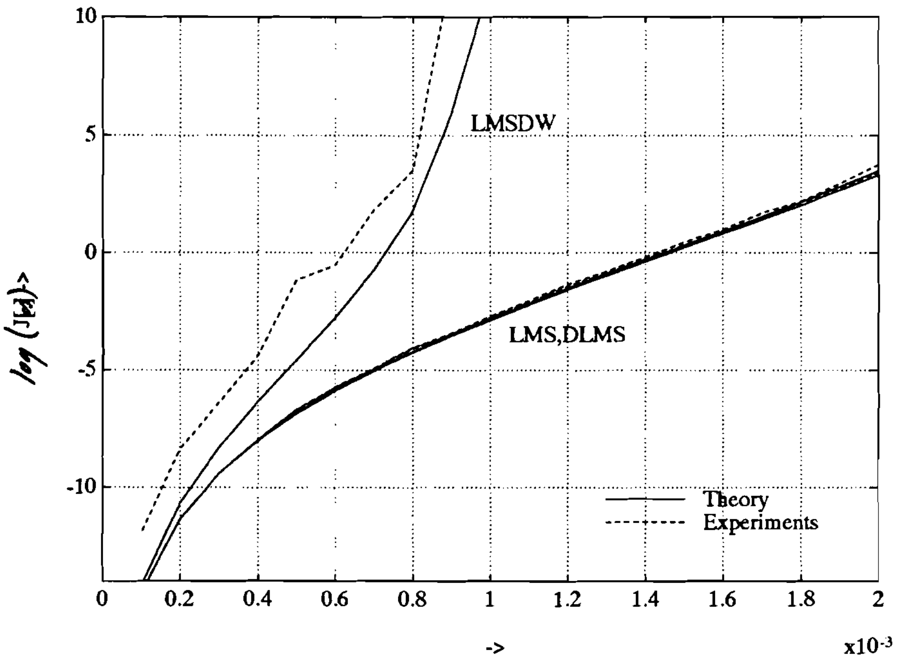


Figure 11: Final misadjustment for different α

In the last experiment a LMS, a DLMS and a LMSDW filter are tested with the four types of signals described in §2.3. All filters are of length N=1024 and α=0.0001. The Delayed LMS filter has a processing delay of L = ^{2}log(N) = 10. The results of the filter responses can be found in figure 12. The filter response of the DLMS algorithm is, as can be seen, almost equal to the response of the LMS algorithm. The LMSDW algorithm has the same initial convergence speed, bud has a worse final misadjustment than the LMS algorithm. Notice that the statistical properties of the signal does influence the convergence properties of the filters.

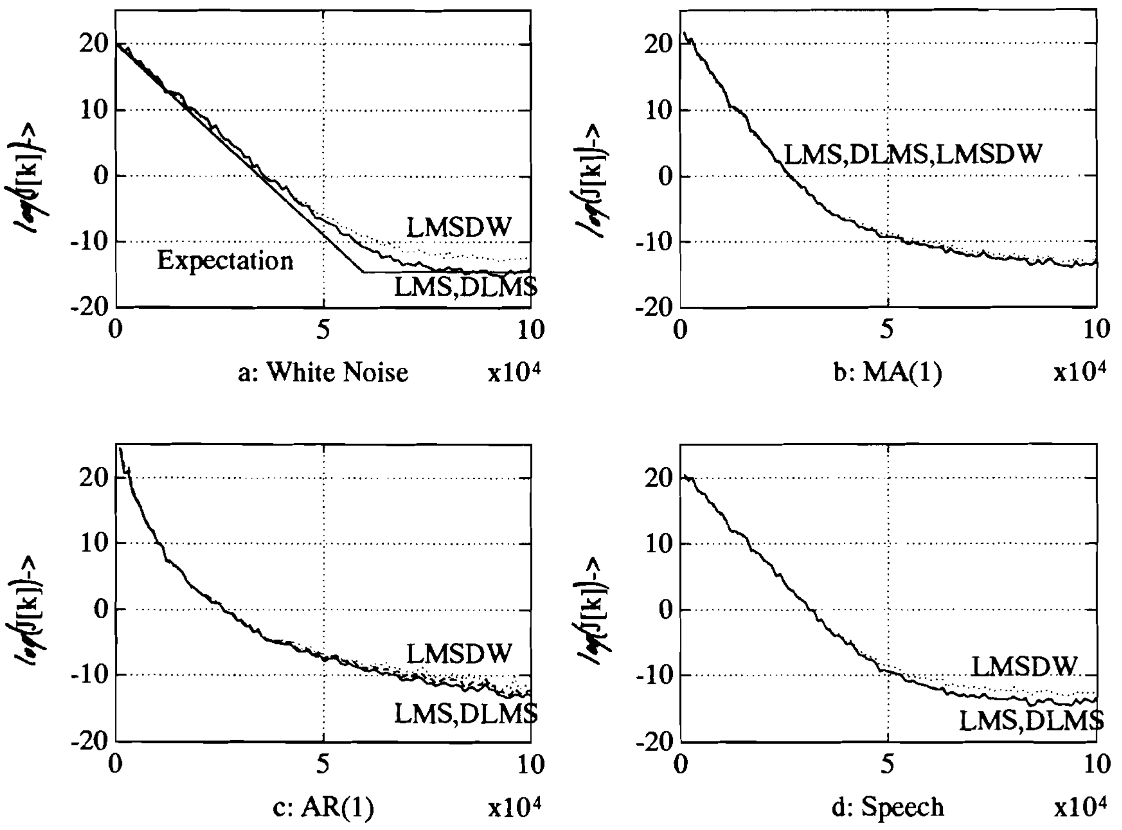


Figure 12: Experiment with different filters and signals

3.4 Conclusions

In this chapter two adapted LMS algorithms are investigated. To compare the different algorithms, the most important characteristics of the 3 algorithms are given in table 1. As can be seen all algorithms need the same number of multiplications but, because some multiplications can be done parallel in the adapted LMS algorithms, the calculation times become smaller. Furthermore all Rates of Convergences are equal.

Table 1: Summary of LMS, DLMS and LMSDW characteristics

Filter Type	$J[\infty]$	ν_{20}	C	L	ΔT
LMS	$\frac{\alpha N\sigma_x^2}{1 - \alpha N\sigma_x^2}$	$\frac{-2}{{}^{10}\log(1 - 4\alpha\sigma_x^2)}$	2N+1	0	2N (No parallel calculations used)
DLMS	$\frac{\alpha N\sigma_x^2}{1 - \alpha(N+2D)\sigma_x^2}$	$\frac{-2}{{}^{10}\log(1 - 4\alpha\sigma_x^2)}$	2N+1	D	2
LMSDW	$\frac{\alpha N\sigma_x^2}{1 - 3\alpha N\sigma_x^2}$	$\frac{-2}{{}^{10}\log(1 - 4\alpha\sigma_x^2)}$	2N+1	0	3

The first algorithm, DLMS, has almost the same behaviour as the LMS algorithm for $2D \ll N$, which can be achieved for large N and an addition tree. Because most multiplications can be done parallel the calculation time becomes very small (ΔT_{DLMS} = 2 multiplications). The disadvantage of this algorithm is that, because of the delay in the calculation of the estimated signal $\hat{e}[k]$, there is a processing delay.

The second algorithm, LMSDW, has for large α a worse behaviour than the LMS algorithm. Only for small α the behaviour is almost the same as the behaviour of the LMS algorithm. The advantage of this algorithm is the pipelined implementation and a very small calculation time ($\Delta T_{\mathrm{LMSDW}}$= 3 multiplications). Furthermore this implementation does not introduce any processing delay.

4 Recursive Least Squares Algorithm

The RLS algorithm is an algorithm from which the convergence properties are independent of the input signal statistical properties. This is done by minimizing the exponential weighted sum of squared errors usualy defined as $\varepsilon[k] = \sum_{i=0}^{k} \gamma^{k-i}(\tilde{e}[i] - \underline{w}^T[k]\underline{x}[i])^2$. This can be achieved by multiplying the update part of the LMS algorithm with the inverse autocorrelation matrix R_x^{-1}. By doing so the convergence properties of the algorithm becomes independent of the statistical properties of the signal. This results in the following update formula:

$$\underline{w}[k+1] = \underline{w}[k] + 2\alpha R_x^{-1}[k]\underline{x}[k]r[k]$$

If the inverse autocorrelation matrix is calculated in the normal way this leads to an algorithm with complexity $O(N^3)$. If the matrix inversion lemma is used the order becomes $O(N^2)$. It is however not necessary to calculate $R_x^{-1}[k]$ because only the filter output has to be explicitly known, and therefore RLS algorithms with $O(N)$ are possible (see §4.1, §4.2 and §4.3).

In this chapter three different types of implementations of the RLS algorithm are given. Because all implementations minimize the same squared error, the final misadjustment and convergence properties are the same and are not investigated here (see Haykin[7]). The RLS algorithm (with $\alpha=1/2$) results in the following misadjustment and final misadjustment (see Bellanger[6]).

$$J[k]\big|_{k>N} = N \cdot \frac{1-\gamma}{1+\gamma} \cdot \frac{1+\gamma^{k-N}}{1-\gamma^{k-N}} \quad \textit{whith } \gamma = \textit{weight factor}$$

$$J[\infty] = N \cdot \frac{1-\gamma}{1+\gamma}$$

4.1 The Fast Kalman Algorithm

One well known group of implementations is the fast Kalman algorithms. The Kalman algorithms (also known as Fast RLS) makes use of the fact that the input data vector is a shifted version of the previous input data vector with one new sample. By doing this, it is possible to reduce the complexity of the algorithm from $O(N^2)$ to $O(N)$.

The FRLS algorithm exists of 4 separate filters, labelled 1 trough 4, which share a common set of inputs. Filters 1 and 2 perform forward and backward linear prediction on the input data. Filter 3 calculates the gain factor of the RLS algorithm using the forward and backward linear prediction of filters 1 and 2. Filter 4 is the adaptive filter which uses the gain factor of filter 3 to calculate the filter response. An outline of the FRLS filter is given in figure 13. In appendix A a more extensive explanation of the 4 filters is given.

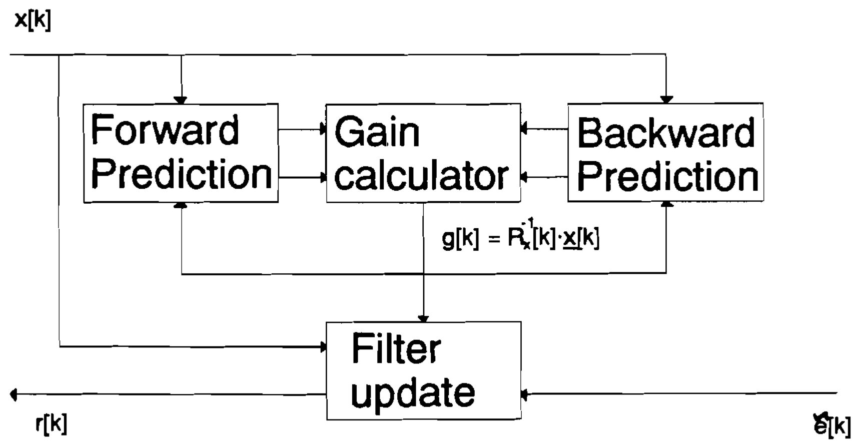


Figure 13: Fast RLS implementation

One of the problems of the FRLS algorithm is, that it can become numerical unstable. Because of this it is necessary to use safeguards to stabilize the algorithm. The problem of stabilizing the FRLS algorithm can be solved in different ways and is well described in literature (see e.g. Botto et.al.[5]).

The FRLS algorithm used is described by Bellanger[6]. The order of the update algorithm is C = 8N and together with the time needed to calculate the filter response this results in C = 10N+6. Although faster FRLS algorithms are known C = 7N, this algorithm is used because its numerical stability is better.

When implementing this filter parallel it can be seen that forward and backward prediction error subfilters can be calculated parallel. This can be done in $\Delta T = 2 + {}^{M}\log(N)$. The calculation of these two subfilters and the other two subfilters (the gain calculator $\Delta T = 5 + M/N$ and the filter update $\Delta T = 1 + {}^{M}\log(N)$) can not be done parallel. (These calculation times are discussed in appendix A). This results in the following quantities:

$$C_{FRLS} = 10N+6$$
$$L_{FRLS} = 0$$
$$\Delta T_{FRLS} = 8 + N/M + 2\cdot{}^{M}\log(N) \leq 8 + 1{,}5N$$

Although the calculation time (ΔT) is very fast and there is no processing delay, the implementation has a very complex structure. Furthermore the Kalman filters suffer from numerical instability problems.

4.2 Givens rotation based RLS using triangular systolic array

Due to their superior numerical stability and accuracy, RLS algorithms based on Givens rotations have received much attention recently. It has been shown that these Givens rotation based algorithms can be implemented using a triangular systolic array (See Haykin[7], McWhirther[8], Ling[9] and Yang et.al.[10]). This implementation does not impose any restrictions on the input data structure.
In figure 14a an implementation of a filter with 4 coefficients is given. The working of the filter is extensively explained in appendix B.

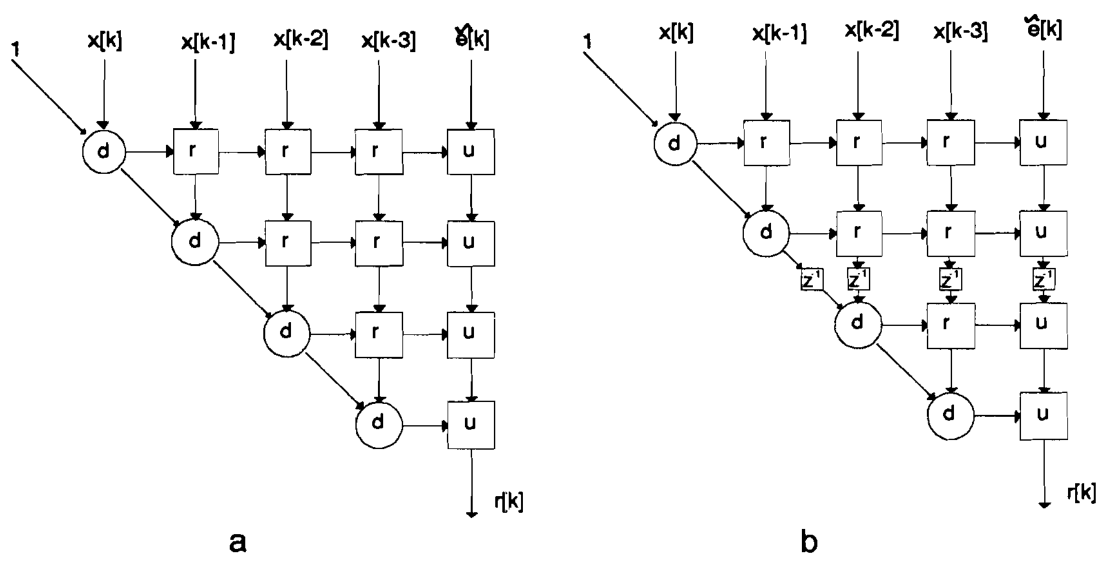


Figure 14: Triangular Systolic Array implementation

Because the implementation is a feed forward systolic array, it is possible to delay intermediate results. In figure 15 the 8 stages of calculating one sample are shown. PE number 1 starts the calculation. If PE 1 is ready the PE's with the number 2 starts and so on. When delaying intermediate results this has to be done between successive numbered PE's. The possible places for delays are given with dashed lines. In figure 14b an example with one delay is given.

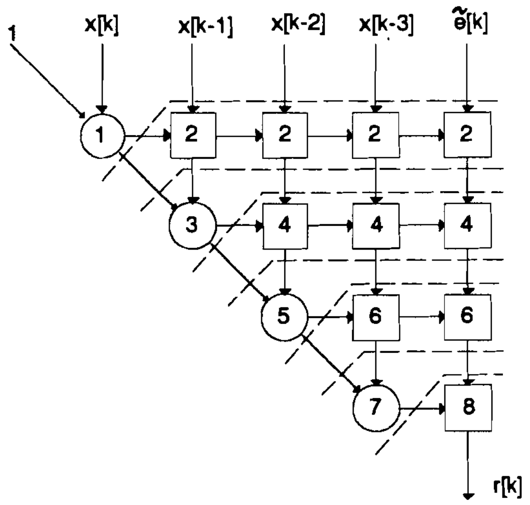


Figure 15: Possible delay positions

By delaying the calculation between 0 and 2N samples the calculation time (ΔT) becomes smaller. This results in a processing delay (L). Notice that if L=N or L=2N the calculation time is independent from the number of filter coefficients but, for large filters, the processing delay can become very large.

This results in the following quantities:

$$C_{TSA} = N^2 + 9N$$
$$0 \leq L_{TSA} < 2N$$

$$\Delta T_{TSA} = \begin{cases} 6\frac{N}{L+1} & \wedge \;\; 0 \leq L_{TSA} < N \\ 8\frac{N}{L+1} & \wedge \;\; N \leq L_{TSA} < 2N \end{cases}$$

This implementation looks promising, but the number of multiplications becomes very large for large filters. If a filter has 1024 coefficients 1.057.792 multiplications are necessary every sample. Because of the large number of multiplications needed this implementation is not investigated further.

4.3 Least Squares Lattice

In the implementation of the Givens algorithm using a triangular systolic array (§4.2), the property that the input data is a combination of a shifted version of the previous input data and a new sample is not used. By exploiting this property it is possible to develop a Least Squares Lattice (LSL) with a complexity of O(N)(see Ling[9][11] and Yang et.al.[10]). The LSL exists of N equal stages, and is therefore easy to develop. The advantage of this LSL algorithm over the FRLS algorithms is that the LSL algorithm has better numerical properties.

The LSL implementation used here is described by Ling[9][11]. It consists of N lattice stages. Each lattice stage does the calculation of one coefficient, so to extend the filter with one coefficient, only one PE has to be added at the end of the filter. The F and B elements calculate the forward and backward reflection coefficient used by the lattice. The E element calculates the weight vectors. A more extensive explanation of the lattice stages and needed calculations is given in appendix C. In figure 16 an outline of a LSL filter with two coefficients is given.

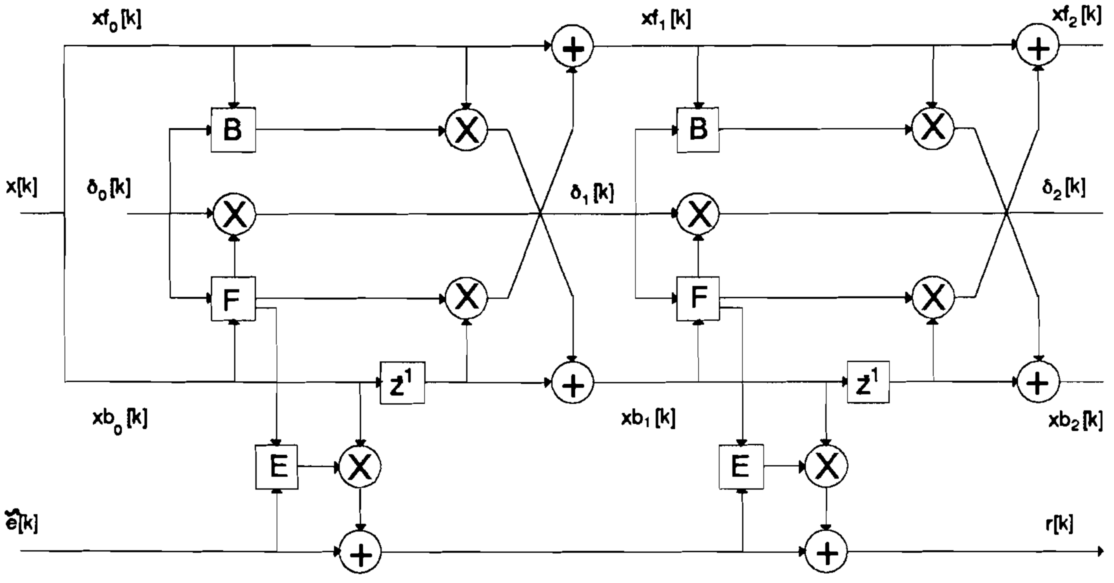


Figure 16: LSL implementation

One of the advantages of this implementation is that, because the implementation is a feed forward systolic array, it is possible to delay the calculations for 1 to N samples without influencing the filter properties. By doing this all lattice sections directly after a delay element can be calculated parallel, all lattice stages after these stages can be calculated parallel and so on. (Each stage can be calculated in ΔT = 4, see appendix C). This results in the following quantities:

C_{LSL} = 22N
$0 \le L_{\mathrm{LSL}} < \mathrm{N}$
$\Delta T_{\mathrm{LSL}} = 4\mathrm{N}/(L+1)$

Although the complexity of this algorithm is high (C = 22N) the calculation time is very small (especially if a large processing delay is possible). Furthermore the implementation is highly regular and therefore easy to implement.

4.4 Experiments

In this paragraph experiments are done with the FRLS and LSL implementations. No experiments are done with the triangular systolic array implementation because this implementation is not suited to be implemented in hardware when a large number of coefficients are needed.

In the experiments the filterlength is N=1024 and a window with γ=0.999 is used. The echopath is an exponentially decaying filter with N coefficients. Furthermore the inputsignals are White Noise, MA(1), AR(1) and a stationary female unvoiced signals (see §2.3). The results of these experiments are given in figure 17a-d.

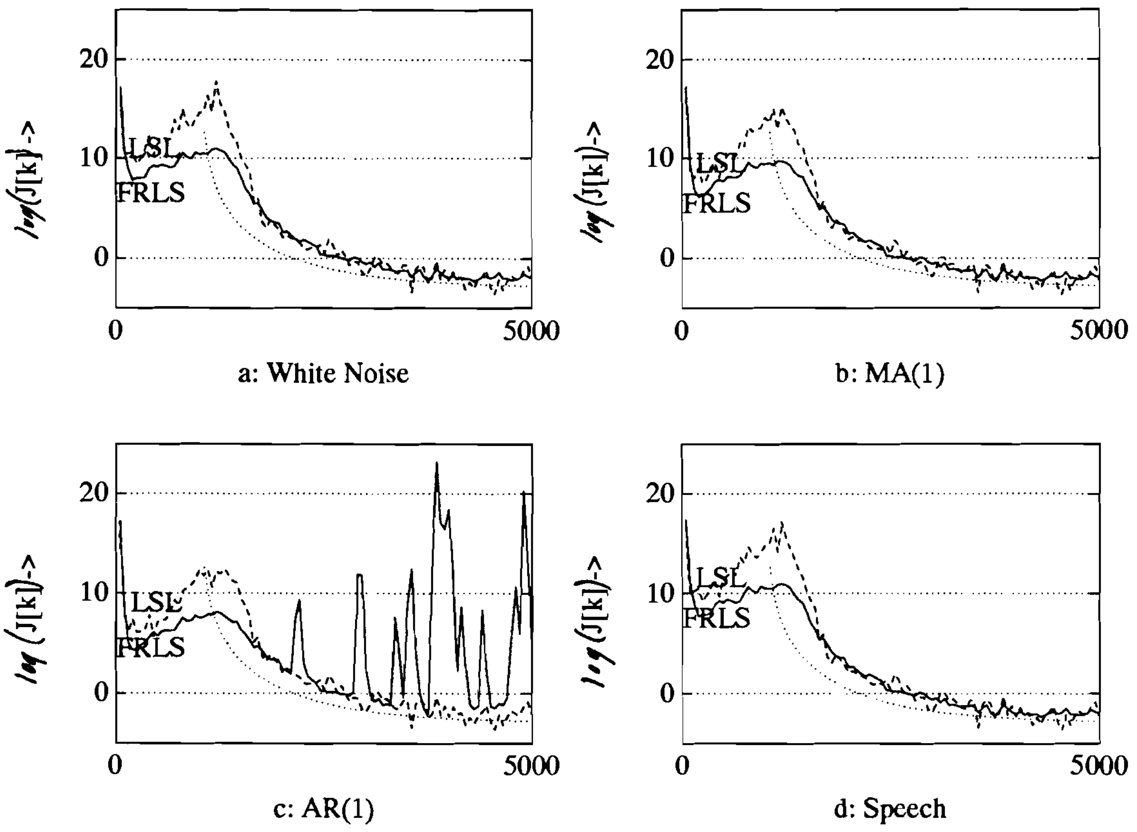


Figure 17: Experiment with FRLS and LSL implementations

Although it was expected that both algorithms had the same response, the FRLS algorithm has, in the beginning, a better response than the LSL algorithm. Notice that the signals are not completely decorrelated in the beginning (especially for the AR(1) signal). This is because the inverse autocorrelation matrix $R_x^{-1}[k]$ is not jet converted and thus is not jet equal to R_x^{-1}. An other important observation is that the FRLS algorithm is not stable in the experiment with the AR(1) signal. Although the algorithm stabilizes itself within a few samples, it is a large disadvantage of this implementation.

4.5 Conclusions

In this chapter 3 types of implementations of the RLS algorithm are investigated. All implementations realize the same RLS algorithm. The difference between the implementations is that the algorithm is calculated in different ways, and because of this, have different numerical properties. To compare the different algorithms, the most important characteristics are given in table 2.

Table 2: Summary of different RLS implementations

Implementation	C	L	ΔT
FRLS	10N+6	0	$8 + N/M + 2 \cdot {}^{M}\log(N)$ $\leq 8 + 1{,}5 \cdot N$
Givens rotation base RLS using triangular systolic array	N^2+9N	$0 \leq L \leq N$ $N \leq L \leq 2N$	$6N/(L+1)$ $8N/(L+1)$
LSL	22N	$0 \leq L \leq N$	$4N/(L+1)$

with M is the number of additions possible per multiplication

First the Fast RLS algorithm is investigated. This algorithm has only a fast calculation time if N/M is small. The advantage is that the algorithm has no processing delay. The disadvantage of this implementation is that it has numerical problems and that the implementation is very complex.

The second type of implementations (the Givens rotation based algorithm using a triangular systolic array) has good numerical properties and is easily implemented in a triangular systolic array. Furthermore it is possible to become fast calculation times by delaying the calculations. The disadvantage of this implementation is that for a large number of filter coefficients, the number of PE's becomes very large. So this parallel implementation is not suited for large filters.

The last type of implementation is the Least Squares Lattice. This implementation is numerical stable and is easily implemented in hardware with a linear systolic array. Also fast calculation times are possible by delaying the calculations. This delaying of the calculations introduces a processing delay equal to the delay in the calculations.

5 Frequency Domain Adaptive Filters

An often used way to decorrelate a signal is to normalise its spectrum. This is possible because the autocorrelationfunction and the power density function form a Fourier transformation pair. To normalise the spectrum a Discrete Fourier Transformation is needed.

Applying this decorrelation to LMS, results in a FDAF filter. In figure 18 an outline of a FDAF filter with 4 coefficients is given. In figure 19 a FDAF update part is given. Notice that the normalisation of the spectrum is done by dividing the update coefficient by the average power of the signal in this frequency band. (See Haykin[7] and Bellanger[6]). For this filter the following formulas apply:

$$\hat{e}[k] = \underline{x}^T[k]\cdot\underline{w}[k] = \frac{1}{N}\,\underline{X}^T[k]\cdot\underline{W}^*[k]$$
$$r[k] = \tilde{e}[k] - \hat{e}[k]$$
$$\underline{W}^*[k+1] = \underline{W}^*[k] + 2\alpha P^{-1}\underline{X}^*[k]r[k]$$

$$\textit{with}\begin{cases} P = \frac{1}{N}\,E\{\underline{X}^*[k]\underline{X}^T[k]\} \\ \underline{X}[k] = F\underline{x}[k] \\ \underline{W}[k] = F\underline{w}[k] \end{cases}$$

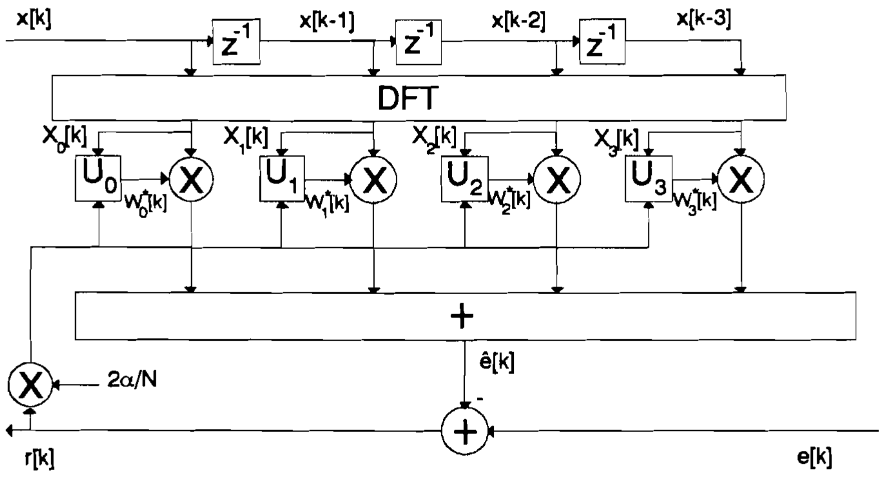


Figure 18: Frequency Domain Adaptive Filter

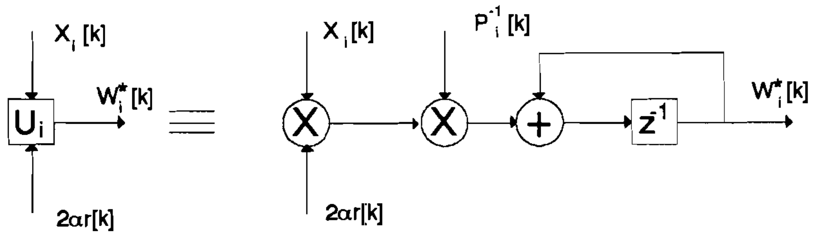


Figure 19: FDAF update part

When implementing the FDAF filter parallel there are two problems: the DFT and the addition tree. As will be shown in the next paragraph it is possible to calculate the DFT with O(N) multiplications. The problem of the addition tree is the same as in the LMS filter.

5.1 Sliding window DFT

When implementing the FDAF algorithm parallel the problem exists of implementing the DFT. Although very fast DFT's are available it is also possible (if a sliding window is used) to use an adapted DFT. To adapt the DFT we write the calculation of the DFT as follows :

$$\begin{aligned} X_l[k] &= \sum_{i=0}^{N-1} e^{-j\cdot\frac{2\pi}{N}\cdot i\cdot l}\cdot x[k-i] \\ &= x[k] + \sum_{i=1}^{N} e^{-j\cdot\frac{2\pi}{N}\cdot i\cdot l}\cdot x[k-i] - x[k-N] \\ &= e^{-j\cdot\frac{2\pi}{N}\cdot l}\cdot X_l[k-1] + x[k] - x[k-N] \end{aligned}$$

Looking at this formula it can be seen that it is numerical unstable because of the (in time) growing numerical error. This can be solved by applying a window before the DFT. The used window is an exponentially decreasing window of length N. This leads to the following formulas:

$$window = \begin{cases} \gamma^k & 0 \le k < N \quad with\ 0 < \gamma < 1 \\ 0 & else \end{cases}$$

$$SDFT : X_l[k] = \sum_{i=0}^{N-1} \gamma^i \cdot e^{-j \cdot \frac{2\pi}{N} \cdot i \cdot l} \cdot x[k-i]$$

$$= \gamma \cdot e^{-j \cdot \frac{2\pi}{N} \cdot l} \cdot X_l[k-1] + x[k] - \gamma^N \cdot x[k-N]$$

This results in the following numerical error :

$$\xi[k] = (1+d)\gamma \cdot \xi[k-1] + d + \gamma^N \cdot d \quad with\ d = machine\ accuracy$$
$$\approx (1+d)\gamma \cdot \xi[k-1] + 2d$$

$$\Rightarrow \xi[\infty] = \frac{2d}{1-\gamma(d+1)} \Rightarrow \gamma < \frac{1}{1+d}$$

The numerical error becomes larger if γ becomes larger. Because of this we have to choose γ as small as possible. On the other hand, if we choose $\gamma \ll 1$ the window will have a negative effect on the decorrelation of the signal. To study the influence of γ on the resulting frequency response we calculate the height of the first side-lob and the 3 dB bandwidth (see figures 21). Comparing these results with the quantities of DFT (side-lob = -13.28 dB and 3 dB bandwidth = 0.81/N, see figure ?) it is seen that if $\gamma >$ $1\text{-}10^{-4}$ the influence of the exponential exponent γ of the window can be neglected.

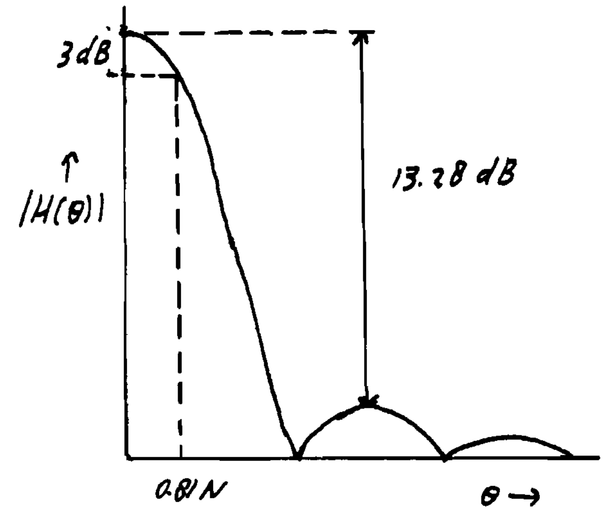


Figure 20: Frequency response of a DFT

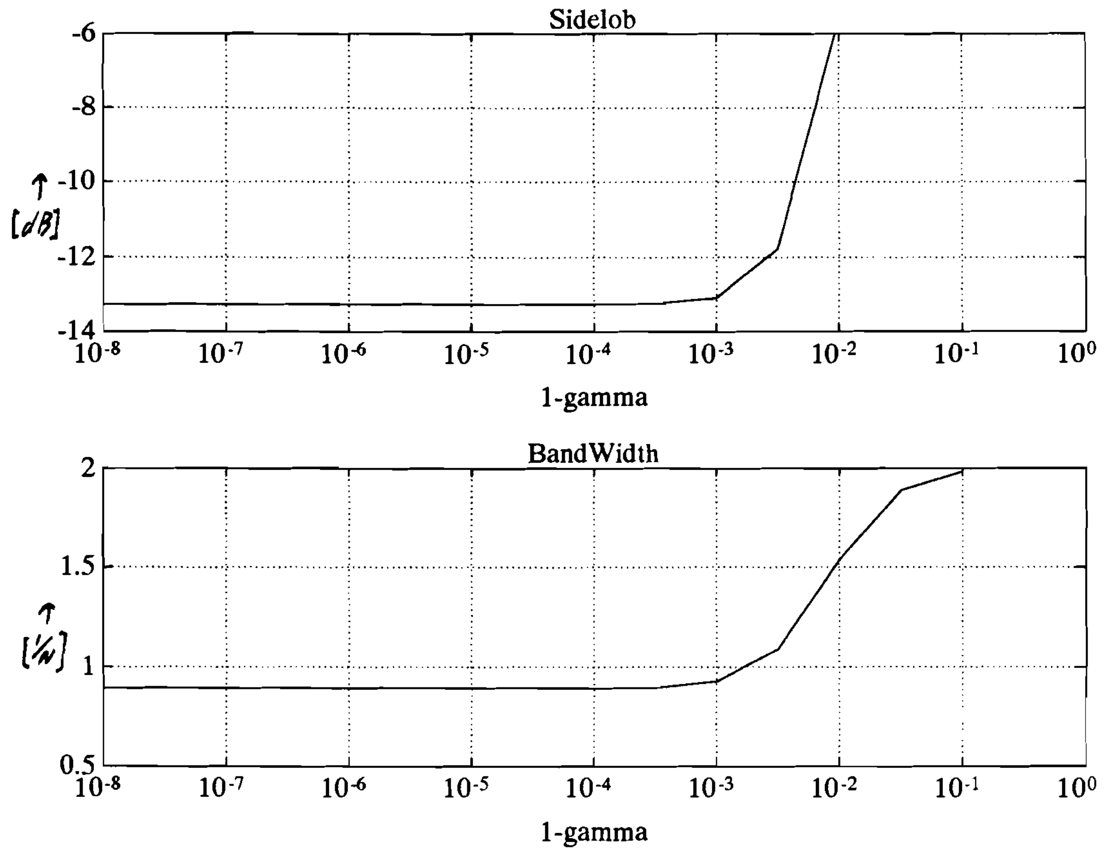


Figure 21: Bandwidth and hight of first side-lob

The advantage of this way of calculation is easily seen in figure 22. Aside of the low computational complexity, it is also very easy to implement the algorithm parallel. The calculations of all frequency bins can be done parallel. Notice that this implementation is not possible when block processing is used.

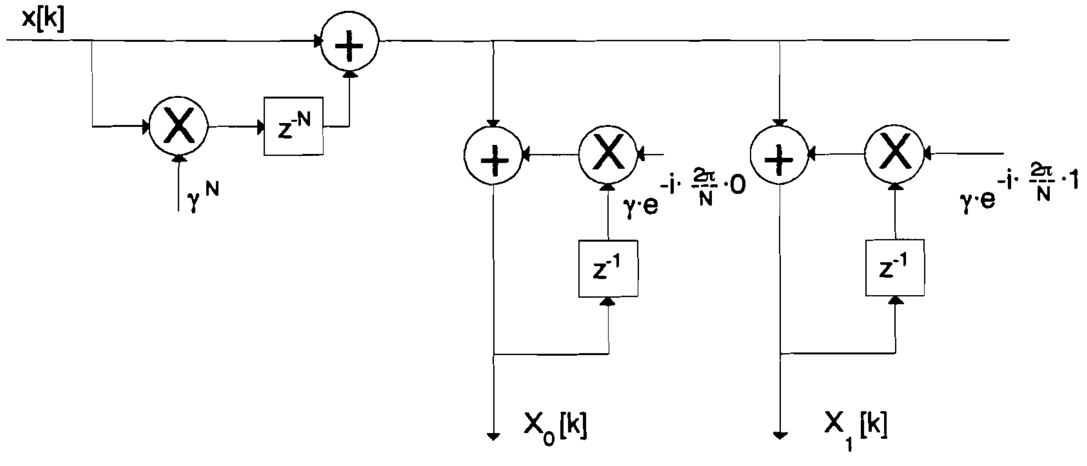


Figure 22: DFT with sliding window implementation

When calculating the SDFT in this way the computational costs are 4N+1 (a complex multiplication exists of 4 real multiplications). If the input signal x[k] is a real signal $X_l[k] = X_{N-l}^*[k]$. This reduces the computational costs of the SDFT to 2N+1.

5.2 Delayed Sliding Window FDAF

When implementing the FDAF algorithm with the sliding window DFT discussed in paragraph 5.1, the same summation problem exists as with the LMS algorithm. The solution of rewriting the convolution as was done in the LMSDW algorithm is not possible because there is no time delayed signal. The other solution, a delayed addition tree as in the DLMS algorithm, is possible. In figure 23 an outline of a Delayed Sliding Window FDAF (DFDAF) with 4 coefficients is given. The update part is the same as in the FDAF implementation (see figure 19).

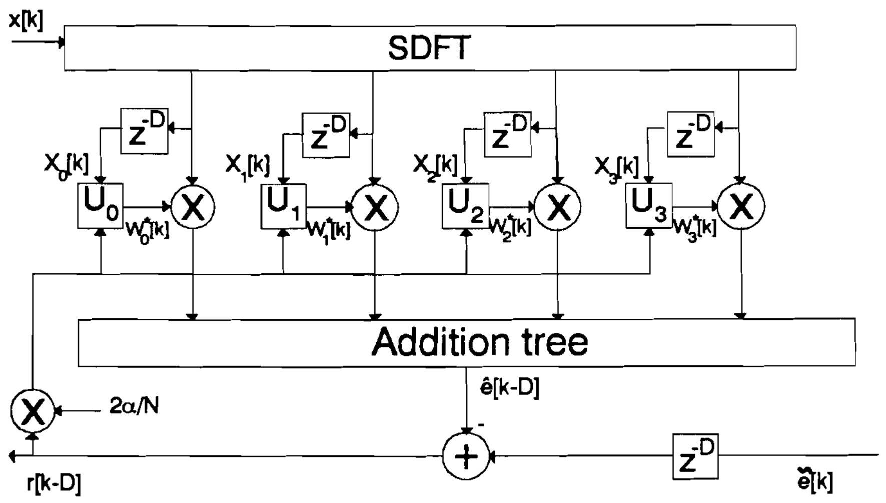


Figure 23: Delayed sliding window FDAF

In this implementation the average signal powers of the frequency bins are calculated with the following formula:

$$P_l[k+1] = \beta \cdot P_l[k] + (1-\beta)\frac{|X_l[k]|^2}{N} \quad \textit{with} \quad 0 < \beta < 1$$

When assuming that the average signal powers are perfectly estimated and D is the delay introduced by the addition tree, this leads to the following final misadjustment and rate of convergence:

$$J[\infty] = \frac{\alpha N}{1 - \alpha(N+2D)}$$

$$\nu_{20} = \frac{-2}{{}^{10}\log(1 - 4\alpha)} \quad (\textit{valid after } k>D)$$

As can be seen these quantities are independent of the input signal statistical properties. Furthermore the influence of the introduced delay is the same as for the DLMS algorithm. Therefore the delay has to be kept as small as possible, which is discussed in paragraph 3.1.

The filter part can be implemented parallel the same way as the DLMS algorithm (see paragraph 3.1). The SDFT part can be implemented parallel as in discussed paragraph 5.1. This results in the following quantities:

$C_{\mathrm{DFDAF}} = 6{,}5 \cdot \mathrm{N}+1$
$L_{\mathrm{DFDAF}} = \mathrm{D}$
$\Delta T_{\mathrm{DFDAF}} = 6$

As can be seen the calculation time is independent of the number of coefficients.

5.3 Experiments

In the first experiment a FDAF filter with FFT and with a SDFT (as described in §5.1) are compared. The filters have a length of N = 1024, $\alpha = 0.0001$ and $\gamma = 0.9999$. These two filters are tested with the 4 signals described in §2.3. The results of this experiment are given in figure 24. As can be seen, the filter response of the FDAF with the SDFT implementation has the same response as the FDAF with the FFT implementation.

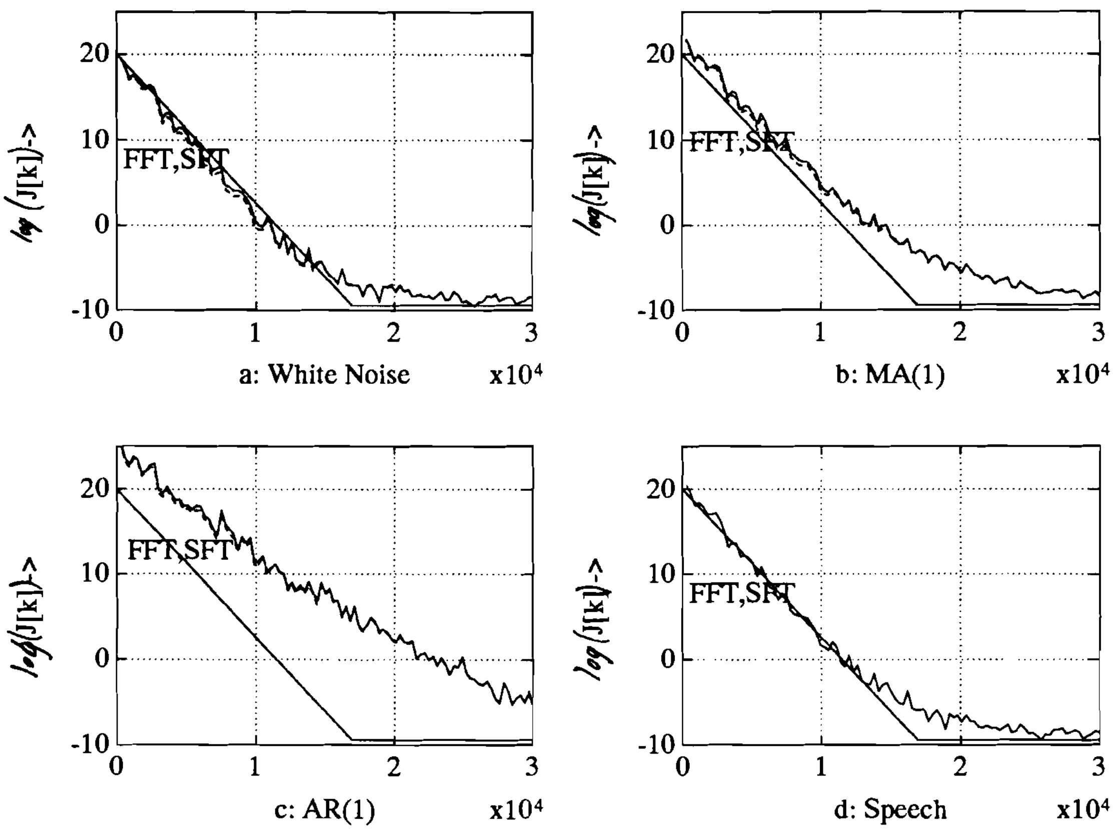


Figure 24: Experiment with FDAF with DFT and SDFT

In the second experiment the final misadjustment of the DFDAF algorithm is investigated with respect to the processing delay of the filter. This is done for white noise and a filter length of N=1024. The processing delay of the filter has a maximum of $L < {}^2\log(N)=10$. In the experiment the echopath is an exponentially decaying impulse response of length N with a factor of 0.75. The results of this experiment can be found in figure 25. From this it follows that the final misadjustment of DFDAF filter does not change noticeably if $2D \ll N$.

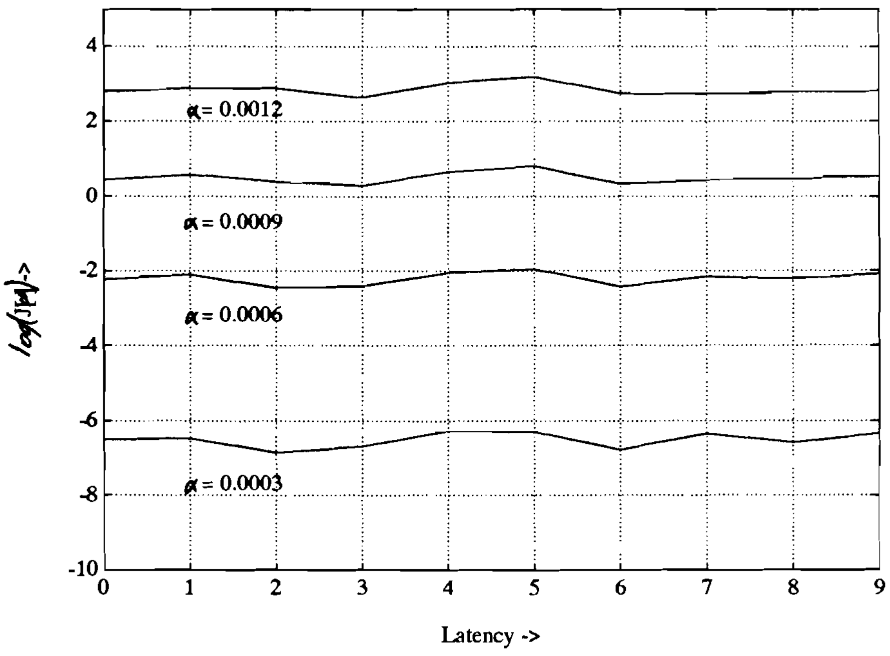


Figure 25: Final misadjustment of DFDAF

In the third experiment, the final misadjustment of a FDAF and a DFDAF is investigated for different α. Both filters have 1024 coefficients and for the DFDAF filter L=10. Furthermore white noise (σ_x^2 = 1/3) is used. The result can be found in figure 26. As can be seen the DFDAF (with 2D<N) and the FDAF implementations have the same final misadjustment.

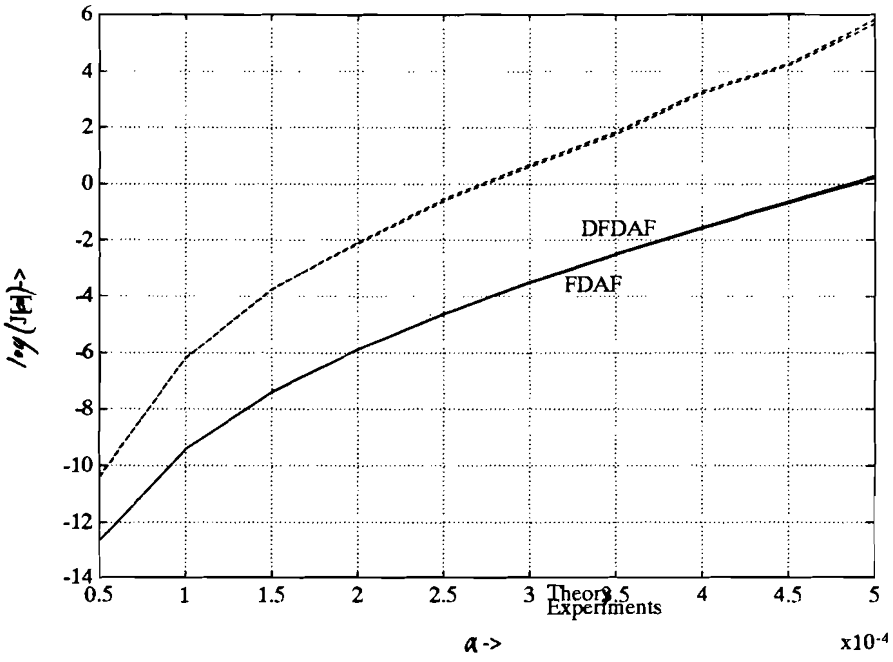


Figure 26: Final misadjustment for different α

In the last experiment a FDAF and a DFDAF filter are tested with the four types of signals described in §2.3. All filters are of length N=1024 and α=0.0001. The DFDAF filter has a processing delay of L = $^2\log(N)$ = 10. The results of the filter responses can be found in figure 27. As can be seen the filter response of the DFDAF algorithm is almost equal to the FDAF algorithm response.

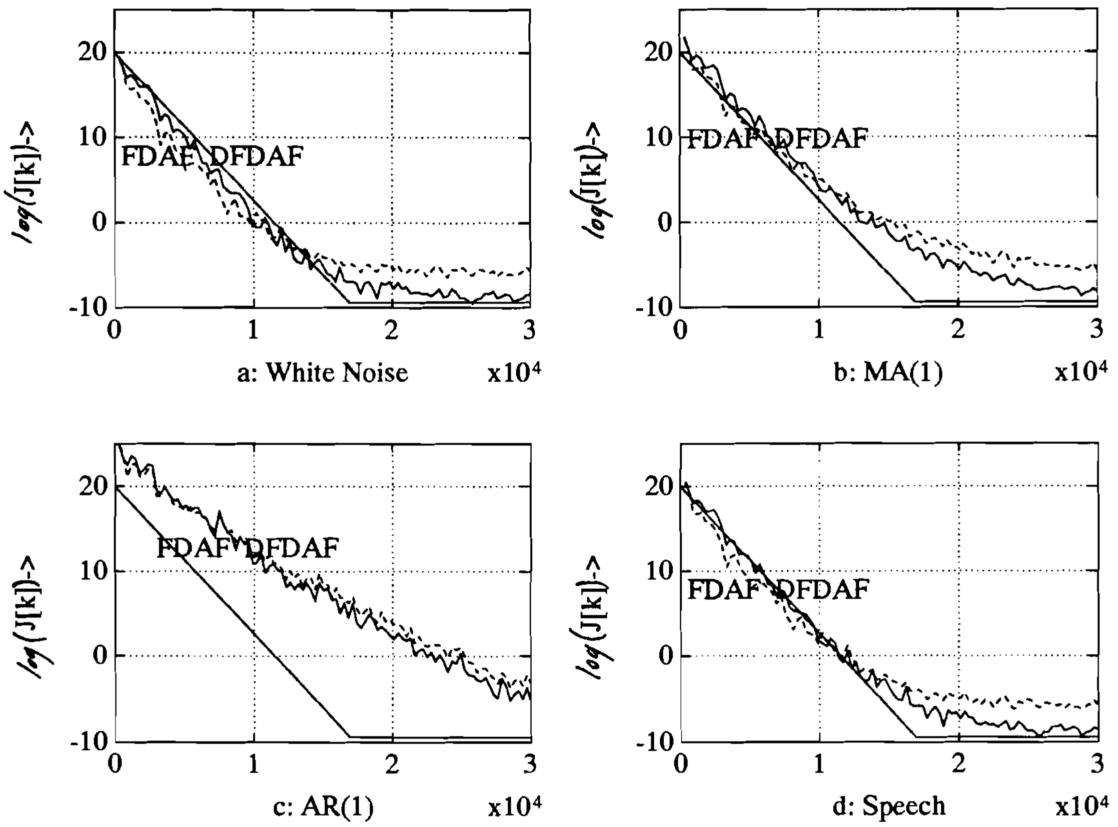


Figure 27: Experiment with different filters and signals

5.4 Conclusions

This chapter first gives an implementation of a DFT with a sliding window. This implementation reduces the order of the DFT calculation from $O(N^2\log(N))$ for FFT to O(N) for SDFT. The other advantage of this implementation is that it can easily be implemented on parallel hardware. The disadvantage of this implementation is that it can only be applied to DFT's with a sliding window.

Secondly a new type of filter is introduced, called Delayed Frequency Domain Adaptive Filer (DFDAF). This new type of filter makes use of a delayed addition tree which introduces a processing delay. The advantage of this implementation is that its calculation time becomes small and independent of the number of coefficients (ΔT = 6). Finally the most important characteristics of the FDAF and DFDAF algorithms are given in table 3.

Table 3: Summary of FDAF and DFDAF characteristics

Filter Type	$J[\infty]$	υ_{20}	C	L	ΔT
FDAF	$\frac{\alpha N}{1 - \alpha N}$	$\frac{-2}{{}^{10}\log(1-4\alpha)}$	6,5·N+1	0	6,5·N+1 (No parallel calculations used)
DFDAF	$\frac{\alpha N}{1 - \alpha(N+2D)}$	$\frac{-2}{{}^{10}\log(1-4\alpha)}$	6,5·N+1	D	6

6 Conclusions and recommendations

Looking at the LMS algorithm it is seen that it is not suited to be implemented parallel. To overcome the problems, two LMS like algorithms are developed. Both algorithms are suited to be implemented parallel. The first algorithm, Delayed LMS (DLMS), has almost the same behaviour as the LMS algorithm, but introduces a small amount of processing delay. The second algorithm, LMS with Delayed Weights (LMSDW), has, for large α, a worse final misadjustment, but has no processing delay. Because of the parallel implementations, both algorithms can be calculated in a small constant time (time needed for 2-3 multiplications) independent of the number of filter coefficients.

Three different implementations of the RLS algorithm are investigated. The first algorithm, called the Fast RLS (FRLS) or Fast Kalman algorithm, has numerical instability problems. Although a speedup in the calculationtime can be achieved by performing the calculations parallel, this algorithm is not very well suited to be implemented parallel. The second algorithm, a Givens rotation based triangular systolic array, can easily be implemented parallel, but because of the needed number of multiplications this algorithm is not suited for large filters. When implementing the last algorithm parallel, the Least Squares Lattice (LSL), fast calculations times can be achieved. By introducing some processing delay, the calculation time can be made very small (down to the time needed for 4 multiplications).

When implementing the FDAF algorithm, two problems exist: the Discrete Fourier Transformation (DFT) and the summation tree. The summation problem can be solved in the same way as was done by the DLMS algorithm. To overcome the DFT problem a Sliding window DFT (SDFT) is used. Although the Fourier transformation properties changes when using a window, the window can be chosen in such a way that the properties only degenerate very little. By using these solutions a new algorithm, Delayed FDAF (DFDAF), is developed. The DFDAF algorithm has almost the same behaviour as the FDAF algorithm, but introduces a small amount of processing delay. Because of the parallel implementation, the DFDAF algorithm can be calculated in a small constant time (time needed for 6 multiplications) independent of the number of filter coefficients.

When looking at the parallel implementations of the adaptive filters it is seen that there is no general solution of how to implement an algorithm parallel, so all algorithms have to be investigated separately. Nevertheless some general observations can be made. A reduction in calculation time by doing calculations parallel, results in more hardware. It is also possible, in some cases, to introduce some processing delay to decrease the calculation time (e.g. DLMS, DFDAF, LSL).

It is further seen that in some cases the algorithm must be changed to get an algorithm which is suited to be implemented parallel. Because of these changes the properties of the algorithm change, but these changes can be made small for most implementations (e.g. DLMS, DFDAF). Other algorithms have already implementations which are suited to be implemented parallel (e.g. RLS). The properties of these algorithms do not change therefore.

Finally some recommendations for further research are given. First the influence of the statistical properties of the input signal of the LMSDW algorithm still has to be investigated. When looking at the research being done at this moment, only research is done on the LMS and RLS algorithms. In this thesis also FDAF algorithms are investigated. Further research areas are therefore other algorithms. Especially the frequency domain algorithms using block processing are an interesting field of further research because knowledge of the algorithms already exists on this university.

Literature

[1] Adaptive FIlter Theory.
Prentice-Hall International Editions,
pp. ....-...., 1991,
S. Haykin.

[2] A modular pipelined implementation of a delayed LMS transversal adaptive filter.
IEEE Circuits and Systems,
Vol. 3, pp. 1943-1946, 1992,
M.D. Meyer and D.P. Agrawal.

[3] The LMS algorithm with Delayed Coefficient Adaption
IEEE Transactions on ASSP,
Vol. 37, No. 9, pp. 1397-1405, 1989,
G. Long, F. Ling and J.G. Proakis.

[4] A fully systolic adaptive filter implementation.
IEEE International Conference on ASSP 1991,
ICASSP 1991, pp. 2109-2112, 1991,
D.B. Chester, R. Young and M. Petrowski.

[5] Stabilizing the Fast Kalman algorithms.
IEEE Transactions on ASSP,
Vol. 37, No. 9, pp. 1342-1348, 1989,
J.L. Botto and G.V. Moustakides.

[6] Adaptive Digital Filters and Signal Analysis.
Marcel Dekker Inc.
pp. 208-209, 1987,
M.C. Bellanger.

[7] Adaptive Filter Theory.
Prentice-Hall International Editions,
pp. 538-544, 1991,
S. Haykin.

[8] Algorithmic engineering: A worked example.
Signal Processing VI: Theories and Applications,
pp. 5-12, 1992,
J.G. McWhirter and I.K. Proudler.

[9] Givens rotation based Least Squares Lattice and related algorithms.
IEEE Transactions on Signal Processing,
Vol. 39, No. 7, july 1991, pp. 1541-1551, 1991,
F. Ling.

[10] Rotation based RLS algorithms: Unified derivations, numerical properties and

parallel implementations.
IEEE Transactions on Signal Processing,
Vol. 40, No. 5, pp. 1151-1166, 1992,
B. Yang and F. Böhme.

[11] Effective LSL algorithms based on Givens rotation with systolic array implementation.
IEEE International Conference on ASSP 1989,
ICASSP 1989, pp. 1290-1293, 1989,
F. Ling.

Appendix A: The Fast Recursive Least Square algorithm

In this appendix a Fast RLS algorithm, as given by Bellanger (see [6]), of C = 8N is described. As described in §4.1, the FRLS algorithm can be divided in 4 subfilters (see figure 13). In this appendix first the 4 subfilters are discussed separately and finally the total filter is discussed.

The forward linear prediction filter (labelled 1) calculates the forward linear prediction and the forward prediction error energy. An outline of the filter with 2 coefficients is given in figure 28. The update part is given in figure 29. The filter is described by the following formula's:

$$\begin{aligned}
&eav[k] = x[k] - \underline{x}^T[k-1]\cdot\underline{a}[k-1]\\
&\underline{a}[k] = \underline{a}[k-1] + eav[k]\cdot\underline{g}[k-1]\\
&epsa[k] = x[k] - \underline{x}^T[k-1]\cdot\underline{a}[k]\\
&ea[k] = \gamma\cdot ea[k-1] + eav[k]\cdot epsa[k]
\end{aligned}$$

with:

eav[k] the forward prediction error
$\underline{a}$[k] the forward prediction coefficients
epsa[k] the new forward prediction error
ea[k] the prediction error energy

When implementing this filter parallel this leads to the following quantaties:

C = 3N+2
L = 0
ΔT = 2 + ^{M}log(N) (^{M}log(N) is the time needed by the summation part)

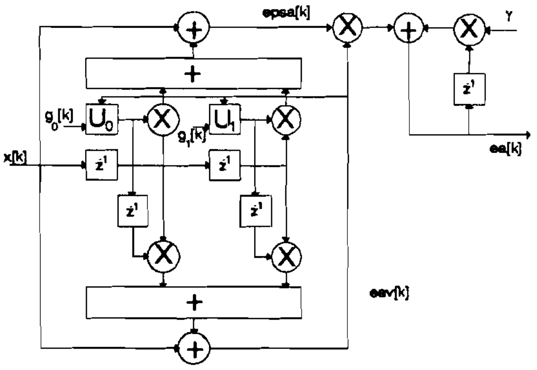


Figure 28: Forward prediction filter

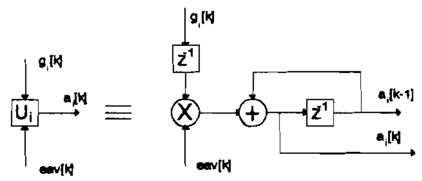


Figure 29: Update part

The backward linear prediction filter (labelled 2) calculates the backward linear prediction and the backward prediction error. An outline of the filter with 2 coefficients is given in figure 30. The update part is given in figure ?. The filter is described by the following formula's:

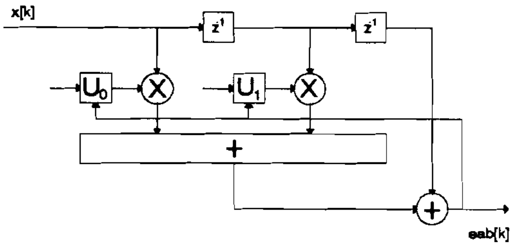
$$eab[k] = x[k-N] - \underline{x}^T[k]\cdot\underline{b}[k-1]$$
$$\underline{b}[k] = \underline{b}[k-1] + eab[k]\cdot\underline{g}[k]$$

with:

eab[k] the backward prediction error
$\underline{b}[k]$ the backward prediction coefficients

When implementing this filter parallel this leads to the following quantaties:

$C = 2N$
$L = 0$
$\Delta T = 1 + {}^{M}\log(N)$ (${}^{M}\log(N)$ is the time needed by the summation part)



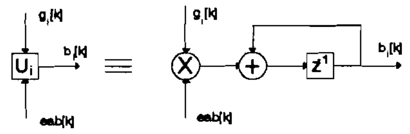


Figure 30: Backward prediction filter **Figure 31: Update part**

Filter 3 calculates the gain $\underline{g}[k]=R_x^{-1}[k]\underline{x}[k]$. This is done by using the forward/backward prediction coefficients and the forward/backward prediction errors calculated by the forward and backward prediction filters (see Bellanger [6]). To calculate a new gain vector, a gain vector with N+1 taps is calculated and then transformed back to a gain vector of N taps. As can be seen in figure 32 the implementation of this subfilter does not have a FIR structure.

Looking at the implementation it can be seen that for the calculation of gl_i it is necessary to calculate gl_{i-1}. Because of this a new addition problem arises because if large filters are used, the time needed by the additions to calculate gl_N cannot be neglected any more. If it is possible to add M signals in the time of one multiplication, the time needed to calculate the Nth signal takes N/M times the time needed for one multiplication. The filter is described by the following formula's:

gl_0 = epsa[k]/ea[k]
$\forall_{i=0,1,\ldots,N-1}$: $gl_{i+1} = gl_i - gl_0 \cdot a_i[k]$
$\forall_{i=0,1,\ldots,N-1}$: $g_i[k] = (gl_N \cdot b_i[k-1] + gl_i) / (1 - eab[k] \cdot gl_N)$

with:

$\underline{gl}$ the (temporary) gain vector (N+1 taps)
$\underline{g}$[k] the gain vector (N taps)
gl_i and $g_i[k]$ the i^th element of the vector

When implementing this filter parallel this leads to the following quantaties:

C = 3N+3
L = 0
ΔT = 5 + M/N (M/N is the time needed by the summation part)

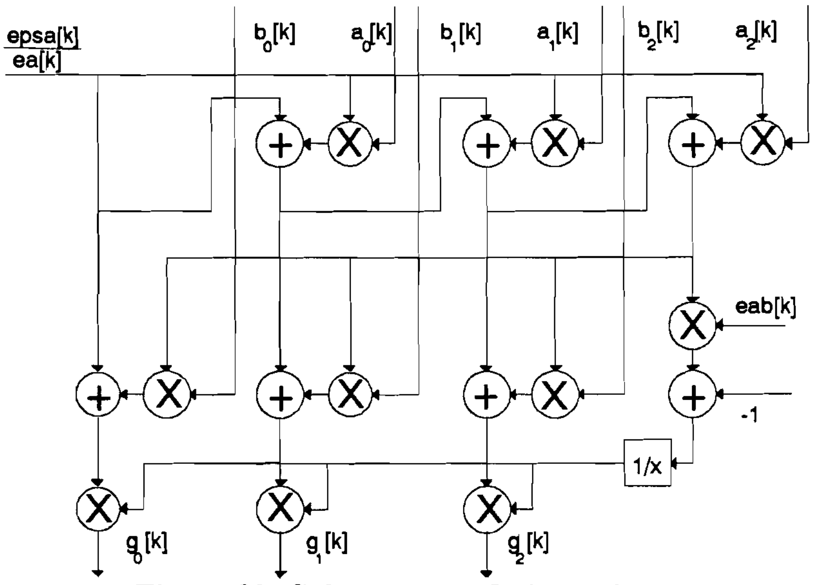


Figure 32: Subfilter calculating gain vector

The last subfilter is the part which adapts the filter weights based on the gain vector $\underline{g}$[k], calculated by the gain subfilter (labelled 3). Furthermore the filter calculates the estimated signal ẽ[k] and the filter output r[k]. This subfilter strongly resembles a LMS filter with the difference that the new update coefficients are calculated with the gain vector ($\underline{g}[k]=R_x^{-1}[k]\cdot\underline{x}[k]$) instead of just the input signal vector $\underline{x}$[k]. An outline of this subfilter with 4 coefficients is given in figure 34. The update part is given in figure ?. The filter is described by the following formula's:

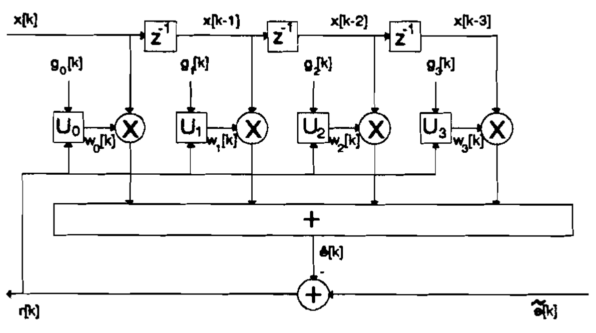
$\underline{w}[k] = \underline{w}[k-1] + \underline{g}[k-1]r[k-1]$
$\hat{e}[k] = \underline{x}^T[k] \cdot \underline{w}[k-1]$
$r[k] = \tilde{e}[k] - \hat{e}[k]$

with:

$\underline{w}[k]$ the vector of filter coefficients
$\hat{e}[k]$ the estimated signal
$r[k]$ the filter output

When implementing this filter parallel this leads to the following quantaties:

$C = 2N$
$L = 0$
$\Delta T = 1 + {}^{M}\log(N)$ (${}^{M}\log(N)$ is the time needed by the summation part)



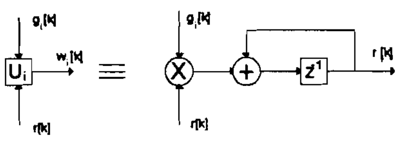


Figure 33: Signal estimator **Figure 34: Update part**

To make the FRLS filter, the 4 subfilters have to be interconnected as shown in figure 13. The problem of this algorithm is that 5 times N signals have to be added, and because N can become very large, the time needed for the additions cannot be neglected any more. To minimize the time needed by the additions 4 summations can be done with addition trees (the summations in filters 1,2 and 4). Looking at the addition problem in the gain filter (filter 3) it is seen that the use of an addition tree is not possible. Therefore the calculation time of the parallel implemented FRLS filter becomes $\Delta T = 8 + N/M + 2\,{}^{M}\log(N)$, which largely depends on the factor N/M.

Appendix B: Givens based triangular systolic array

In this appendix a Givens based triangular systolic array, as given by Haykin (see [7]), is described. Because only the posteriori estimation error r[k] has to be known, it is not necessary to calculate the weight vector $\underline{w}$[k]. The posteriori estimation error r[k] can also be calculated by multiplying the priori estimation error α[k] with the conversion factor δ[k] (r[k] = α[k]$\cdot\delta$[k]).

To see how the algorithm works the following formula's apply:

$$Q[k]\cdot\Lambda^{\frac{1}{2}}[k]\cdot X[k] = \begin{bmatrix} R[k] \\ O \end{bmatrix} = T[k]\cdot\begin{bmatrix} \sqrt{\gamma}\cdot R[k-1] \\ 0 \\ X^T[k] \end{bmatrix}$$

$$Q[k]\cdot\Lambda^{\frac{1}{2}}[k]\underline{r}[k] = \begin{bmatrix} p[k] \\ v[k] \end{bmatrix} = T[k]\cdot\begin{bmatrix} \sqrt{\gamma}\cdot p[k-1] \\ \sqrt{\gamma}\cdot v[k] \\ \bar{e}[k] \end{bmatrix}$$

$$\sqrt{\delta[k]} = \prod_{i=0}^{N-1} c_i[k]$$

$$J_i[k] = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & c_i & 0 & \dots & s_i^* \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & -s_i & 0 & \dots & c_i \end{bmatrix} \quad \textit{with} \quad \begin{aligned} c_i &= \frac{|y_{ii}|}{\sqrt{y_{ii}^2 + y_{ki}^2}} \\ s_i &= \frac{y_{ki}}{y_{ii}}\cdot c_i \end{aligned}$$

with:

Q[k] unitary matrix (k*k)
Λ[k] = diag(γ^{k-1}, γ^{k-2}, ..., γ^1, 1)

$$X[k] = \begin{bmatrix} x[1] & x[2] & x[3] & \dots & x[k] \\ 0 & x[1] & x[2] & \dots & x[k-1] \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & x[k-N+1] \end{bmatrix} \text{ a (k*N) matrix}$$

R[k] is the (N*N) upper triangle matrix used by QR-decomposition (notice that R[k] is not equal to the autocorrolation matrix).
T[k] = $J_{N-1}[k]\cdot J_{N-2}[k]\cdot\ldots\cdot J_0[k]$ a (k*k) matrix consisting of k Givens rotations
J_i[k] a (k*k) Givens rotation
$\underline{r}^T$[k] = (r[k], r[k-1], ..., r[0])
p[k] a (N*1) matrix
v[k] a ((k-N)*1) matrix

$\delta[k]$ the conversion factor between priori and the posteriori estimation errors
y_{ij} element on ith column, jth row in matrix on which the Givens rotation works

Because only the conversion factor and the priori estimation error have to be calculated it is not necessary to calculate v[k]. Also only the first N rows and columns of the matrix R[k] have to be calculated. This prevents that the matrixes and vectors from keep on growing. A possible implementation of this algorithm is given in figure 35.

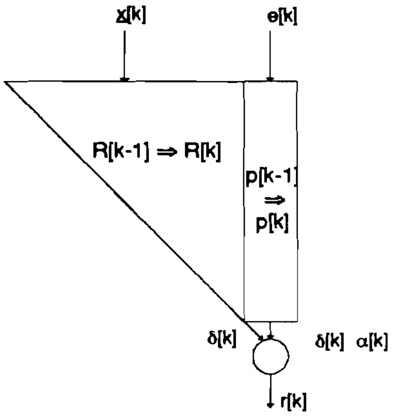


Figure 35: Possible RLS implementation

The implementation that is given by Haykin (see [7]) is a rewritten version of the above algorithm so that there are no square roots (which need a large calculation time) necessary and the last multiplication is not necessary any more. This results in the implementation given in figure 14. Comparing this implementation with the implementation of figure 35 it can be seen that the Processing Elements (PE's) labelled d and r calculate the matrix R[k] and that the PE's labelled u calculate the vector p[k].

The PE's labelled d are called the angle computers and calculate the Givens rotations of the matrix. An outline of an angle computer is given in figure 36. Each of these PE's does the following calculations:

$$d[k] = \gamma^2 \cdot d[k-1] + \delta_{in}[k] \cdot | x_{in}[k] |^2$$
$$s[k] = \delta_{in}[k] \cdot (z[k]/d[k])$$
$$z[k] = x_{in}[k]$$
$$\delta_{out}[k] = \delta_{in}[k] \gamma^2 \cdot (d[k-1]/d[k])$$

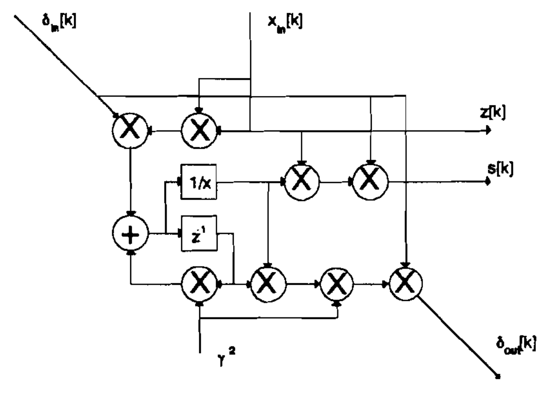


Figure 36: Angle computer

The PE's labelled r and u are rotators and apply the Givens rotation to the matrix and the vector. An outline of a rotator is given in figure 37. Each of these PE's does the following calculations:

$x_{out}[k] = x_{in}[k] - z[k]\cdot r[k-1]$
$r[k] = r[k-1] + s[k]\cdot x_{out}[k]$

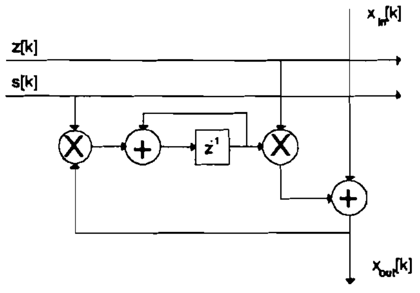


Figure 37: Rotator

Appendix C: LSL algorithm and systolic array implementation

In this appendix a Least Square Lattice is described. The implementation of the LSL filter is given in figure 16 and is proposed by Ling (see Ling[9]). The F and B elements calculate the reflection coefficients ($rf_i[k]$ and $rb_i[k]$) used by the lattice. The E element calculates the weights ($re_i[k]$). Notice that the weight coefficients are not equal to the weight coefficients in the FRLS algorithms because the estimated signal $\hat{e}[k]$ is calculated by $\underline{xb}^T[k]\cdot\underline{re}[k]$ in the LSL algorithms and by $\underline{x}^T[k]\cdot\underline{w}[k]$ in the FRLS algorithm. The calculations done by the 3 different elements are described by the following formula's:

The F-element calculates the forward reflection coefficient. An outline of this element is given in figure ?. This is done by the following formula's:

$$
\begin{aligned}
db_i[k] &= \gamma\cdot db_i[k-1] + \delta_i[k]\cdot|\,xb_i[k]\,|^2 \\
cb_i[k] &= \gamma\cdot db_i[k-1]/db_i[k] \\
sb_i[k] &= \delta_i[k]\cdot xb_i[k]/db_i[k] \\
rf_i[k] &= cb_i[k-1]\cdot rf_i[k-1] + sb_i[k-1]\cdot xf_i[k]
\end{aligned}
$$

Parallel implementation of this element results in the following quantaties:

$$
\begin{aligned}
C &= 9 \\
L &= 0 \\
\Delta T &= 4
\end{aligned}
$$

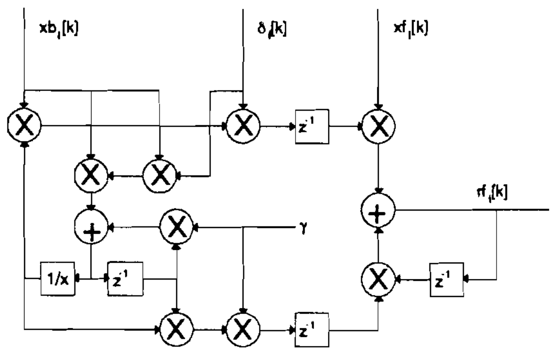


Figure 38: F-element

The B-element calculates the backward reflection coefficient. An outline of this element is given in figure ?. This is done by the following formula's:

$$df_i[k] = \gamma \cdot df_i[k-1] + \delta_i[k-1] \cdot |xf_i[k]|^2$$
$$cf_i[k] = \gamma \cdot df_i[k-1]/df_i[k]$$
$$sf_i[k] = \delta_i[k-1] \cdot xf_i[k]/df_i[k]$$
$$rb_i[k] = cf_i[k] \cdot rb_i[k-1] + sf_i[k] \cdot xb_i[k-1]$$

Parallel implementation of this element results in the following quantaties:

$C = 9$
$L = 0$
$\Delta T = 4$

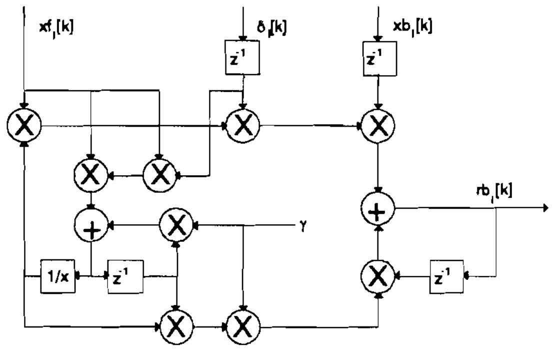


Figure 39: B-element

The E-element calculates the filter weights. An outline of this element is given in figure ?. This is done by the following formula's:

$$re_i[k] = cb_i[k] \cdot re_i[k-1] + sb_i[k] \cdot xe_i[k]$$

Parallel implementation of this element results in the following quantaties:

$C = 2$
$L = 0$
$\Delta T = 1$

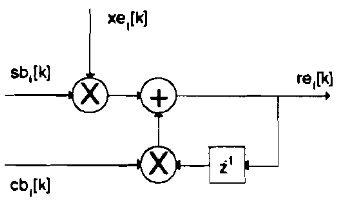


Figure 40: E-element