8. Adaptive Filters

The filters we have discussed so far had been designed for applications where the requirements for the “optimal” coefficients did not change over time, i.e., they were LTI systems. However, many real-world signals we find in typical DSP fields like speech processing, communications, radar, sonar, seismology, or biomedicine, require that the “optimal” filter or system coefficients need to be adjusted over time depending on the input signal. If the parameter changes slowly compared with the sampling frequency we can compute a “better” estimation for our optimal coefficients and adjust the filter appropriate.

In general, any filter structure, FIR or IIR, with the many architectural variations we have discussed before, may be used as an adaptive digital filter (ADF). Comparing the different structural options, we note that:

- For FIR filters the direct form from Fig. 3.1 (p. 166) seems to be advantageous because the coefficient update can be done at the same time instance for all coefficients.
- For IIR filters the lattice structure shown in Fig. 4.12 (p. 227) seems to be a good choice because lattice filters possess a low fixed-point arithmetic roundoff error sensitivity and a simplified stability control of the coefficients.

From the published literature, however, it appears that FIR filters have been used more successfully than IIR filters and our focus in this chapter will therefore be efficient and fast implementation of adaptive FIR filters.

The FIR filter algorithms should converge to the optimum nonrecursive estimator solution given (originally for continuous signal) through the Wiener–Hopf equation [251]. We will then discuss the optimum recursive estimator (Kalman filter). We will compare the different options in terms of computational complexity, stability of the algorithms, initial speed of convergence, consistency of convergence, and robustness to additive noise.

Adaptive filters can now be seen to be a mature DSP field. Many books in their first edition had been published in the mid-1980s and can be used for a more in-depth study [252, 253, 254, 255, 256, 257]. More recent results may be found in textbook like [258, 259, 260]. Recent journal publications like IEEE Transactions on Signal Processing show, especially in the area of stability of LMS and its variations, essential research activity.
8.1 Application of Adaptive Filter

Although the application fields of adaptive filters are quite broad in nature, they can usually be described with one of the following four system configurations:

- Interference cancellation
- Prediction
- Inverse modeling
- Identification

We wish to discuss in the following the basic idea of these systems and present some typical successful applications for these classes. Although it may not always exactly describe the nature of the specific signals it is common to use the following notation for all systems, namely

\[ x = \text{input to the adaptive filter} \]
\[ y = \text{output of the adaptive filter} \]
\[ d = \text{desired response (of the adaptive filter)} \]
\[ e = d - y = \text{estimation error} \]

8.1.1 Interference Cancellation

In these very popular applications of the adaptive filter the incoming signal contains, beside the information-bearing signal, also an interference, which may, for example, be a random white noise or the 50/60 Hz power-line hum. Figure 8.1 shows the configuration for this application. The incoming (sensor) signal \( d[n] \) and the adaptive filter output response \( y[n] \) to a reference signal \( x[n] \) is used to compute the error signal \( e[n] \), which is also the system output in the interference cancellation configuration. Thus, after convergence, the (modified) reference signal, which will represent the additive inverse of the interference is subtracted from the incoming signal.

![Fig. 8.1. Basic configuration for interference cancellation.](image)
We will later study a detailed example of the interference cancellation of the power-line hum. A second popular application is the adaptive noise cancellation of echoes on telephone systems. Interference cancellation has also been used in an array of antennas (called beamformer) to adaptively remove noise interfering from unknown directions.

8.1.2 Prediction

In the prediction application the task of the adaptive filter is to provide a best prediction (usually in the least mean square sense) of a present value of a random signal. This is obviously only possible if the input signal is essential different from white noise. Prediction is illustrated in Fig. 8.2. It can be seen that the input \( d[n] \) is applied over a delay to the adaptive filter input, as well as to compute the estimation error.

The predictive coding has been successfully used in image and speech signal processing. Instead of coding the signal directly, only the prediction error is encoded for transmission or storage. Other applications include the modeling of power spectra, data compression, spectrum enhancement, and event detection [253].

8.1.3 Inverse Modeling

In the inverse modeling structure the task is to provide an inverse model that represents the best fit (usually in the least squares sense) to an unknown time-varying plant. A typical communication example would be the task to estimate the multipath propagation of the signal to approximate an ideal transmission. The system shown in Fig. 8.3 illustrates this configuration. The input signal \( d[n] \) enters the plant and the output of the unknown plant \( x[n] \) is the input to the adaptive filter. A delayed version of the input \( d[n] \) is then used to compute the error signal \( e[n] \) and to adjust the filter coefficients of the adaptive filter. Thus, after convergence, the adaptive filter transfer function approximates the inverse of the transfer function of the unknown plant.

Besides the already-mentioned equalization in communication systems, inverse modeling with adaptive filters has been successfully used to improve

![Fig. 8.2. Block diagram for prediction.](image-url)
S/N ratio for additive narrowband noise, for adaptive control systems, in speech signal analysis, for deconvolution, and digital filter design [253].

8.1.4 Identification

In a system identification application the task is that the filter coefficients of the adaptive filter represent an unknown plant or filter. The system identification is shown in Fig. 8.4 and it can be seen that the time series, \( x[n] \), is input simultaneously to the adaptive filter and another linear plant or filter with unknown transfer function. The output of the unknown plant \( d[n] \) becomes the output of the entire system. After convergence the adaptive filter output \( y[n] \) will approximate \( d[n] \) in an optimum (usually least mean squares) sense. Provided that the order of the adaptive filter matches the order of the unknown plant and the input signal \( x[n] \) is WSS the adaptive filter coefficients will converge to the same values as the unknown plant. In a practical application there will normally be an additive noise present at the output of the unknown plant (observation errors) and the filter structure will not exactly match that of the unknown plant. This will result in deviation from the perfect performance described. Due to the flexibility of this structure and the ability to individually adjust a number of input parameters independently it is one of the structures often used in the performance evaluations of adaptive filters. We will use these configurations to make a detailed comparison between LMS and RLS, the two most popular algorithms to adjust the filter coefficient of an adaptive filter.

Such system identification has been used for modeling in biology, or to model social and business systems, for adaptive control systems, digital filter design, and in geophysics [253]. In a seismology exploration, such systems have been used to generate a layered-earth model to unravel the complexities of the earth’s surface [252].

8.2 Optimum Estimation Techniques

Required signal properties. In order to use successfully the adaptive filter algorithms presented in the following and to guarantee the convergence and
stability of the algorithms, it is necessary to make some basic assumptions about the nature of our input signals, which from a probabilistic standpoint, can be seen as a vector of random variables. First, the input signal (i.e., the random variable vector) should be **ergodic**, i.e., statistical properties like mean

$$\eta = E\{x\} = \lim_{N \to \infty} \frac{1}{N} \sum_{n=0}^{N-1} x[n]$$

or variance

$$\sigma^2 = E\{x^2\} = \lim_{N \to \infty} \frac{1}{N} \sum_{n=0}^{N-1} (x[n] - \eta)^2$$

computed using a single input signal should show the same statistical properties like the average over an ensemble of such random variables. Secondly, the signals need to be wide sense stationary (WSS), i.e., statistics measurements like average or variance measured over the ensemble averages are not a function of the time, and the autocorrelation function

$$r[\tau] = E\{x[t_1]x[t_2]\} = E\{x[t + \tau]x[t]\}$$

$$= \lim_{N \to \infty} \frac{1}{N} \sum_{n=0}^{N-1} x[n]x[n + \tau]$$

depends only on the difference $\tau = t_1 - t_2$. We note in particular that

$$r[0] = E\{x[t]x[t]\} = E\{|x[t]|^2\} \tag{8.1}$$

computes the average power of the WSS process.

**Definition of cost function.** The definition of the cost function applied to the estimator output is a critical parameter in all adaptive filter algorithms. We need to “weight” somehow the estimation error

$$e[n] = d[n] - y[n], \tag{8.2}$$

![Fig. 8.4. Basic configuration for identification.](image-url)
where $d[n]$ is the random variable to be estimated, and $y[n]$ is the computed estimate via the adaptive filter. The most commonly used cost function is the least-mean-squares (LMS) function given as

$$J = E\{e^2[n]\} = (d[n] - y[n])^2.$$  

(8.3)

It should be noted that this is not the only cost function that may be used. Alternatives are functions such as the absolute error or the nonlinear threshold functions as shown in Fig 8.5 on the right. The nonlinear threshold type may be used if a certain error level is acceptable and as we will see later can reduce the computational burden of the adaptation algorithm. It may be interesting to note that the original adaptive filter algorithms by Widrow [255] uses such a threshold function for the error.

On the other hand, the quadratic error function of the LMS method will enable us to build a stochastic gradient approach based on the Wiener–Hopf relation originally developed in the continuous signal domain. We review the Wiener–Hopf estimation in the next subsection, which will directly lead to the popular LMS adaptive filter algorithms first proposed by Widrow et al. [261, 262].

### 8.2.1 The Optimum Wiener Estimation

The output of the adaptive FIR filter is computed via the convolution sum

$$y[n] = \sum_{k=0}^{L-1} f_k x[n-k],$$  

(8.4)

where the filter coefficients $f_k$ have to be adjusted in such a way that the defined cost function $J$ is minimum. It is, in general, more convenient to write the convolution with vector notations according to

$$y[n] = x^T[n] f = f^T x[n],$$  

(8.5)
with \( \mathbf{f} = [f_0 f_1 \ldots f_{L-1}]^T \), \( \mathbf{x}[n] = [x[n]x[n-1] \ldots x[n-(L-1)]]^T \), are size \((L \times 1)\) vectors and \(T\) means matrix transposition or the Hermitian transposition for complex data. For \( \mathbf{A} = [a[k, l]] \) the transposed matrix is “mirrored” at the main diagonal, i.e., \( \mathbf{A}^T = [a[l, k]] \). Using the definition of the error function (8.2) we get

\[
e[n] = d[n] - y[n] = d[n] - \mathbf{f}^T \mathbf{x}[n].
\] (8.6)

The mean square error function now becomes

\[
J = E\{e^2[n]\} = E\{d[n] - y[n]\}^2 = E\{d[n] - \mathbf{f}^T \mathbf{x}[n]\}^2
= E\{(d[n] - \mathbf{f}^T \mathbf{x}[n])(d[n] - \mathbf{x}^T[n]\mathbf{f})\}
= E\{d[n]^2 - 2d[n]\mathbf{f}^T \mathbf{x}[n] + \mathbf{f}^T \mathbf{x}[n]\mathbf{x}^T[n]\mathbf{f}\}.
\] (8.7)

Note that the error is a quadratic function of the filter coefficients that can be pictured as a concave hyperparaboloidal surface, a function that never goes negative, see Fig. 8.6 for an example with two filter coefficients. Adjusting the filter weights to minimize the error involves descending along this surface with the objective of getting to the bottom of the bowl. Gradient methods are commonly used for this purpose. The choice of mean square type of cost function will enable a well-behaved quadratic error surface with a single unique minimum. The cost is minimum if we differentiate (8.7) with respect to \( \mathbf{f} \) and set this gradient to zero, i.e.,

\[
\nabla = \frac{\partial J}{\partial \mathbf{f}^T} = E \{( -2d[n]\mathbf{x}[n] + 2\mathbf{x}^T[n]\mathbf{x}[n]\mathbf{f}_{opt} \} = 0.
\]

Assuming that the filter weight vector \( \mathbf{f} \) and the signal vector \( \mathbf{x}[n] \) are statistically independent (i.e., uncorrelated), it follows, that

\[
E\{d[n]\mathbf{x}[n]\} = E\{\mathbf{x}[n]\mathbf{x}^T[n]\}\mathbf{f}_{opt},
\]

then the optimal filter coefficient vector \( \mathbf{f}_{opt} \) can be computed with,

\[
\mathbf{f}_{opt} = E\{\mathbf{x}[n]\mathbf{x}^T[n]\}^{-1}E\{d[n]\mathbf{x}[n]\}. \quad \text{(8.8)}
\]

The expectation terms are usually defined as follows:

\[
\mathbf{R}_{xx} = E\{\mathbf{x}[n]\mathbf{x}^T[n]\}
\]

\[
= \begin{bmatrix}
  x[n]x[n] & x[n]x[n-1] & \cdots & x[n]x[n-(L-1)] \\
  x[n-1]x[n] & x[n-1]x[n-1] & \cdots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
  x[n-(L-1)]x[n] & \cdots & & \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  r[0] & r[1] & \cdots & r[L-1] \\
  r[1] & r[0] & \cdots & r[L-2] \\
  \vdots & \vdots & \ddots & \vdots \\
  r[L-1] & r[L-2] & \cdots & r[0] \\
\end{bmatrix}
\]

\[
\mathbf{R}_{xx} = E\{\mathbf{x}[n]\mathbf{x}^T[n]\}
\]

\[
= \begin{bmatrix}
  x[n]x[n] & x[n]x[n-1] & \cdots & x[n]x[n-(L-1)] \\
  x[n-1]x[n] & x[n-1]x[n-1] & \cdots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
  x[n-(L-1)]x[n] & \cdots & & \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  r[0] & r[1] & \cdots & r[L-1] \\
  r[1] & r[0] & \cdots & r[L-2] \\
  \vdots & \vdots & \ddots & \vdots \\
  r[L-1] & r[L-2] & \cdots & r[0] \\
\end{bmatrix}
\]
is the \((L \times L)\) autocorrelation matrix of the input signal sequence, which has the form of the Toeplitz matrix, and

\[
\mathbf{r}_{dx} = E\{d[n]x[n]\}
\]

\[
= E \begin{bmatrix}
d[n]x[n] \\
d[n]x[n-1] \\
\vdots \\
d[n]x[n-(L-1)]
\end{bmatrix}
= \begin{bmatrix}
r_{dx}[0] \\
r_{dx}[1] \\
\vdots \\
r_{dx}[L-1]
\end{bmatrix}
\]

is the \((L \times 1)\) cross-correlation vector between the desired signal and the reference signal. With these definitions we can now rewrite (8.8) more compactly as

\[
f_{\text{opt}} = \mathbf{R}_{xx}^{-1}\mathbf{r}_{dx}.
\]  

(8.9)

This is commonly recognized as the Wiener–Hopf equation [251], which yields the optimum LMS solution for the filter coefficient vector \(f_{\text{opt}}\). One requirement to have a unique solution for (8.9) is that \(\mathbf{R}_{xx}^{-1}\) exist, i.e., the autocorrelation matrix must be nonsingular, or put differently, the determinant is nonzero. Fortunately, it can be shown that for WSS signals the \(\mathbf{R}_{xx}\) matrix is nonsingular [252, p. 41] and the inverse exists.
Using (8.7) the residue error of the optimal estimation becomes:

\[
J_{\text{opt}} = E\{d[n] - f_{\text{opt}}^T x[n]\}^2
= E\{d[n]\}^2 - 2 f_{\text{opt}}^T r_{dx} + f_{\text{opt}}^T \underbrace{R_{xx}}_{r_{dx}} f_{\text{opt}}
\]

\[
J_{\text{opt}} = r_{dd}[0] - f_{\text{opt}}^T r_{dx}, \tag{8.10}
\]

where \(r_{dd}[0] = \sigma_d^2\) is the variance of \(d\).

We now wish to demonstrate the Wiener–Hopf algorithm with the following example.

**Example 8.1: Two-tap FIR Filter Interference Cancellation**

Suppose we have an observed communication signal that consists of three components: The information-bearing signal, which is a Manchester encoded sensor signal \(m[n]\) with amplitude \(B = 10\), shown in Fig. 8.7a; an additive white Gaussian noise \(n[n]\), shown in Fig. 8.7b; and a 60-Hz power-line hum interference with amplitude \(A = 50\), shown in Fig. 8.7c. Assuming the sampling frequency is 4 times the power-line hum frequency, i.e., \(4 \times 60 = 240\) Hz, the observed signal can therefore be formulated as follows.
d[n] = A \cos[\pi n/2] + Bm[n] + \sigma^2 n[n].

The reference signal x[n] (shown in Fig. 8.7d), which is applied to the adaptive filter input, is given as

x[n] = \cos[\pi n/2 + \phi],

where \phi = \pi/6 is a constant offset. The two-tap filter then has the following output:

x[n] = f_0 \cos \left[ \frac{\pi}{2} n + \phi \right] + f_1 \cos \left[ \frac{\pi}{2} (n - 1) + \phi \right].

To solve (8.9) we compute first the autocorrelation for x[n] with delays 0 and 1:

r_{xx}[0] = E\{(\cos[\pi n/2 + \phi])^2\} = \frac{1}{2}

r_{xx}[1] = E\{\cos[\pi n/2 + \phi] \sin[\pi n/2 + \phi]\} = 0.

For the cross-correlation we get

r_{dx}[0] = E \left\{ (A \cos[\pi n/2] + Bm[n] + \sigma^2 n[n]) \cos[\pi n/2 + \phi] \right\}
= \frac{A}{2} \cos(\phi) = 12.5 \sqrt{3}

r_{dx}[1] = E \left\{ (A \cos[\pi n/2] + Bm[n] + \sigma^2 n[n]) \sin[\pi n/2 + \phi] \right\}
= \frac{A}{2} \cos(\phi - \pi) = \frac{50}{4} = 12.5.

As required for the Wiener–Hopf equation (8.9) we can now compute the (2 x 2) autocorrelation matrix and the (2 x 1) cross-correlation vector and get

\[ f_{opt} = R_{xx}^{-1} r_{dx} \begin{bmatrix} r_{xx}[0] & r_{xx}[1] \\ r_{xx}[1] & r_{xx}[0] \end{bmatrix}^{-1} \begin{bmatrix} r_{dx}[0] \\ r_{dx}[1] \end{bmatrix} \]

\[ = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{-1} \begin{bmatrix} 12.5 \sqrt{3} \\ 12.5 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 12.5 \sqrt{3} \\ 12.5 \end{bmatrix} = \begin{bmatrix} 43.3 \\ 25 \end{bmatrix}. \]

The simulation of these data is shown in Fig. 8.8. It shows (a) the sum of the three signals (Manchester-coded 5 bits, power-line hum of 60 Hz, and the additive white Gaussian noise) and the system output (i.e., e[n]) with the canceled power-line hum.
There may exist a couple of reasons why we wish to avoid a direct computation of the Wiener estimation (8.9). First, the generation of the autocorrelation matrix $R_{xx}$ and the cross-correlation vector $r_{dx}$ are already computationally intensive. We need to compute the autocorrelation of $x$ and the cross-correlation between $d$ and $x$ and we may, for instance, not know how many data samples we need to use in order to have sufficient statistics. Secondly, if we have constructed the correlation functions we still have to compute the inverse of the autocorrelation matrix $R_{xx}^{-1}$, which can be very time consuming, if the filter order gets larger. Even if a procedure is available to invert $R_{xx}$, the precision of the result may not be sufficient because of the many computational steps involved, especially with a fixed-point arithmetic implementation.

The Widrow–Hoff least mean square (LMS) adaptive algorithm [261] is a practical method for finding a close approximation to (8.9) in real time. The algorithm does not require explicit measurement of the correlation functions, nor does it involve matrix inversion. Accuracy is limited by statistical sample size, since the filter coefficient values are based on the real-time measurements of the input signals.

The LMS algorithm is an implementation of the method of the steepest descent. According to this method, the next filter coefficient vector $f[n+1]$ is equal to the present filter coefficient vector $f[n]$ plus a change proportional to the negative gradient:

$$f[n+1] = f[n] - \frac{\mu}{2} \nabla[n]. \quad (8.11)$$

The parameter $\mu$ is the learning factor or step size that controls stability and the rate of convergence of the algorithm. During each iteration the true gradient is represented by $\nabla[n]$. 

Fig. 8.8. Canceling 60-Hz power-line interference of a Manchester-coded data signal using optimum Wiener estimation.
8. Adaptive Filters

The LMS algorithm estimates an instantaneous gradient in a crude but efficient manner by assuming that the gradient of \( J = e[n]^2 \) is an estimate of the gradient of the mean-square error \( E\{e[n]^2\} \). The relationship between the true gradient \( \nabla[n] \) and the estimated gradients \( \hat{\nabla}[n] \) is given by the following expression:

\[
\nabla[n] = \left[ \frac{\partial E\{e[n]^2\}}{\partial f_0}, \frac{\partial E\{e[n]^2\}}{\partial f_1}, \ldots, \frac{\partial E\{e[n]^2\}}{\partial f_{L-1}} \right]^T \tag{8.12}
\]

\[
\hat{\nabla}[n] = \left[ \frac{\partial e[n]^2}{\partial f_0}, \frac{\partial e[n]^2}{\partial f_1}, \ldots, \frac{\partial e[n]^2}{\partial f_{L-1}} \right]^T = 2e[n] \left[ \frac{\partial e[n]}{\partial f_0}, \frac{\partial e[n]}{\partial f_1}, \ldots, \frac{\partial e[n]}{\partial f_{L-1}} \right]^T. \tag{8.13}
\]

The estimated gradient components are related to the partial derivatives of the instantaneous error with respect to the filter coefficients, which can be obtained by differentiating (8.6), it follows that

\[
\hat{\nabla}[n] = -2e[n] \frac{\partial e[n]}{\partial f} = -2e[n]x[n]. \tag{8.14}
\]

Using this estimate in place of the true gradient in (8.11) yields:

\[
f[n+1] = f[n] - \mu e[n]x[n]. \tag{8.15}
\]

Let us summarize all necessary step for the LMS algorithm in the following

**Algorithm 8.2: Widrow–Hoff LMS Algorithm**

The Widrow–Hoff LMS algorithm to adjust the L filter coefficients of an adaptive uses the following steps:

1) Initialize the \((L \times 1)\) vector \( f = x = 0 = [0, 0, \ldots, 0]^T \).
2) Accept a new pair of input samples \( \{x[n], d[n]\} \) and shift \( x[n] \) in the reference signal vector \( x[n] \).
3) Compute the output signal of the FIR filter, via
\[
y[n] = f^T[n]x[n]. \tag{8.16}
\]
4) Compute the error function with
\[
e[n] = d[n] - y[n]. \tag{8.17}
\]
5) Update the filter coefficients according to
\[
f[n+1] = f[n] + \mu e[n]x[n]. \tag{8.18}
\]

Now continue with step 2.

Although the LMS algorithm makes use of gradients of mean-square error functions, it does not require squaring, averaging, or differentiation. The al-

---

Note that in the original presentation of the algorithm [261] the update equation \( f[n+1] = f[n] + 2\mu e[n]x[n] \) is used because the differentiation of the gradient in (8.14) produces a factor 2. The update equation (8.15) follows the notation that is used in most of the current textbooks on adaptive filters.
8.3 The Widrow–Hoff Least Mean Square Algorithm

Algorithm is simple and generally easy to implement in software (MATLAB code see, for instance, [260, p. 332]; C code [263], or PDSP assembler code [264]).

A simulation using the same system configuration as in Example 8.1 (p. 485) is shown in Fig. 8.9 for different values of the step size $\mu$. Adaptation starts after 1 second. System output $e[n]$ is shown in the left column and the filter coefficient adaptation on the right. We note that depending on the value $\mu$ the optimal filter coefficients approach $f_0 = 43.3$ and $f_1 = 25$.

It has been shown that the gradient estimate used in the LMS algorithm is unbiased and that the expected value of the weight vector converges to the Wiener weight vector (8.9) when the input signals are WSS, which was anyway required in order to be able to compute the inverse of the autocorrelation matrix $R_{xx}^{-1}$ for the Wiener estimate. Starting with an arbitrary initial filter coefficient vector, the algorithm will converge in the mean and will remain stable as long as the learning parameter $\mu$ is greater than 0 but less than an upper bound $\mu_{\text{max}}$. Figure 8.10 shows an alternative form to...
Fig. 8.10. Demonstration of the convergence of the power-line interference example using a 2D contour plot for $\mu = 1/16$.

represent the convergence of the filter coefficient adaptation by a projection of the coefficient values in a $(f_0, f_1)$ mapping. The figure also shows the contour line with equal error. It can be seen that the LMS algorithm moves in a zigzag way towards the minimum rather than the true gradient, which would move exactly orthogonal to these error contour lines.

Although the LMS algorithm is considerably simpler than the RLS algorithm (we will discuss this later) the convergence properties of the LMS algorithm are nonetheless difficult to analysis rigorously. The simplest approach to determine an upper bound of $\mu$ makes use of the eigenvalues of $R_{xx}$ by solving the homogeneous equation

$$0 = \det(\lambda I - R_{xx}),$$

where $I$ is the $L \times L$ identity matrix. There are $L$ eigenvalues $\lambda[k]$ that have the following properties

$$\det(R_{xx}) = \prod_{k=0}^{L-1} \lambda[k] \quad \text{and} \quad \text{trace}(R_{xx}) = \sum_{k=0}^{L-1} \lambda[k].$$

(8.20) (8.21)
8.3 The Widrow–Hoff Least Mean Square Algorithm 491

From this eigenvalue analysis of the autocorrelation matrix it follows that
the LMS algorithm will be stable (in the mean sense) if

$$0 < \mu < \frac{2}{\lambda_{\text{max}}}.$$  \hfill (8.22)

Although the filter is assumed to be stable, we will see later that this upper
bound will not guarantee a finite mean square error, i.e., that $$f[n]$$ converges
to $$f_{\text{opt}}$$ and a much more stringent bound has to be used.

The simulation of the LMS algorithm in Fig. 8.9 also reveals the underly-
ing exponential nature of the individual learning curves. Using the eigenvalue
analysis we may also transform the filter coefficient in independent so-called
“modes” that are no longer linear dependent. The number of natural modes
is equal to the number of degrees of freedom, i.e., the number of independent
components and in our case identically with the number of filter coefficients.
The time constant of the $$k^{th}$$ mode is related to the $$k$$ eigenvalue $$\lambda[k]$$ and the
parameter $$\mu$$ by

$$\tau[k] = \frac{1}{2\mu \lambda[k]}.$$ \hfill (8.23)

Hence the longest time constant, $$\tau_{\text{max}}$$, is associated with the smallest eigen-
value, $$\lambda_{\text{min}}$$ via

$$\tau_{\text{max}} = \frac{1}{2\mu \lambda_{\text{min}}}.$$ \hfill (8.24)

Combining (8.22) and (8.24) gives

$$\tau_{\text{max}} > \frac{\lambda_{\text{max}}}{2\lambda_{\text{min}}},$$ \hfill (8.25)

which suggests that the larger the eigenvalue ratio (EVR), $$\lambda_{\text{max}}/\lambda_{\text{min}}$$ of the
autocorrelation matrix $$R_{xx}$$ the longer the LMS algorithm will take to con-
verge. Simulation results that confirm this finding can be found for instance,
in [259, p. 64] and will be discussed, in Sect. 8.3.1 (p. 493).

The results presented so far on the ADF stability can be found in most
original published work by Widrow and many textbooks. However, these
conditions do not guarantee a finite variance for the filter coefficient vector,
neither do they guarantee a finite mean-square error! Hence, as many users
of the algorithm realized, considerably more stringent conditions are required
to ensure convergence of the algorithm. In the examples in [260, p. 130], for
instance, you find the “rule of thumb” that a factor 10 smaller values for $$\mu$$
should be used.

More recent results indicate that the bound from (8.22) must be more
restrictive. For example, the results presented by Horowitz and Senne [265]
and derived in a different way by Feuer and Weinstein [266] show that the
step size (assuming that the elements of the input vector $$x[n]$$ are statistically
independent) has to be restricted via the two conditions:
8. Adaptive Filters

Fig. 8.11. Simulation of the power-line interference cancellation using the maximum step size values for the LMS algorithm. (left) System output $e[n]$. (right) Filter coefficients.

\[ 0 < \mu < \frac{1}{\lambda_l} \quad l = 0, 1, \ldots, L - 1 \quad \text{and} \]

\[ \sum_{l=0}^{L-1} \frac{\mu \lambda_l}{1 - \mu \lambda_l} < 2, \]  

(8.26) 

(8.27)

to ensure convergence. These conditions can not be solved analytically, but it can be shown that they are closely bounded by the following condition:

\[ 0 < \mu < \frac{2}{3 \times \text{trace}(R_{xx})} = \frac{2}{3 \times L \times r_{xx}[0]}. \]  

(8.28)

The upper bound of (8.28) has a distinct practical advantage. Trace of $R_{xx}$ is, by definition, (see (8.21), p. 490) the total average input signal power of the reference signal, which can easily be estimated from the reference signal $x[n]$.

Example 8.3: Bounds on Step Size
From the analysis in (8.22) we see that we first need to compute the eigenvalues of the $R_{xx}$ matrix, i.e.

$$0 = \det(\lambda I - R_{xx}) = \begin{bmatrix} r_{xx}[0] - \lambda & r_{xx}[1] \\ r_{xx}[1] & r_{xx}[0] - \lambda \end{bmatrix}$$

(8.29)

$$= \det \begin{bmatrix} 0.5 - \lambda & 0 \\ 0 & 0.5 - \lambda \end{bmatrix} = (0.5 - \lambda)^2$$

(8.30)

Using (8.22) gives

$$\lambda[1, 2] = 0.5.$$  

(8.31)

Using the more restrictive bound from (8.28) yields

$$\mu_{\text{max}} = \frac{2}{\lambda_{\text{max}}} = 4.$$  

(8.32)

The simulation results in Fig. 8.11 indicate that in fact $\mu = 4$ does not show convergence, while $\mu = 2/3$ converges.

We also note from the simulation shown in Fig. 8.11 that even with $\mu_{\text{max}} = 2/3$ the convergence is much faster, but the coefficients “ripple around” essentially. Much smaller values for $\mu$ are necessary to have a smooth approach of the filter coefficient to the optimal values and to stay there.

The condition found by Horowitz and Senne [265] and Feuer and Weinstein [266] made the assumption that all inputs $x[n]$ are statistically independent. This assumption is true if the input data come, for instance, from an antenna array of $L$ independent sensors, however, for ADFs with the tapped delay structure, it has been shown, for instance, by Butterweck [252], that for a long filter the stability bound can be relaxed to

$$0 < \mu < \frac{2}{L \times r_{xx}[0]},$$

(8.34)

i.e., compared with (8.28) the upper bound can be relaxed by a factor of 3 in the denominator. But the condition (8.34) only applies for a long filter and it may therefore save to use (8.28).

### 8.3.1 Learning Curves

Learning curve, i.e., the error function $J$ displayed over the number of iterations is an important measurement instrument when comparing the performance of different algorithms and system configurations. We wish in the following to study the LMS algorithm regarding the eigenvalue ratio $\lambda_{\text{max}}/\lambda_{\text{min}}$ and the sensitivity to signal-to-noise (S/N) ratio in the system to be identified.
A typical performance measurement of adaptive algorithms using a system-identification problem is displayed in Fig. 8.12. The adaptive filter has a length of $L = 16$ the same length as the “unknown” system, whose coefficients have to be learned. The additive noise level behind the “unknown” system has been set to two different levels $-10$ dB for a high-noise environment and to $-48$ dB for a low-noise environment equivalent to an 8-bit quantization.
Table 8.1. Four different noise-shaping FIR filters to generate power-of-ten eigenvalue ratios for $L = 16$.

<table>
<thead>
<tr>
<th>No.</th>
<th>Impulse response</th>
<th>EVR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$0 + 1z^{-1} + 0.0z^{-2}$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$0.247665 + 0.936656z^{-1} + 0.247665z^{-2}$</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>$0.577582 + 0.576887z^{-1} + 0.577582z^{-2}$</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>$0.432663 + 0.790952z^{-1} + 0.432663z^{-2}$</td>
<td>1000</td>
</tr>
</tbody>
</table>

For the LMS algorithm the eigenvalue ratio (EVR) is the critical parameter that determines the convergence speed, see (8.25), p. 491. In order to generate a different eigenvalue ratio we use a white Gaussian noise source with $\sigma^2 = 1$ that is shaped by a digital filter. We may, for instance, use a first-order IIR filter that generates a first-order Markov process, see Exercise 8.10 (p. 533). We may alternatively filter the white noise by a three-tap symmetrical FIR filter whose coefficients are $c^T = [a, b, a]$. The FIR filter has the advantage that we can easily normalize the power. The coefficients should be normalized $\sum_k c_k^2 = 1$ in such a way that input and output sequences have the same power. This requires that

$$1 = a^2 + b^2 + a^2 \quad \text{or} \quad a = 0.5 \times \sqrt{1 - b^2}.$$  \hspace{1cm} (8.35)

With this filter it is possible to generate different eigenvalue ratios $\lambda_{\text{max}}/\lambda_{\text{min}}$ as shown in Fig. 8.13 for different system size $L = 2, 4, 8, \text{ and } 16$. We can now use Table 8.1 to get power-of-ten EVRs for the system of length $L = 16$.

For a white Gaussian source the $R_{xx}$ matrix is a diagonal matrix $\sigma^2 I$ and the eigenvalues are therefore all one, i.e., $\lambda_l = 1; l = 0, 1, \ldots, L - 1$. The other EVRs can be verified with MatLab, see Exercise 8.9 (p. 533). The impulse response of the unknown system $g_k$ is an odd filter with coefficients $1, -2, 3, -4, \ldots, -3, 2, -1$ as shown in Fig. 8.14a. The step size for the LMS algorithm has been determined with

$$\mu_{\text{max}} = \frac{2}{3 \times L \times E\{x^2\}} = \frac{1}{24}.$$ \hspace{1cm} (8.36)

In order to guarantee perfect stability the step size has been chosen to be $\mu = \mu_{\text{max}}/2 = 1/48$. The learning curve, or coefficient error is computed via the normalized error function

$$J[n] = 20 \log_{10} \left( \frac{\sum_{k=0}^{15}(g_k - f_k[n])^2}{\sum_{k=0}^{15}g_k^2} \right).$$ \hspace{1cm} (8.37)

The coefficient adaptation for a single adaptation run with EVR=1 is shown in Fig. 8.14b. It can be seen that after 200 iterations the adaptive filter has learned the coefficient of the unknown system without an error. From the learning curves (average over 50 adaptation cycles) shown in Fig. 8.14c and d it can be seen that the LMS algorithm is very sensitive to the EVR. Many
Fig. 8.14. Learning curves for the LMS algorithm using the system identification configuration shown in Fig. 8.12. (a) Impulse response $g_k$ of the “unknown” system. (b) Coefficient learning over time. (c) Average over 50 learning curves for large system noise. (d) Average over 50 learning curves for small system noise.

iterations are necessary in the case of the high EVR. Unfortunately, many real-world signals have high EVR. Speech signals, for instance, may have EVR of 1874 [267]. On the other hand, we see from Fig. 8.14c that the LMS algorithm still adapts well in a high-noise environment.

8.3.2 Normalized LMS (NLMS)

The LMS algorithm discussed so far uses a constant step size $\mu$ proportional to the stability bound $\mu_{max} = 2/(L \times r_{xx}[0])$. Obviously this requires knowledge of the signal statistic, i.e., $r_{xx}[0]$, and this statistic must not change over time. It is, however, possible that this statistic changes over time, and we wish to adjust $\mu$ accordingly. These can be accomplished by computing a temporary estimate for the signal power via

$$r_{xx}[0] = \frac{1}{L} a^T[n] x[n], \quad (8.38)$$

and the “normalized” $\mu$ is given by
Fig. 8.15. Learning curves for the normalized LMS algorithm using the system identification configuration shown in Fig. 8.12. (a) The reference signal input $x[n]$ to the adaptive filter and the “unknown” system. (b) Coefficient learning over time for the normalized LMS. (c) Step size $\mu$ used for LMS and NLMS. (d) Average over 50 learning curves.

$$\mu_{\text{max}}[n] = \frac{2}{x^T[n]x[n]}.$$  (8.39)

If we are concerned that the denominator can temporary become very small and $\mu$ too large, we may add a small constant $\delta$ to $x^T[n]x[n]$, which yields

$$\mu_{\text{max}}[n] = \frac{2}{\delta + x^T[n]x[n]}.$$  (8.40)

To be on the safe side, we would not choose $\mu_{\text{max}}[n]$. Instead we would use a somewhat smaller value, like $0.5 \times \mu_{\text{max}}[n]$. The following example should demonstrate the normalized LMS algorithm.
Example 8.4: Normalized LMS

Suppose we have again the system identification configuration from Fig. 8.12 (p. 494), only this time the input signal $x[n]$ to the adaptive filter and the “unknown system” is the noisy pulse-amplitude-modulated (PAM) signal shown in Fig. 8.15a. For the conventional LMS we compute first $r_{xx}[0]$, and calculate $\mu_{\text{max}} = 0.0118$. The step size for the normalized LMS algorithm is adjusted depending on the momentary power $\sum x[n]^2$ of the reference signal. For the computation of $\mu_{\text{NLMS}}[n]$ shown in Fig. 8.15c it can be seen that at times when the absolute value of the reference signal is large the step size is reduced and for small absolute values of the reference signal, a larger step size is used. The adaptation of the coefficient displayed over time in Fig. 8.15b reflects this issue. Larger learning steps can be seen at those times when $\mu_{\text{NLMS}}[n]$ is larger. An average over 50 adaptations is shown in the learning curves in Fig. 8.15d. Although the EVR of the noisy PAM is larger than 600, it can be seen that the normalized LMS has a positive effect on the convergence behavior of the algorithm.

The power estimation using (8.38) is a precise power snapshot of the current data vector $x[n]$. It may, however, be desired to have a longer memory in the power computation to avoid a temporary small value and a large $\mu$ value. This can be accomplished using a recursive update of the previous estimations of the power, with

$$P[n] = \beta P[n-1] + (1 - \beta)|x[n]|^2,$$

(8.41)

with $\beta$ less than but close to 1. For a nonstationary signal such as the one shown in Fig. 8.15 the choice of the parameter $\beta$ must be done carefully. If we select $\beta$ too small the NLMS will more and more have the performance of the original LMS algorithm, see Exercise 8.14 (p. 534).

8.4 Transform Domain LMS Algorithms

LMS algorithms that solve the filter coefficient adjustment in a transform domain have been proposed for two reasons. The goal of the fast convolution techniques [268] is to lower the computational effort, by using block update and transforming the convolution to compute the adaptive filter output and the filter coefficient adjustment in the transform domain with the help of a fast cyclic convolution algorithm. The second method that uses transform-domain techniques has the main goal to improve the adaptation rate of the LMS algorithm, because it is possible to find transforms that allow a “decoupling” of the modes of the adaptive filter [267, 269].

8.4.1 Fast-Convolution Techniques

Fast cyclic convolution using transforms like FFTs or NTTs can be applied to FIR filters. For the adaptive filter this leads to a block-oriented processing
of the data. Although we may use any block size, the block size is usually chosen to be twice the size of the adaptive filter length so that the time delay in the coefficient update becomes not too large. It is also most often from a computational effort a good choice. In the first step a block of $2L$ input values $x[n]$ are convolved via transform with the filter coefficients $f_L$, which produces $L$ new filter output values $y[n]$. These results are then used to compute $L$ error signals $e[n]$. The filter coefficient update is then done also in the transform domain, using the already transformed input sequence $x[n]$. Let us go through these block processing steps using a $L = 3$ example. We compute the three filter output signals in one block:

$$
\begin{align*}
y[n] &= f_0 x[n] + f_1 x[n-1] + f_2 x[n-2] \\
y[n+1] &= f_0 x[n+1] + f_1 x[n] + f_2 x[n-1] \\
y[n+2] &= f_0 x[n+2] + f_1 x[n+1] + f_2 x[n].
\end{align*}
$$

These can be interpreted as a cyclic convolution of

$$\{f_0, f_1, f_2, 0, 0, 0\} \star \{x[n+2], x[n+1], x[n], x[n-1], x[n-2], 0\}.$$ 

The error signals follow then with

$$\begin{align*}
e[n+2] &= d[n+2] - y[n+2].
\end{align*}$$

The block processing for the filter gradient $\nabla$ can now be written as

$$\begin{align*}
\nabla[n] &= e[n] x[n] & \nabla[n+1] &= e[n+1] x[n+1] \\
\nabla[n+2] &= e[n+2] x[n+2].
\end{align*}$$

The update for each individual coefficient is then computed with

$$\begin{align*}
\nabla_0 &= e[n] x[n] + e[n+1] x[n+1] + e[n+2] x[n+2] \\
\nabla_1 &= e[n] x[n-1] + e[n+1] x[n] + e[n+2] x[n-1] \\
\end{align*}$$

We again see that this is a cyclic convolution, only this time the input sequence $x[n]$ appears in reverse order

$$\{0, 0, 0, e[n], e[n+1], e[n+2]\} \star \{0, x[n-2], x[n-1], x[n], x[n+1], x[n+2]\}.$$ 

In the Fourier domain the reverse order in time yields that we need to compute the conjugate transform of $X$. The coefficient update then becomes

$$f[n+L] = f[n] + \frac{\mu_B}{L} \nabla[n]. \quad (8.42)$$

Figure 8.16 shows all the necessary steps, when using the FFT for the fast convolution.

From the stability standpoint the block delay in the coefficient is not uncritical. Feuer [270] has shown that the step size has to be reduced to
for a block update of $B$ steps each. If we compare this result with the result for $\mu_{\text{max}}$ from (8.28) page 492 we note that the values are very similar. Only for large block sizes $B \gg L$ will the change in $\mu_B$ have considerable impact. This reduces to (8.28) for a block size of $B = 1$. However, the time constant is measured in blocks of $L$ data and it follows that the largest time constant for the BLMS algorithm is $L$ times larger then the largest time constant associated with the LMS algorithm.

### 8.4.2 Using Orthogonal Transforms

We have seen in Sect. 8.3.1 (p. 493) that the LMS algorithm is highly sensitive to the eigenvalue ratio (EVR). Unfortunately, many real-world signals have high EVRs. Speech signals, for instance, may have EVR of 1874 [267]. But it is also well known that the transform-domain algorithms allow a “decoupling” of the mode of the signals. The Karhunen–Loéve transform (KLT) is the optimal method in this respect, but unfortunately not a real time option, see Exercise 8.11 (p. 534). Discrete cosine transforms (DCT) and fast Fourier transform (FFT), followed by other orthogonal transforms like Walsh,
Hadamard, or Haar are the next best choice in terms of convergence speed, see Exercise 8.13, (p. 534) [271, 272].

Let us try in the following to use this concept to improve the learning rate of the identification experiment presented in Sect. 8.3.1 (p. 493), where the adaptive filter has to “learn” an impulse response of an unknown 16-tap FIR filter, as shown in Fig. 8.12 (p. 494). In order to apply the transform techniques and still to monitor the learning progress we need to compute in addition to the LMS algorithm 8.2 (p. 488) the DCT of the incoming reference signal $x[n]$ as well as the IDCT of the coefficient vector $f_n$. In a practical application we do not need to compute the IDCT, it is only necessary to compute it once after we reach convergence. The following MATLAB code demonstrates the transform-domain DCT-LMS algorithm.

```matlab
for k = L:Iterations % adapt over full length
    x = [xin;x(1:L-1)]; % get new sample
    din = g'*x + n(k); % "unknown" filter output + AWGN
    z = dct(x); % LxL orthogonal transform
    y = f' * z; % transformed filter output
    err = din-y; % error: primary - reference
    f = f + err*mu.*z; % update weight vector
    fi = idct(f); % filter in original domain
    J(k-L+1) = J(k-L+1) + sum((fi-g).^2); % Learning curve
end
```

The effect of a transform $T$ on the eigenvalue spread can be computed via

$$R_{zz} = TR_{xx}T^H,$$

where the superscript $H$ denotes the transpose conjugate.

The only thing we have not considered so far is that the $L$ “modes” or frequencies of the transformed input signal $z[l]$ are now more or less statistically independent input vectors and the step size $\mu$ in the original domain may no longer be appropriate to guarantee stability, or allow fast convergence. In fact, the simulations by Lee and Un [272] show that if no power normalization is used in the transform domain then the convergence did not improve compared with the time-domain LMS algorithm. It is therefore reasonable to compute for these $L$ spectral components different step sizes according to the stability bound (8.28), p. 492, just using the power of the transform components:

$$\mu_{\text{max}}[k] = \frac{2}{3 \times L \times r_{zz,k}[0]} \quad \text{for} \quad k = 0, 1, \ldots, L - 1.$$  

The additional effort is now the computation of the power normalization of all $L$ spectral components. The MATLAB code above already includes a componentwise update via $\mu.*z$, where the $.*$ stands for the componentwise multiplication.
Fig. 8.17. Optimal step size for the DCT-LMS transform-domain algorithm using the system identification configuration shown in Fig. 8.12 (p. 494) for four different eigenvalue ratios. (a) Eigenvalue ratios of 1. (b) Eigenvalue ratios of 10. (c) Eigenvalue ratios of 100. (d) Eigenvalue ratios of 1000.

The adjustment in $\mu$ is somewhat similar to the normalized LMS algorithm we have discussed before. We may therefore use directly the power normalization update similar to (8.39) p. 496 for the frequency component. The effect of power normalization and transform $T$ on the eigenvalue spread can be computed via

$$ R_{zz} = \lambda^{-1} T R_{xx} T^H \lambda^{-1}, \quad (8.45) $$

where $\lambda^{-1}$ is a diagonal matrix that normalizes $R_{zz}$ in such a way that the diagonal elements all become 1, see [271].

Figure 8.17 shows the computed step sizes for four different eigenvalue ratios of the $L = 16$ FIR filter. For a pure Gaussian input all spectral components should be equal and the step size is almost the same, as can be seen from Fig. 8.17a. The other filter shapes the noise in such a way that the power of these spectral components is increased (decreased) and the step size has to be set to a lower (higher) value.

From Fig. 8.18 the positive effect on the performance of the DCT-LMS transform-domain approach can be seen. The learning converges, even for
very high eigenvalue ratios like 1000. Only the error floor and consistency of the error at $-48$ dB is not reached as well for high EVRs as for the lower EVRs.

One factor that must be considered in choosing the transform for real-time application algorithms is the computational complexity. In this respect, real transforms like DCT or DST transforms are superior to complex transform like the FFT, transforms with fast algorithms are better than the algorithms without. Integer transforms like Haar or Hadamard, that do not need multiplications at all, are desirable [271]. Lastly, we also need to take into account that the RLS (discussed later) is another alternative, which has, in general, a higher complexity than the LMS algorithm, but may be more efficient than a transform-domain filter approach and also yield as fast a convergence as the KLT-based LMS algorithm.

8.5 Implementation of the LMS Algorithm

We now wish to look at the task to implement the LMS algorithm with FPGAs. Before we can proceed with a HDL design, however, we need to ensure that quantization effects are tolerable. Later in this section we will
then try to improve the throughput by using pipelining, and we need to ensure then also that the ADF is still stable.

8.5.1 Quantization Effects

Before we can start to implement the LMS algorithm in hardware we need to ensure that the parameter and data are well in the “green” range. This can be done if we change the software simulation from full precision to the desired integer precision. Figure 8.19 shows the simulation for 8-bit integer data and \( \mu = 1/4, 1/8 \) and 1/16. Note that we can not choose \( \mu \) too small, otherwise we will no longer get convergence through the large scaling of the gradient \( e[n]x[n] \) with \( \mu \) in the coefficient update equation (8.18), p. 488. The smaller the step size \( \mu \) the more problem the algorithm has to converge to the optimal values \( f_0 = 43.3 \) and \( f_1 = 25 \). This is somehow a contrary requirement to the upper bound on \( \mu \) given through the stability requirement of the algorithm. It can therefore be necessary to add fractional bits to the system to overcome these two contradictions.

8.5.2 FPGA Design of the LMS Algorithm

A possible implementation of the algorithm represented as a signal flow graph is shown in Fig. 8.20. From a hardware implementation standpoint we note that we need one scaling for \( \mu \) and \( 2L \) general multipliers. The effort is therefore more than twice the effort of the programmable FIR filter as discussed in Chap. 3, Example 3.1 (p. 167).

We wish to study in the following the FPLD implementation of the LMS algorithm.

Example 8.5: Two-tap Adaptive LMS FIR Filter

The VHDL design\(^3\) for a filter with two coefficients \( f_0 \) and \( f_1 \) with a step size of \( \mu = 1/4 \) is shown in the following listing:

```vhdl
-- This is a generic LMS FIR filter generator
-- It uses W1 bit data/coefficients bits
LIBRARY lpm; -- Using predefined packages
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY fir_lms IS ------> Interface
GENERIC (W1 : INTEGER := 8; -- Input bit width
          W2 : INTEGER := 16; -- Multiplier bit width 2*W1
          L : INTEGER := 2 -- Filter length
          );
```

\(^3\) The equivalent Verilog code `fir_lms.v` for this example can be found in Appendix A on page 719. Synthesis results are shown in Appendix B on page 731.
Fig. 8.19. Simulation of the power-line interference cancellation using the LMS algorithm for integer data. (left) System output $e[n]$. (right) Filter coefficients.
Fig. 8.20. Signal flow graph of the LMS algorithm.

SIGNAL x, f : ARRAY_N1BIT; -- Coeff/Data arrays
SIGNAL p, xemu : ARRAY_N2BIT; -- Product arrays

BEGIN

dsx: PROCESS (d) -- 16 bit signed extension for input d
BEGIN
sxtd(7 DOWNTO 0) <= d;
FOR k IN 15 DOWNTO 8 LOOP
 sxtd(k) <= d(d'high);
END LOOP;
END PROCESS;

Store: PROCESS -------> Store these data or coefficients
BEGIN
WAIT UNTIL clk = '1';
d <= d_in;
x(0) <= x_in;
x(1) <= x(0);
f(0) <= f(0) + xemu(0)(15 DOWNTO 8); -- implicit
f(1) <= f(1) + xemu(1)(15 DOWNTO 8); -- divide by 2
END PROCESS Store;

MulGen1: FOR I IN 0 TO L-1 GENERATE
FIR: lpm_mult -- Multiply p(i) = f(i) * x(i);
GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W1,
LPM_REPRESENTATION => "SIGNED",
LPM_WIDTHP => W2,
LPM_WIDTHS => W2)
PORT MAP ( dataa => x(I), datab => f(I),
result => p(I));
END GENERATE;
8.5 Implementation of the LMS Algorithm

y <= p(0) + p(1); -- Compute ADF output

ysxt: PROCESS (y) -- Scale y by 128 because x is fraction
BEGIN
    sxty(8 DOWNTO 0) <= y(15 DOWNTO 7);
    FOR k IN 15 DOWNTO 9 LOOP
        sxty(k) <= y(y'high);
    END LOOP;
END PROCESS;
e <= sxtd - sxty;
emu <= e(8 DOWNTO 1); -- e*mu divide by 2 and
                        -- 2 from xemu makes mu=1/4
MulGen2: FOR I IN 0 TO L-1 GENERATE
    FUPDATE: lpm_mult -- Multiply xemu(i) = emu * x(i);
    GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W1,
                  LPM_REPRESENTATION => "SIGNED",
                  LPM_WIDTHP => W2,
                  LPM_WIDTHS => W2)
    PORT MAP ( dataa => x(I), datab => emu,
               result => xemu(I));
END GENERATE;
y_out <= sxty; -- Monitor some test signals
e_out <= e;
f0_out <= f(0);
f1_out <= f(1);
END fpga;

The design is a literal interpretation of the adaptive LMS filter architecture found in Fig. 8.20 (p. 506). The output of each tap of the tapped delay line is multiplied by the appropriate filter coefficient and the results are added. The response of the adaptive filter $y$ and of the overall system $e$ to a reference signal $x$ and a desired signal $d$ is shown in Fig. 8.21. The filter adapts after approximately 20 steps at 1 $\mu$s to the optimal values $f_0 = 43.3$ and $f_1 = 25$. The design uses 50 LEs, 4 embedded multipliers, and has a 74.59 MHz Registered Performance.

The previous example also shows that the standard LMS implementation has a low Registered Performance due to the fact that two multipliers and several add operations have to be performed in one clock cycle before the filter coefficient can be updated. In the following section we wish therefore to study how to achieve a higher throughput.

8.5.3 Pipelined LMS Filters

As can be seen from Fig. 8.20 (p. 506) the original LMS adaptive filter has a long update path and hence the performance already for 8-bit data and coefficients is relatively slow. It is therefore no surprise that many attempts
Fig. 8.21. VHDL simulation of the power-line interference cancellation using the LMS algorithm.

have been made to improve the throughput of the LMS adaptive filter. The optimal number of pipeline stages from Fig. 8.20 (p. 506) can be computed as follows: For the \( (b \times b) \) multiplier \( f_k \) a total of \( \log_2(b) \) stages are needed, see also (2.30) p. 85. For the adder tree an additional \( \log_2(L) \) pipeline stages would be sufficient and one additional stage for the computation of the error. The coefficient update multiplication requires an additional \( \log_2(b) \) pipeline stages. The total number of pipeline stages for a maximum throughput are therefore

\[
D_{\text{opt}} = 2 \log_2(b) + \log_2(L) + 1,
\]

where we have assumed that \( \mu \) is a power-of-two constant and the scaling with \( \mu \) can be done without the need of additional pipeline stages. If, however, the normalized LMS is used, then \( \mu \) will no longer be a constant and depending on the bit width of \( \mu \) additional pipeline stages will be required.

Pipelining an LMS filter is not as simple as for an FIR filter, because the LMS has, as the IIR filter, feedback. We need therefore to ensure that the coefficient of the pipelined filter still converges to the same coefficient as the adaptive filter without pipelining. Most of the ideas to pipeline IIR filters can be used to pipeline an LMS adaptive filter. The suggestion include

- Delayed LMS [263, 273, 274]
- Look-ahead transformation of the pipelined LMS [258, 275, 276]
- Transposed form LMS filter [277]
- Block transformation using FFTs [268]

We have already discussed the block transform algorithms and now wish in the following to briefly review the other techniques to improve the LMS throughput.

The Delayed LMS Algorithm. In the delayed LMS algorithm (DLMS) the assumption is that the gradient of the error \( \nabla[n] = e[n]x[n] \) does not change much if we delay the coefficient update by a couple of samples, i.e., \( \nabla[n] \approx \nabla[n - D] \). It has been shown [273, 274] that as long as the delay is less than the system order, i.e., filter length, this assumption is well true and the update does not degrade the convergence speed. Long’s original DLMS
algorithm only considered pipelining the adder tree of the adaptive filter assuming also that multiplication and coefficient update can be done in one clock cycle (like for programmable digital signal processors [263]), but for a FPGA implementation multiplier and the coefficient update requires additional pipeline stages. If we introduce a delay of $D_1$ in the filter computation path and $D_2$ in the coefficient update path the LMS Algorithm 8.2 (p. 488) becomes:

$$e[n - D_1] = d[n - D_1] - f^T[n - D_1]x[n - D_1]$$
$$f[n + 1] = f[n - D_1 - D_2] + \mu e[n - D_1 - D_2]x[n - D_1 - D_2].$$

**The Look-ahead DLMS Algorithm.** For long adaptive filters with $D = D_1 + D_2 < L$ the delayed coefficient update presented in the previous section, in general, does not change the convergence of the ADF much. It can, however, for shorter filters become necessary to reduce or even remove the change in system function completely. From the IIR pipelining method we have discussed in Chap. 4, the time domain interleaving method can always be applied. We perform just a look-ahead in coefficient computation, without alternating the overall system. Let us start with the DLMS update equations with pipelining only in the coefficient computation, i.e.,

$$e^{DLMS}[n - D] = d[n - D] - x^T[n - D]f[n - D]$$
$$f[n + 1] = f[n] + \mu e[n - D]x[n - D].$$

But the error function of the LMS would be

$$e^{LMS}[n - D] = d[n - D] - x^T[n]f[n - D].$$

We follow the idea from Poltmann [275] and wish to compute the correction term $A[n]$, which cancels the change of the DLMS error computation compared with the LMS, i.e.,

$$A[n] = e^{LMS}[n - D] - e^{DLMS}[n - D].$$

The error function of the DLMS is now changed to


We need therefore to determine the term

$$A[n] = x^T[n - D](f[n] - f[n - D]).$$

The term in brackets can be recursively determined via

$$f[n] - f[n - D]$$
$$= f[n - 1] + \mu e[n - D - 1]x[n - D_1] - f[n - D]$$
$$= f[n - 2] + \mu e[n - D - 2]x[n - D - 2]$$
$$+ \mu e[n - D - 1]x[n - D - 1] - f[n - D]$$
\[ = \sum_{s=1}^{D} \mu e[n - D - s]x[n - D - s], \]

and it follows for the correction term \( A[n] \) finally

\[ A[n] = x^T(n - D) \left( \sum_{s=1}^{D} \mu e[n - D - s]x[n - D - s] \right) \]

\[ e^{\text{DLMS}}[n - D] = d[n - D] - x^T(n - D)f[n - D] - x^T(n - D) \left( \sum_{s=1}^{D} \mu e[n - D - s]x[n - D - s] \right). \]

It can be seen that this correction term needs an additional \( 2D \) multiplication, which may be too expensive in some applications. It has been suggested [276] to “relax” the requirement for the correction term but some additional multipliers are still necessary.

We can, however, remove the influence of the coefficient update delay, by applying the look-ahead principle [258], i.e.,

\[ f[n + 1] = f[n - D_1] + \mu \sum_{k=0}^{D_2-1} e[n - D_1 - k]x[n - D_1 - k]. \] (8.47)

The summation in (8.47) builds the moving average over the last \( D_2 \) gradient values, and makes it intuitively clear that the convergence will proceed more smoothly. The advantage compared with the transformation from Poltmann is that this look-ahead computation can be done without a general multiplication. The moving average in (8.47) may even be implemented with a first-order CIC filter (see Fig. 5.15, p. 260), which reduced the arithmetic effort to one adder and a subtractor.

Similar approaches to the idea from Poltmann to improve the DLMS algorithm have also been suggested [278, 279, 280].

### 8.5.4 Transposed Form LMS Filter

We have seen that the DLMS algorithm can be smoothed by introducing a look-ahead computation in the coefficient update, as we have used in IIR filters, but is, in general, not without additional cost. If we use, however, the transposed FIR structure (see Fig. 3.3, p. 167) instead of the direct structure, we can eliminate the delay by the adder tree completely. This will reduce the requirement for the optimal number of pipeline stages from (8.46), p. 508, by \( \log_2(L) \) stages. For a LTI system both direct and transposed filters are described by the same convolution equation, but for a time-varying coefficient we need to change the filter coefficient from

\[ f_k[n] \quad \text{to} \quad f_k[n - k]. \] (8.48)
The equation for the estimated gradient (8.14) on page 488 now becomes

\[ \hat{\nabla}[n] = -2e[n] \frac{\partial e[n-k]}{\partial f_k[n]} \] (8.49)

\[ = -2e[n]x[n-k] \frac{f_k[n-k]}{\partial f_k[n]} \] (8.50)

If we now assume that the coefficient update is relatively slow, i.e., \( f_k[n-k] \approx f_k[n] \) the gradient becomes,

\[ \hat{\nabla}[n] \approx -2e[n]x[n], \] (8.51)

and the coefficient update equation becomes:

\[ f_k[n-k+1] = f_k[n-k] + \mu e[n]x[n]. \] (8.52)

The learning characteristics of the transposed-form adaptive filter algorithms have been investigated by Jones [277], who showed that we will get a somewhat slower convergence rate when compared with the original LMS algorithm. The stability bound regarding \( \mu \) also needs to be determined and is found to be smaller than for the LMS algorithm.

8.5.5 Design of DLMS Algorithms

If we wish to pipeline the LMS filter from Example 8.5 (p. 504) we conclude from the discussion above (8.46) that the optimal number of pipeline stages becomes:

\[ D_{opt} = 2 \log_2(b) + \log_2(L) + 1 = 2 \times 3 + 1 + 1 = 8. \] (8.53)

On the other hand, pipelining the multiplier can be done without additional costs and we may therefore consider only using 6 pipeline stages. Figure 8.22 shows a MATLAB simulation in 8-bit precision with a delay 6. Compared with the original LMS design from Example 8.5 (p. 504) it shows some “overswing” in the adaptation process.

Example 8.6: Two-tap Pipelined Adaptive LMS FIR Filter

The VHDL design\(^4\) for a filter with two coefficients \( f_0 \) and \( f_1 \) with a step size of \( \mu = 1/4 \) is shown in the following listing.

\[ -- This is a generic DLMS FIR filter generator \]
\[ -- It uses W1 bit data/coefficients bits \]
\[ LIBRARY lpm; -- Using predefined packages \]
\[ USE lpm.lpm_components.ALL; \]
\[ LIBRARY ieee; \]
\[ USE ieee.std_logic_1164.ALL; \]
\[ USE ieee.std_logic_arith.ALL; \]
\[ USE ieee.std_logic_signed.ALL; \]

\[^4\] The equivalent Verilog code \texttt{fir_lms.v} for this example can be found in Appendix A on page 721. Synthesis results are shown in Appendix B on page 731.
Fig. 8.22. 8-bit MATLAB simulation of the power-line interference cancellation using the DLMS algorithm with a delay of 6.

ENTITY fir6dlms IS -------- Interface
  GENERIC (W1 : INTEGER := 8; -- Input bit width
    W2 : INTEGER := 16; -- Multiplier bit width 2*W1
    L : INTEGER := 2; -- Filter length
    Delay : INTEGER := 3 -- Pipeline Delay
  );
  PORT ( clk : IN STD_LOGIC;
    x_in : IN STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
    d_in : IN STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
    e_out, y_out : OUT STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
    f0_out, f1_out : OUT STD_LOGIC_VECTOR(W1-1 DOWNTO 0));
END fir6dlms;

ARCHITECTURE fpga OF fir6dlms IS

  SUBTYPE N1BIT IS STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
  SUBTYPE N2BIT IS STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
  TYPE ARRAY_N1BITF IS ARRAY (0 TO L-1) OF N1BIT;
  TYPE ARRAY_N1BITX IS ARRAY (0 TO Delay+L-1) OF N1BIT;
  TYPE ARRAY_N1BITD IS ARRAY (0 TO Delay) OF N1BIT;
  TYPE ARRAY_N1BIT IS ARRAY (0 TO L-1) OF N1BIT;
  TYPE ARRAY_N2BIT IS ARRAY (0 TO L-1) OF N2BIT;

  SIGNAL xemu0, xemu1 : N1BIT;
  SIGNAL emo : N1BIT;
  SIGNAL y, sxty : N2BIT;

  SIGNAL e, sxtd : N2BIT;
  SIGNAL f : ARRAY_N1BITF; -- Coefficient array
  SIGNAL x : ARRAY_N1BITX; -- Data array
  SIGNAL d : ARRAY_N1BITD; -- Reference array
  SIGNAL p, xemu : ARRAY_N2BIT; -- Product array

BEGIN
dsxt: PROCESS (d)  -- make d a 16 bit number
BEGIN
  sxtd(7 DOWNTO 0) <= d(Delay);
  FOR k IN 15 DOWNTO 8 LOOP
    sxtd(k) <= d(3)(7);
  END LOOP;
END PROCESS;

Store: PROCESS  -------> Store these data or coefficients
BEGIN
  WAIT UNTIL clk = '1';
  d(0) <= d_in; -- Shift register for desired data
  d(1) <= d(0);
  d(2) <= d(1);
  d(3) <= d(2);
  x(0) <= x_in; -- Shift register for data
  x(1) <= x(0);
  x(2) <= x(1);
  x(3) <= x(2);
  x(4) <= x(3);
  f(0) <= f(0) + xemu(0)(15 DOWNTO 8); -- implicit
  f(1) <= f(1) + xemu(1)(15 DOWNTO 8); -- divide by 2
END PROCESS Store;

MulGen1: FOR I IN 0 TO L-1 GENERATE
FIR: lpm_mult  -- Multiply p(i) = f(i) * x(i);
  GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W1,
                 LPM_REPRESENTATION => "SIGNED",
                 LPM_PIPELINE => Delay,
                 LPM_WIDTHP => W2,
                 LPM_WIDTHS => W2)
  PORT MAP ( dataa => x(I), datab => f(I),
             result => p(I), clock => clk);
END GENERATE;

y <= p(0) + p(1); -- Computer ADF output

ysxt: PROCESS (y)  -- scale y by 128 because x is fraction
BEGIN
  sxty(8 DOWNTO 0) <= y(15 DOWNTO 7);
  FOR k IN 15 DOWNTO 9 LOOP
    sxty(k) <= y(y'high);
  END LOOP;
END PROCESS;

  e <= sxtd - sxty; -- e*mu divide by 2 and 2
emu <= e(8 DOWNTO 1); -- from xemu makes mu=1/4

MulGen2: FOR I IN 0 TO L-1 GENERATE
FUPDATE: lpm_mult  -- Multiply xemu(i) = emu * x(i);
  GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W1,
                 LPM_REPRESENTATION => "SIGNED",
Table 8.2. Size and performance data of different pipeline options of the DLMS algorithms.

<table>
<thead>
<tr>
<th>$D$</th>
<th>LEs</th>
<th>$9 \times 9$-bit multipliers</th>
<th>MHz</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50</td>
<td>4</td>
<td>74.59</td>
<td>Original LMS</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>4</td>
<td>109.28</td>
<td>Original DLMS</td>
</tr>
<tr>
<td>3</td>
<td>74</td>
<td>4</td>
<td>123.30</td>
<td>Pipeline of $f$ update only</td>
</tr>
<tr>
<td>6</td>
<td>138</td>
<td>4</td>
<td>176.15</td>
<td>Pipeline all multipliers</td>
</tr>
<tr>
<td>8</td>
<td>179</td>
<td>4</td>
<td>368.19</td>
<td>Optimal number of stages</td>
</tr>
</tbody>
</table>

Fig. 8.23. VHDL simulation of the power-line interference cancelation using the DLMS algorithm with a delay of 6.

```
LPM_PIPELINE => Delay,
LPM_WIDTHP => W2,
LPM_WIDTHS => W2)
PORT MAP ( dataa => x(I+Delay), datab => emu,
         result => xemu(I), clock => clk);
END GENERATE;

y_out <= sxty;  -- Monitor some test signals
e_out  <= e;
f0_out <= f(0);
f1_out <= f(1);

END fpga;
```

The design is a literal interpretation of the adaptive LMS filter architecture found in Fig. 8.20 (p. 489) with the additional delay of three pipeline stages for each multiplier. The output of each tap of the tapped delay line is multiplied by the appropriate filter coefficient and the results are added. Note the additional delays for $x$ and $d$ in the Store: PROCESS to make the signals coherent. The response of the adaptive filter $y$ and of the overall system $e$ to a reference signal $x$ and a desired signal $d$ is shown in the VHDL simulation in Fig. 8.23. The filter adapts after approximately 30 steps at 1.5 $\mu$s to the optimal values $f_0 = 43.3$ and $f_1 = 25$, but it also shows some overswing in the adaptation process. The design uses 138 LEs, 4 embedded multipliers, and has a 176.15 MHz Registered Performance.
Compared with the previous example we may also consider other pipelining options. We may, for instance, use pipelining only in the coefficient update, or we may implement the optimal number of pipeline stages, i.e., 8. Table 8.2 gives an overview of the different options.

From Table 8.2 it can be seen that compared to the original LMS algorithm we may gain up to a factor of 4 speed improvement, while at the same time the additional hardware cost are only about 10%. The additional effort comes from the extra delays of the reference data \(d[n]\) and the filter input \(x[n]\). The limitation is just that it may become necessary for large pipeline delays to adjust \(\mu\) in order to guarantee stability.

8.5.6 LMS Designs using SIGNUM Function

We saw in the previous section that the implementation cost of the LMS algorithm is already high for short filter length. The highest cost of the filter comes from the large number of general multipliers and the major goal in reducing the effort is to reduce the number of multipliers. Obviously the FIR filter part can not be reduced, but different simplifications in the computation of the coefficient update have been investigated. Given the fact that to ensure stability usually the step size is chosen much smaller than \(\mu_{\text{max}}\), the following suggestions have been made:

- Use only the sign of the reference data \(x[n]\) not the full precision value to update the filter coefficients.
- Use only the sign of the error \(e[n]\) not the full precision value to update the filter coefficients.
- Use both of the previous simplifications via the sign of error and data.

The three modifications can be described with the following coefficient update equations in the LMS algorithm:

\[
\begin{align*}
    f[n+1] &= f[n] + \mu \times e[n] \times \text{sign}(x[n]) & \text{sign data function} \\
    f[n+1] &= f[n] + \mu \times x[n] \times \text{sign}(e[n]) & \text{sign error function} \\
    f[n+1] &= f[n] + \mu \times \text{sign}(e[n]) \times \text{sign}(x[n]) & \text{sign-sign function}.
\end{align*}
\]

We note from the simulation of the three possible simplifications shown in Fig. 8.24 that for the sign-data function almost the same result occurs as in the full precision case. This is no surprise, because our input reference signal \(x[n] = \cos(\pi n/2 + \phi)\) will not be much quantized through the sign operation anyway. This is much different for the sign-error function. Here the quantization through the sign operation essentially alters the time constant of the system. But finally, after about 2.5 s the correct values are reached, although from the system output \(e[n]\) we note the essential ripple in the output function even after a long simulation time. Finally, the sign-sign algorithm converges faster than the sign-error algorithm, but here also the
system output shows essential ripple for $e[n]$. From the simulation it can be seen that the sign-function simplification (to save the $L$ multiplications in the filter coefficient update) has to be evaluated carefully for the specific application to still guarantee a stable system and acceptable time constants of the system. In fact, it has been shown that for specific signals and application the sign algorithms does not converge, although the full precision algorithm would converge. Besides the sign effect we also need to ensure that the integer quantization through the implementation does not alter the desired system properties.

Another point to consider when using the sign function is the error floor that can be reached. This is discussed in the following example.

**Example 8.7: Error Floor in Signum LMS Filters**

Suppose we have a system identification configuration as discussed in Sect. 8.3.1 (p. 493), and we wish to use one of the signum-type ADF algorithms. What will then be the error floor that can be reached? Obviously through the signum operation we will lose some precision and we expect that we will not
reach the same low-noise level as with a full-precision LMS algorithm. We also expect that the learning rate will be somewhat decreased when compared with the full-precision LMS algorithm. This can be verified by the simulation results shown in Fig. 8.25 for an average over 50 learning curves and two different eigenvalue ratios (EVRs). The sign data algorithms shows some delay in the adaptation when compared with the full-precision LMS algorithm, but reaches the error floor, which was set to $-60$ dB. Signed error and sign-sign algorithms show larger delays in the adaptation and also reach only an error floor of about $-40$ dB. This larger error may or may not be acceptable for some applications.

The sign-sign algorithm is attractive from a software or hardware implementation standpoint and has been used for the International Telecommunication Union (ITU) standard for adaptive differential pulse code modulation (ADPCM) transmission. From a hardware implementation standpoint we actually do not need to implement the sign-sign algorithm, because the multiplication with $\mu$ is just a scaling with a constant and one of the single sign
8.6 Recursive Least Square Algorithms

In the LMS algorithm we have discussed in the previous sections the filter coefficients are gradually adjusted by a stochastic gradient method to finally approximate the Wiener–Hopf optimal solution. The recursive least square (RLS) algorithm takes another approach. Here, the estimation of the \((L \times L)\) autocorrelation matrix \(R_{xx}\) and the cross-correlation vector \(r_{dx}\) are iteratively updated with each new incoming data pair \((x[n], d[n])\). The simplest approach would be to reconstruct the Wiener–Hopf equation (8.9), i.e., \(R_{xx}f_{\text{opt}} = r_{dx}\) and resolve it. However, this would be the equivalent of one matrix inversion as each new data point pair arrives and has the potential of being computationally expensive. The main goal of the different RLS algorithms we will discuss in the following is therefore to seek a (iterative) time recursion for the filter coefficients \(f[n+1]\) in terms of the previous least square estimate \(f[n]\) and the new data pair \((x[n], d[n])\). Each incoming new value \(x[n]\) is placed in the length-\(L\) data array \(x[n] = [x[n], x[n-1], \ldots, x[n-(L-1)]]^T\). We then wish to add \(x[n]x[n]\) to \(R_{xx}[0,0], x[n]x[n-1]\) to \(R_{xx}[0,1]\), etc. Mathematically we just compute the product \(xx^T\) and add this \((L \times L)\) matrix to the previous estimation of the autocorrelation matrix \(R_{xx}[n]\). The recursive computation may be computed as follows:

\[
R_{xx}[n+1] = R_{xx}[n] + x[n]x[n]^T = \sum_{s=0}^{n} x[s]x[s]^T.
\] (8.54)

For the cross-correlation vector \(r_{dx}[n+1]\) we also build an “improved” estimate by adding with each new pair \((x[n], d[n])\) the vector \(d[n]x[n]\) to the previous estimation of \(r_{dx}[n]\). The recursion for the cross-correlation becomes

\[
r_{dx}[n+1] = r_{dx}[n] + d[n]x[n],
\] (8.55)

we can now use the Wiener–Hopf equation in a time recursive fashion and compute

\[
R_{xx}[n+1]f_{\text{opt}}[n+1] = r_{dx}[n+1].
\] (8.56)

For the true estimates of cross- and autocorrelation matrices we would need to scale by the number of summations, which is proportional to \(n\), but the cross- and autocorrelation matrices are scaled by the same factor, which cancel each other out in the iterative algorithm and we get for the filter coefficient update

\[
f_{\text{opt}}[n+1] = R_{xx}^{-1}[n+1]r_{dx}[n+1].
\] (8.57)

Although this first version of the RLS algorithms is computationally intensive (approximately \(L^3\) operations are needed for the matrix inversion) it
still shows the principal idea of the RLS algorithm and can be quickly programmed, for instance in MATLAB, as the following code segment shows the inner loop for length-$L$ RLS filter algorithm:

```matlab
x = [xin;x(1:L-1)];  % get new sample
y = f' * x;          % filter output
err = din - y;       % error: reference - filter output
Rxx = Rxx + x*x';    % update the autocorrelation matrix
rdx = rdx + din .* x; % update the cross-correlation vector
f = Rxx^(-1) * rdx;  % compute filter coefficients
```

where $\mathbf{R}_{xx}$ is a $(L \times L)$ matrix and $\mathbf{rdx}$ is a $(L \times 1)$ vector. The cross-correlation vector is usually initialized with $\mathbf{rdx}[0] = \mathbf{0}$. The only problem with the algorithm so far arises at the first $n < L$ iterations, when $\mathbf{R}_{xx}[n]$ only has a few nonzero entries, and consequently will be singular and no inverse exists. There are a couple of ways to tackle this problem:

- We can wait with the computation of the inverse until we find that the autocorrelation matrix is nonsingular, i.e., det $(\mathbf{R}_{xx}[n]) > 0$.
- We can use $\mathbf{R}^+_{xx}[n] = (\mathbf{R}_{xx}[n]^T \mathbf{R}_{xx}[n])^{-1} \mathbf{R}_{xx}[n]^T$ the so-called pseudoinverse, which is a standard result in linear algebra regarding the solution of an overdetermined set of linear equations.
- We can initialize the autocorrelation matrix $\mathbf{R}_{xx}$ with $\delta \mathbf{I}$ where $\delta$ is chosen to be a small (large) constant for high (low) S/N ratio of the input signal.

The third approach is the most popular due to the computational benefit and the possibility to set an initial “learning rate” using the constant $\delta$. The influence of the initialization in the RLS algorithm for an experiment similar to Sect. 8.3.1 (p. 493) with an average over 5 learning curves is shown in Fig. 8.26. The upper row shows the full-length simulation over 4000 iterations, while the lower row shows the first 100 iterations only. For high S/N ($-48$ dB) we may use a large value for the initialization, which yields a fast convergence. For low S/N values ($-10$ dB) small initialization values should be used, otherwise large errors at the first iterations can occur, which may or may not be tolerable for the specific application.

A more computationally attractive approach than the first “brute force” RLS algorithm will be discussed in the following. The key idea is that we do not compute the matrix inversion at all and use a time recursion directly for $\mathbf{R}^{-1}_{xx}[n]$, we actually will never have (or need) $\mathbf{R}_{xx}[n]$ available. To do so, we substitute the Wiener equation for time $n+1$, i.e., $\mathbf{f}[n+1] \mathbf{R}_{xx}[n+1] = \mathbf{rdx}[n+1]$ into (8.55) it follows that

$$
\mathbf{R}_{xx}[n+1] \mathbf{f}[n+1] = \mathbf{R}_{xx}[n] \mathbf{f}[n] + d[n+1] \mathbf{x}[n+1].
$$

(8.58)

Now we use (8.54) to get

$$
\mathbf{R}_{xx}[n+1] \mathbf{f}[n+1] = (\mathbf{R}_{xx}[n+1] - \mathbf{x}[n+1] \mathbf{x}^T[n+1]) \mathbf{f}[n] + d[n+1] \mathbf{x}[n+1].
$$

(8.59)
Fig. 8.26. Learning curves of the RLS algorithms using different initialization of $R_{xx}^{-1}[0] = \delta I$ or $R_{xx}[0] = \delta^{-1} I$. High S/N is -48 dB and low is -10 dB. $\delta = 1000$, 1 or 1/1000.

We can rearrange (8.59) by multiplying by $R_{xx}^{-1}[n + 1]$ to have $f[n + 1]$ on the lefthand side of the equation:

$$f[n + 1] = f[n] + R_{xx}^{-1}[n + 1]x[n + 1] \left( d[n + 1] - f^T[n]x[n + 1] \right)$$

$$= f[n] + k[n + 1]e[n + 1],$$

where the *a priori error* is defined as

$$e[n + 1] = d[n + 1] - f^T[n]x[n + 1],$$

and the *Kalman gain vector* is defined as

$$k[n + 1] = R_{xx}^{-1}[n + 1]x[n + 1].$$

(8.60)

As mentioned above the direct computation of the matrix inversion is computationally intensive, and it is much more efficient to use again the iteration equation (8.54) to actually avoid the inversion at all. We use the so-called “matrix inversion lemma,” which can be written as the following matrix identity
\[(A + BCD)^{-1} = A^{-1} - A^{-1}B(A^{-1}BDA^{-1})(C + DA^{-1}B)^{-1},\]

which holds for all matrices \(A, B, C,\) and \(D,\) of compatible dimensions and nonsingular \(A.\) We make the following associations:

\[
\begin{align*}
A &= R_{xx}[n + 1] & B &= x[n] \\
C &= 1 & D &= x^T[n].
\end{align*}
\]

The iterative equation for \(R_{xx}^{-1}\) becomes:

\[
R_{xx}^{-1}[n + 1] = \left( R_{xx}^{-1}[n] + x[n]x^T[n] \right)^{-1} = R_{xx}^{-1}[n] + \frac{R_{xx}^{-1}[n]x[n]x^T[n]R_{xx}^{-1}[n]}{1 + x^T[n]R_{xx}^{-1}[n]x[n]}.
\] (8.61)

If we use the Kalman gain factor \(k[n]\) from (8.60) we can rewrite (8.61) more compactly as:

\[
R_{xx}^{-1}[n + 1] = \left( R_{xx}^{-1}[n] + x[n + 1]x^T[n + 1] \right)^{-1} = R_{xx}^{-1}[n] + \frac{k[n]k^T[n]}{1 + x^T[n]k[n]}.
\]

This recursion is as mentioned before initialized [252] with

\[
R_{xx}^{-1}[0] = \delta I \quad \text{with} \quad \delta = \begin{cases} 
\text{large positive constant for high SNR} \\
\text{small positive constant for low SNR.}
\end{cases}
\]

With this recursive computation of the inverse autocorrelation matrix the computation effort is now proportional to \(L^2,\) an essential saving for large values of \(L.\) Figure 8.27 shows a summary of the RLS adaptive filter algorithm.

### 8.6.1 RLS with Finite Memory

As we can see from (8.54) and (8.55) the adaptive algorithm derived so far has an infinite memory. The values of the filter coefficients are functions of all past inputs starting with time zero. As will be discussed next it is often useful to introduce a “forgetting factor” into the algorithm, so that recent data are given greater importance than older data. This not only reduces the influence of older data, it also accomplishes that through the update of the cross- and autocorrelation with each new incoming data pair no overflow in the arithmetic will occur. One way of accomplishing a finite memory is to replace the sum-of-squares cost function, by an exponentially weighted sum of the output:

\[
J = \sum_{s=0}^{n} \rho^{n-s} e^2[s],
\] (8.62)
where $0 \leq \rho \leq 1$ is a constant determining the effective memory of the algorithm. The case $\rho = 1$, is the infinite-memory case, as before. When $\rho < 1$ the algorithm will have an effective memory of $\tau = -1/\log(\rho) \approx 1/(1 - \rho)$ data points. The exponentially weighted RLS algorithm can now be summarized as:

**Algorithm 8.8: RLS Algorithm**

The exponentially weighted RLS algorithm to adjust the $L$ coefficients of an adaptive filter uses the following steps:

1) Initialize $x = f = [0, 0, \ldots, 0]^T$ and $R_{xx}^{-1}[0] = \delta I$.

2) Accept a new pair of input samples $\{x[n+1], d[n+1]\}$ and shift $x[n+1]$ input the reference signal vector $x[n+1]$.

3) Compute the output signal of the FIR filter, via

$$y[n+1] = f^T[n]x[n+1].$$  \hspace{1cm} (8.63)

4) Compute the a priori error function with

$$e[n+1] = d[n+1] - y[n+1].$$  \hspace{1cm} (8.64)

5) Compute the Kalman gain factor with

$$k[n+1] = R_{xx}^{-1}[n+1]x[n+1].$$  \hspace{1cm} (8.65)

6) Update the filter coefficient according to

$$f[n+1] = f[n] + k[n+1]e[n+1].$$  \hspace{1cm} (8.66)

7) Update the filter inverse autocorrelation matrix according to

$$R_{xx}^{-1}[n+1] = \frac{1}{\rho} \left( R_{xx}^{-1}[n] + \frac{k[n+1]k^T[n+1]}{\rho + x^T[n+1]k[n+1]} \right).$$  \hspace{1cm} (8.67)

Next continue with step 2.
The computational cost of the RLS are $(3L^2 + 9L)/2$ multiplications and $(3L^2 + 5L)/2$ additions or subtractions, per input sample, which is still more essential than the LMS algorithm. The advantage as we will see in the following example will be a higher rate of convergence and no need to select the step size $\mu$, which may at times be difficult when stability of the adaptive algorithm has to be guaranteed.

**Example 8.9: RLS Learning Curves**

In this example we wish to evaluate a configuration called system identification to compare RLS and LMS convergence. We have used this type of performance evaluation already for LMS ADF in Sect. 8.3.1 (p. 493). The system configuration is shown in Fig. 8.12 (p. 494). The adaptive filter has a length of $L = 16$, the same length as the “unknown” system, whose coefficients have to be learned. The additive noise level behind the “unknown system” has been set to $-48$ dB equivalent for an 8-bit quantization. For the LMS algorithm the eigenvalue ratio (EVR) is the critical parameter that determines the convergence speed, see (8.25), p. 491. In order to generate a different eigenvalue ratio we use a white Gaussian noise source with $\sigma^2 = 1$ that is filtered by a FIR type filter shown in Table 8.1 (p. 495). The coefficients are normalized to $\sum_k h[k]^2 = 1$, so that the signal power does not change. The impulse response of the unknown system is an odd filter with coefficients $1, -2, 3, -4, \ldots, -3, 2, -1$ as shown in Fig. 8.28a. The step size for the LMS algorithm has been determined with

$$\mu_{\text{max}} = \frac{2}{3 \times L \times E\{x^2\}} = \frac{1}{24}.$$  

(8.68)

In order to guarantee perfect stability the step size for the LMS algorithm has been chosen to be $\mu = \mu_{\text{max}}/2 = 1/48$. For the transform-domain DCT-LMS algorithm a power normalization for each coefficient is used, see Fig. 8.17 (p. 502). From the simulation results shown in Fig. 8.29 it can be seen that the RLS converges faster than the LMS with increased EVR. DCT-LMS converges faster than LMS and in some cases quite as fast as the RLS algo-
Fig. 8.29. Simulation results for a \( L = 16 \)-tap adaptive filter system identification. Learning curve \( J \) for LMS, transform-domain DCT-LMS, and RLS with \( R_{xx}[0] = I \). (a) EVR = 1. (b) EVR = 10. (c) EVR = 100. (d) EVR = 1000.

8.6.2 Fast RLS Kalman Implementation

For the least-square FIR fast Kalman algorithm first presented by Ljung et al. [281] the concept of single-step linear forward and backward prediction play a central role. Using these forward and backward coefficients in an all-recursive, one-dimensional Levison–Durbin type algorithm it will be possible to update the Kalman gain vector with only an \( O(L) \) type effort.

A one-step forward predictor is presented in Fig. 8.30. The predictor estimates the present value \( x[n] \) based on its \( L \) most recent past values. The \textit{a posteriori} error in the prediction is quantified by
The superscript indicates that it is the forward prediction error, while the subscript describes the order (i.e., length) of the predictor. We will drop the index \( L \) and the vector length should be \( L \) for the remainder of this section, if not otherwise noted. It is also advantageous to compute also the a priori error that is computed using the filter coefficient of the previous iteration, i.e.,

\[
e_f^L[n] = x[n] - \hat{x}[n] = x[n] - a^T[n]x_L[n - 1].
\]  

The least-square minimum of \( e_f^L[n] \) can be computed via

\[
\frac{\partial(e_f^T[n])^2}{\partial a^T[n]} = -E\{(x[s] - a^T[s]x[n])x[n - s]\} = 0
\]  

for \( s = 1, 2, \ldots, L \).

This leads again to an equation with the \((L \times L)\) autocorrelation matrix, but the right-hand side is different from the Wiener–Hopf equation:

\[
R_{xx}[n - 1]a[n] = r_f^T[n] = \sum_{s=0}^{n} x[s - 1]x[s].
\]  

The minimum value of the cost function is given by

\[
\alpha_f[n] = r_0^f[n] - a^T[n]r_f^T[n],
\]  

where \( r_0^f[n] = \sum_{s=0}^{n} x[s]^2 \).

The important fact about this predictor is now that the Levinson–Durbin algorithm can solve the least-square error minimum of (8.69) in a recursive fashion, without computing a matrix inverse. To update the predictor coefficient we need the same Kalman gain factor as in (8.66) for updating the filter coefficients, namely

\[
a_L[n + 1] = a_L[n] + k_L[n]e_f^L[n].
\]  

We will see later how the linear prediction coefficients can be used to iteratively update the Kalman gain factor. In order to take advantage of the fact that the data vectors from one iteration to the next only differ in the
first and last element, we use an augmented-by-one version \( k_{L+1}[n] \) of the Kalman gain update equation (8.65) which is given by

\[
k_{L+1}[n + 1] = R_{x\bar{x},L+1}[n+1]x_{L+1}[n+1].
\]

(8.74)

\[
= \begin{bmatrix} r_{0L}[n+1] & r_{L}^{T}[n+1] \\ \end{bmatrix} \begin{bmatrix} r_{L}^{-1} \\ R_{x\bar{x},L}[n] \end{bmatrix} \begin{bmatrix} x[n+1] \\ x_L[n] \end{bmatrix}.
\]

(8.75)

In order to compute the matrix inverse of \( R_{x\bar{x},L+1}[n] \) we use a well-known theorem of matrix inversion of block matrices, i.e.,

\[
M^{-1} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \frac{(AD^{-1}C - A)^{-1}}{D^{-1}C - (AD^{-1}C - A)^{-1}BD^{-1}} \frac{(AD^{-1}C - A)^{-1}BD^{-1}}{D^{-1} - (D^{-1}CBD^{-1})(AD^{-1}C - A)^{-1}},
\]

if \( D^{-1} \) is nonsingular. We now make the following associations:

\[
A = r_{0L}[n+1] \quad B = r_{L}^{T}[n+1] \\
C = r_{L}[n] \quad D = R_{x\bar{x}}[n],
\]

we then get

\[
D^{-1}C = R_{x\bar{x},L}[n]r_{L}^{T}[n] = a_{L}[n+1] \\
BD^{-1} = r_{L}^{T}[n+1]R_{x\bar{x}}^{-1}[n] = a_{L}^{T}[n+1] \\
-(AD^{-1}C - A)^{-1} = -r_{L}^{T}[n+1]R_{x\bar{x},L}[n]r_{L}^{T}[n] + r_{0L}[n+1] \\
= r_{0L}[n+1] - a_{L}^{T}[n+1]r_{L}^{T}[n] = a_{L}[n+1].
\]

We can now rewrite \( R_{x\bar{x},L+1}[n+1] \) from (8.74) as

\[
R_{x\bar{x},L+1}[n+1] = \begin{bmatrix} 1 \\ -1 + \frac{a_{L}[n+1]}{\alpha_{L}[n+1]} \end{bmatrix} \begin{bmatrix} a_{L}^{T}[n+1] \\ \alpha_{L}[n+1] \end{bmatrix} R_{x\bar{x},L}[n] + \frac{a_{L}[n+1]a_{L}^{T}[n+1]}{\alpha_{L}[n+1]}. 
\]

(8.77)

After some rearrangements (8.74) can be written as

\[
k_{L+1}[n+1] = \begin{bmatrix} 0 \\ k_{L}[n+1] \end{bmatrix} + \begin{bmatrix} \epsilon_{L}[n+1] \\ \alpha_{L}[n+1] \end{bmatrix} \begin{bmatrix} \gamma_{L} \end{bmatrix}
\]

\[
= \begin{bmatrix} g_{L}[n+1] \\ \gamma_{L}[n+1] \end{bmatrix}.
\]

Unfortunately, we do not have a closed recursion so far. For the iterative update of the Kalman gain vector, we need besides the forward prediction coefficients, also the coefficients of the one-step backward predictor, whose \textit{a posteriori} error function is

\[
\epsilon_{b}[n] = x[n-L] - \hat{x}[n-L] = x[n-L] - b^{T}[n]x[n],
\]

(8.78)
Fig. 8.31. Linear backward prediction of order \( L \).

again all vectors are of size \((L \times 1)\). The linear backward predictor is shown in Fig. 8.31.

The \textit{a priori} error for the backward predictor is given by

\[
e_b^L[n] = x[n - L] - b^T[n - 1]x_L[n],
\]

The iterative equation to compute the least-square coefficients for the backward predictor is equivalent to the forward case and given by

\[
R_{xx}[n]b[n] = r^b[n] = \sum_{s=0}^{n} x[s]x[s - L],
\]

and the minimum value for the total squared error becomes

\[
\alpha^f[n] = r_0^b[n] - b^T[n]r^b[n],
\]

where \( r_0^b[n] = \sum_{s=0}^{n} x[s - L]^2 \). To update the backward predictor coefficient we need again the Kalman gain factor in (8.66) as for the updating of the filter coefficients, namely

\[
b_L[n + 1] = b_L[n] + k_L[n + 1]e_L^b[n + 1].
\]

Now we can again find a Levinson–Durbin type of recursive equation for the extended Kalman gain vector, only this time using the backward prediction coefficients. It follows that

\[
k_L+1[n + 1] = R_{xx,L+1}^{-1}[n + 1]x_{L+1}[n + 1].
\]

To solve the matrix inversion, we define as in (8.76) a \((L + 1) \times (L + 1)\) block matrix \( M \), only this time the block \( A \) needs to be nonsingular and it follows that

\[
M^{-1} = \left[ \begin{array}{c|c} A & B \\ \hline C & D \end{array} \right]^{-1} = \frac{\left( A^{-1}BCA^{-1}\right)(CA^{-1}B - D)^{-1}A^{-1}B(CA^{-1}B - D)^{-1}}{(CA^{-1}B - D)^{-1}CA^{-1} - (CA^{-1}B - D)^{-1}}.
\]
We now make the following associations:
\[ A = R_{xx,L}[n] \quad B = r^b_L[n + 1] \]
\[ C = r^{kT}_L[n] \quad D = r^b_{0L}[n + 1]. \]
we then get the following intermediate results
\[ A^{-1}B = R^{-1}_{xx,L}[n]r^b_L[n + 1] = b_L[n + 1] \]
\[ CA^{-1} = r^{kT}_L[n + 1]R^{-1}_{xx,L}[n] = b^T_L[n + 1] \]
\[ -(CA^{-1}B - D) = -b^T_L[n + 1]r^b_L[n + 1] + r^b_{0L}[n + 1] = \alpha^b_L[n + 1]. \]
Using this intermediate results in (8.78) we get
\[ R^{-1}_{xx,L+1}[n] = \begin{bmatrix} R^{-1}_{xx,L}[n] + \frac{b_L[n+1]b^T_L[n+1]}{\alpha^f_L[n]} & \frac{b^T_L[n+1]}{\alpha^f_L[n]} \\ \frac{b_L[n+1]}{\alpha^f_L[n]} & 1 \end{bmatrix}. \]
After some rearrangements (8.80) can now, using the backward prediction coefficients, be written as
\[ k_{L+1}[n + 1] = \begin{bmatrix} k_L[n + 1] \\ 0 \end{bmatrix} + \frac{\epsilon^b_L[n + 1]}{\alpha^b_L[n + 1]} \begin{bmatrix} b_L[n + 1] \\ 1 \end{bmatrix} = \begin{bmatrix} g_L[n + 1] \\ \gamma_L[n + 1] \end{bmatrix}. \]
The only iterative update equation missing so far is for the minimum values of the total square errors, which is given by
\[ \alpha^f_L[n + 1] = \alpha^f_L[n] + \epsilon^f_L[n + 1]e^f_L[n + 1] \quad (8.82) \]
\[ \alpha^b_L[n + 1] = \alpha^b_L[n] + \epsilon^b_L[n + 1]e^b_L[n + 1]. \quad (8.83) \]
We now have all iterative equations available to define the
Algorithm 8.10: Fast Kalman RLS Algorithm

The prewindowed fast Kalman RLS algorithm to adjust the $L$ filter coefficients of an adaptive filter uses the following steps:

1) Initialize $x = a = b = f = k = [0, 0, \ldots, 0]^T$ and $\alpha_f = \alpha_b = \delta$

2) Accept a new pair of input samples $\{x[n+1], d[n+1]\}$.

3) Compute now the following equations to update $a$, $b$, and $k$ in sequential order

   \begin{align*}
   e_L^f[n+1] &= x[n+1] - a^T[n]x_L[n] \\
   a_L[n+1] &= a_L[n] + k_L[n]e_L^f[n+1] \\
   e_L^b[n+1] &= x[n+1] - a^T[n+1]x_L[n] \\
   \alpha_L^f[n+1] &= \alpha_L^f[n] + e_L^f[n+1]e_L^f[n+1] \\
   k_{L+1}[n+1] &= \begin{bmatrix} 0 \\ k_L[n+1] \end{bmatrix} = \frac{e_L^f[n+1]}{\alpha_L^f[n+1]} \begin{bmatrix} 1 \\ a_L[n+1] \end{bmatrix} \\
   e_L^b[n+1] &= x[n+1] - L - b^T[n]x_L[n+1] \\
   f_L[n+1] &= f_L[n] + k_L[n+1]e_L[n+1].
   \end{align*}

4) Shift the $x[n+1]$ in the reference signal vector $x[n+1]$ and compute the following two equations in order to update the adaptive filter coefficients:

   \begin{align*}
   f_L[n+1] &= f_L[n] + k_L[n+1]e_L[n+1].
   \end{align*}

Next continue with step 2.

Counting the computational effort we find that step 3 needs 2 divisions, $8L+2$ multiplications, and $7L+2$ add or subtract operations. The coefficient update in step 4 uses an additional $2L$ multiply and add/subtract operations, that the total computational effort is $10L+2$ multiplications, $9L+2$ add/subtract operations and 2 divisions.

8.6.3 The Fast a Posteriori Kalman RLS Algorithm

A careful inspection of Algorithm 8.10 reveals that the original fast Kalman algorithm as introduced by Ljung et al. [281] is mainly based on the a priori error equations. In the fast a posteriori error sequential technique (FAEST) introduced by Carayannis et al. [282] to a greater extent the a posteriori error is used. The algorithm explores even more the iterative nature of the different parameters in the fast Kalman algorithm, which will reduce the computational effort by an additional $2L$ multiplications. Otherwise, the original fast Kalman and the FAEST use mainly the same ideas, i.e., extended by one
length Kalman gain, and the use of the forward and backward predictions \textbf{a} and \textbf{b}. We also introduce the forgetting factor \( \rho \). The following listing shows the inner loop of the FAEST algorithm in MatLab:

\begin{verbatim}
%*********** FAEST Update of k, a, and b
ef=xin - a’*x;     % a priori forward prediction error
ediva=ef/(rho*af); % a priori forward error/minimal error
ke(1)=-ediva;      % extended Kalman gain vector update
ke(2:l+1)=k - ediva*a; % split the l+1 length vector
epsf=ef*psi;       % a posteriori forward error
a=a+epsf*k;        % update forward coefficients
k=ke(1:1) + ke(l+1).*b; % Kalman gain vector update
eb=-rho*alphab*ke(l1); % a priori backward error
alphaf=rho*alphaf+ef*epsf; % forward minimal error
alpha=alpha+ke(l+1)*eb+ediva*ef; % prediction crosspower
psi=1.0/alpha;     % psi makes it a 2 div algorithm
epsb=eb*psi;       % a posteriori backward error update
alphab=rho*alphab+eb*epsb; % minimum backward error
b=b-k*epsb;        % update backward prediction coefficients
x=[xin;x(1:l-1)]; % shift new value into filter taps
%*********** Time updating of the LS FIR filter
e=din-f’*x;       % error: reference - filter output
eps=-e*psi;       % a posteriori error of adaptive filter
f=f+w*eps;        % coefficient update
\end{verbatim}

The total effort (not counting the exponential weight with \( \rho \)) is 2 divisions, \( 7L + 8 \) multiplications and \( 7L + 4 \) additions or subtractions.

### 8.7 Comparison of LMS and RLS Parameters

Finally, Table 8.3 compares the algorithms we have introduced in this chapter. The table shows a comparison in terms of computation complexity for the basic stochastic gradient (SG) methods like signed LMS (SLMS), normalized LMS (NLMS) or block LMS (BLMS) algorithm using a FFT. Transform-domain algorithms are listed next, but the effort does not include the power normalization, i.e., \( L \) normalizations in the transform domain. From the RLS algorithms we have discussed the (fast) Kalman algorithm and the FAEST algorithm. Lattice algorithm (not discussed) in general, require a large number of division and square root computations and it has been suggested to use the logarithmic number system (see Chap. 2, p. 65) in this case [283].

The data in Table 8.3 are based on the discussion in Chap. 6 of DCT and DFT and their implementation using fast DIF or DIT algorithms. For DCT or DFT of length 8 and 16 more efficient (Winograd-type) algorithms have been developed using even fewer operations. A length-8 DCT (see Fig. 6.23, p. 389),
Table 8.3. Complexity comparison for LMS and RLS algorithms for length-$L$ adaptive filter. TDLMS without normalization. Add $L$ multiplications and $2L$ add/subtract and $L$ divide, if normalization is used in the TDLMS algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Implementation</th>
<th>Computational load</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mult</td>
<td>add/sub</td>
</tr>
<tr>
<td>SG</td>
<td>LMS</td>
<td>$2L$</td>
<td>$2L$</td>
</tr>
<tr>
<td></td>
<td>SLMS</td>
<td>$L$</td>
<td>$2L$</td>
</tr>
<tr>
<td></td>
<td>NLMS</td>
<td>$2L + 1$</td>
<td>$2L + 2$</td>
</tr>
<tr>
<td></td>
<td>BLMS (FFT)</td>
<td>$10 \log_2(L) + 8$</td>
<td>$15 \log_2(L) + 30$</td>
</tr>
<tr>
<td>SG</td>
<td>Hadamard</td>
<td>$2L$</td>
<td>$4L - 2$</td>
</tr>
<tr>
<td>TDLMS</td>
<td>Haar</td>
<td>$2L$</td>
<td>$2L + 2 \log_2(L)$</td>
</tr>
<tr>
<td></td>
<td>DCT</td>
<td>$2L + \frac{3L}{2} \log_2(L) + L$</td>
<td>$2L + \frac{3L}{2} \log_2(L)$</td>
</tr>
<tr>
<td></td>
<td>DFT</td>
<td>$2L + \frac{3L}{2} \log_2(L)$</td>
<td>$2L + \frac{3L}{2} \log_2(L)$</td>
</tr>
<tr>
<td></td>
<td>KLT</td>
<td>$2L + L^2 + L$</td>
<td>$2L + 2L$</td>
</tr>
<tr>
<td>RLS</td>
<td>direct</td>
<td>$2L^2 + 4L$</td>
<td>$2L^2 + 2L - 2$</td>
</tr>
<tr>
<td></td>
<td>fast Kalman</td>
<td>$10L + 2$</td>
<td>$9L + 2$</td>
</tr>
<tr>
<td></td>
<td>lattice</td>
<td>$8L$</td>
<td>$8L$</td>
</tr>
<tr>
<td></td>
<td>FAEST</td>
<td>$7L + 8$</td>
<td>$7L + 4$</td>
</tr>
</tbody>
</table>

For instance, uses 12 multiplications and a DCT transform-domain algorithm can then be implemented with $2 \times 8 + 12 = 28$ multiplications, which compares to the FAEST algorithms $7 \times 8 + 8 = 64$. But this calculation does not take into account that a power normalization is mandatory for all TDLMS (otherwise there is no fast convergence compared with the standard LMS algorithm [271, 272]). The effort for the division may be larger than the multiplication effort. When the power normalization factor can be determined beforehand it may be possible to implement the division with hardwired scaling operations. FAEST needs only 2 divisions, independent of the ADF length.

A comparison of the RLS and LMS adaptation speed was presented in Example 8.9 (p. 523), which shows that RLS-type algorithms adapt much faster than the LMS algorithm, but the LMS algorithm can be improved essentially with transform-domain algorithms, like the DCT-LMS. Also, error floor and consistency of the error is, in general, better for the RLS algorithm, when compared with LMS or TDLMS algorithms. But none of the RLS-type algorithms can be implemented without division operations, which will require usually a larger overall system bit width, at least a fractional number representation, or even a floating-point representation [283]. The LMS algorithm on the other hand, can be implemented with only a few bits as presented in Example 8.5 (p. 504).
Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 29. If not otherwise noted use the EP2C35F672C6 from the Cyclone II family for the Quartus II synthesis evaluations.

8.1: Suppose the following signal is given
\[ x[n] = A \cos[2\pi n/T + \phi]. \]
(a) Determine the power or variance \( \sigma^2 \).
(b) Determine the autocorrelation function \( r_{xx}[\tau] \).
(c) What is the period of \( r_{xx}[\tau] \)?

8.2: Suppose the following signal is given
\[ x[n] = A \sin[2\pi n/T + \phi] + n[n], \]
where \( n[n] \) is a white Gaussian noise with variance \( \sigma^2_n \).
(a) Determine the power or variance \( \sigma^2 \) of the signal \( x[n] \)
(b) Determine the autocorrelation function \( r_{xx}[\tau] \).
(c) What is the period of \( r_{xx}[\tau] \)?

8.3: Suppose the following two signals are given:
\[ x[n] = \cos[2\pi n/T_0] \quad y[n] = \cos[2\pi n/T_1]. \]
(a) Determine the cross-correlation function \( r_{xy}[\tau] \).
(b) What is the condition for \( T_0 \) and \( T_1 \) that \( r_{xy}[\tau] = 0 \)?

8.4: Suppose the following signal statistics have been determined:
\[ R_{xx} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \quad r_{dx} = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \quad R_{dd}[0] = 20. \]
Compute
(a) Compute \( R_{xx}^{-1} \).
(b) The optimal Wiener filter weight.
(c) The error for the optimal filter weight.
(d) The eigenvalues and the eigenvalue ratio.

8.5: Suppose the following signal statistics for a second-order system are given:
\[ R_{xx} = \begin{bmatrix} r_0 & r_1 \\ r_1 & r_0 \end{bmatrix} \quad r_{dx} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \quad R_{dd}[0] = \sigma_d^2. \]
The optimal filter with coefficient should be \( f_0 \) and \( f_1 \).
(a) Compute \( R_{xx}^{-1} \).
(b) Determine the optimal filter weight error as a function of \( f_0 \) and \( f_1 \).
(c) Determine \( f_0 \) and \( f_1 \) as a function of \( r \) and \( c \).
(d) Assume now that \( r_1 = 0 \). What are the optimal filter coefficients \( f_0 \) and \( f_1 \)?
8.6: Suppose the desired signal is given as:
\[ d[n] = \cos[2\pi n/T_0]. \]
The reference signal \( x[n] \) that is applied to the adaptive filter input is given as
\[ x[n] = \sin[2\pi n/T_0] + 0.5 \cos[2\pi n/T_1], \]
where \( T_0 = 5 \) and \( T_1 = 3 \). Compute for a second-order system:
(a) \( R_{xx}, r_{dx} \), and \( R_{dd}[0] \).
(b) The optimal Wiener filter weight.
(c) The error for the optimal filter weight.
(d) The eigenvalues and the eigenvalue ratio.
(e) Repeat (a)–(d) for a third-order system.

8.7: Suppose the desired signal is given as:
\[ d[n] = \cos[2\pi n/T_0] + n[n], \]
where \( n[n] \) is a white Gaussian noise with variance 1. The reference signal \( x[n] \) that is applied to the adaptive filter input is given as
\[ x[n] = \sin[2\pi n/T_0], \]
where \( T_0 = 5 \). Compute for a second-order system:
(a) \( R_{xx}, r_{dx} \), and \( R_{dd}[0] \).
(b) The optimal Wiener filter weight.
(c) The error for the optimal filter weight.
(d) The eigenvalues and the eigenvalue ratio.
(e) Repeat (a)–(d) for a third-order system.

8.8: Suppose the desired signal is given as:
\[ d[n] = \cos[4\pi n/T_0] \]
where \( n[n] \) is a white Gaussian noise with variance 1. The reference signal \( x[n] \), which is applied to the adaptive filter input, is given as
\[ x[n] = \sin[2\pi n/T_0] - \cos[4\pi n/T_0], \]
with \( T_0 = 5 \). Compute for a second-order system:
(a) \( R_{xx}, r_{dx} \), and \( R_{dd}[0] \).
(b) The optimal Wiener filter weight.
(c) The error for the optimal filter weight.
(d) The eigenvalues and the eigenvalue ratio.
(e) Repeat (a)–(d) for a third-order system.

8.9: Using the 4 FIR filters given in Sect. 8.3.1 (p. 493) use C or MATLAB to compute the autocorrelation function and the eigenvalue ratio using the autocorrelation for of a (filtered) sequence of 10 000 white noise samples. For the following system length (i.e., size of autocorrelation matrix):
(a) \( L = 2 \).
(b) \( L = 4 \).
(c) \( L = 8 \).
(d) \( L = 16 \).
Hint: The MATLAB functions: \texttt{randn, filter, xcorr, toeplitz, eig} are helpful.

8.10: Using an IIR filter with one pole \( 0 < \rho < 1 \) use C or MATLAB to compute the autocorrelation function and plot the eigenvalue ratio using the autocorrelation for
a (filtered) sequence of 10 000 white noise samples. For the following system length
(i.e., size of autocorrelation matrix):
(a) \( L = 2 \).
(b) \( L = 4 \).
(c) \( L = 8 \).
(d) \( L = 16 \).
(e) Compare the results from (a) to (d) with the theoretical value \( \text{EVR} = \frac{(1 + \rho)}{(1 - \rho)^2} \) of Markov-1 processes [269].
Hint: The MATLAB functions: \texttt{randn, filter, xcorr, toeplitz, eig} are helpful.

8.11: Using the FIR filter for \( \text{EVR} = 1000 \) given in Sect. 8.3.1 (p. 493) use C or MATLAB to compute the eigenvectors of the autocorrelation for \( L = 16 \). Compare the eigenvectors with the DCT basis vectors.

8.12: Using the FIR filter for \( \text{EVR} = 1000 \) given in Sect. 8.3.1 (p. 493) use C or MATLAB to compute the eigenvalue ratios of the transformed power normalized autocorrelation matrices from (8.45) on page 502 for \( L = 16 \) using the following transforms:
(a) Identity transform (i.e., no transform).
(b) DCT.
(c) Hadamard.
(d) Haar.
(e) Karhunen–Loéve.
(f) Build a ranking of the transform from (a)–(e).

8.13: Using the one pole IIR filter from Exercise 8.10 use C or MATLAB to compute for 10 values of \( \rho \) in the range 0.5 to 0.95 the eigenvalue ratios of the transformed power normalized autocorrelation matrices from (8.45) on page 502 for \( L = 16 \) using the following transforms:
(a) Identity transform (i.e., no transform).
(b) DCT.
(c) Hadamard.
(d) Haar.
(e) Karhunen–Loéve.
(f) Build a ranking of the transform from (a)–(e).

8.14: Use C or MATLAB to rebuild the power estimation shown for the nonstationary signal shown in Fig. 8.15 (p. 497). For the power estimation use
(a) Equation (8.38) page 496.
(b) Equation (8.41) page 498 with \( \beta = 0.5 \).
(c) Equation (8.41) page 498 with \( \beta = 0.9 \).

8.15: Use C or MATLAB to rebuild the simulation shown in Example 8.1 (p. 485) for the following filter length:
(a) \( L = 2 \).
(b) \( L = 3 \).
(c) \( L = 4 \).
(d) Compute the exact Wiener solution for \( L = 3 \).
(e) Compute the exact Wiener solution for \( L = 4 \).

8.16: Use C or MATLAB to rebuild the simulation shown in Example 8.3 (p. 492) for the following filter length:
(a) \( L = 2 \).
(b) \( L = 3 \).
(c) \( L = 4 \).
8.17: Use C or MATLAB to rebuild the simulation shown in Example 8.6 (p. 511) for the following pipeline configuration:
(a) DLMS with 1 pipeline stages.
(b) DLMS with 3 pipeline stages.
(c) DLMS with 6 pipeline stages.
(d) DLMS with 8 pipeline stages.

8.18: (a) Change the filter length of the adaptive filter in Example 8.5 (p. 504) to three.
(b) Make a functional compilation (with the Quartus II compiler) of the HDL code for the filter.
(c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
(d) Compare the results with the simulation in Exercise 8.15b and d.

8.19: (a) Change the filter length of the adaptive filter in Example 8.5 (p. 504) to four.
(b) Make a functional compilation (with the Quartus II compiler) of the HDL code for the filter.
(c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
(d) Compare the results with the simulation in Exercise 8.15c and e.

8.20: (a) Change the DLMS filter design from Example 8.6 (p. 511) pipeline of $e[n]$ only, i.e. DLMS with 1 pipeline stage.
(b) Make a functional compilation (with the Quartus II compiler) of the HDL code for the filter.
(c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
(d) Compare the results with the simulation in Exercise 8.17a.
(e) Determine the Registered Performance and the used resources (LEs, multipliers, and M4Ks) of your D=1 design using the device EP2C35F672C6 from the Cyclone II family.
(f) Repeat (e) for the EPF10K70RC240-4 from the Flex 10K family.

8.21: (a) Change the DLMS filter design from Example 8.6 (p. 511) pipeline of $f$ update only, i.e. DLMS with 3 pipeline stages.
(b) Make a functional compilation (with the Quartus II compiler) of the HDL code for the filter.
(c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
(d) Compare the results with the simulation in Exercise 8.17b.
(e) Determine the Registered Performance and the used resources (LEs, multipliers, and M4Ks) of your D=3 design using the device EP2C35F672C6 from the Cyclone II family.
(f) Repeat (e) for the EPF10K70RC240-4 from the Flex 10K family.

8.22: (a) Change the DLMS filter design from Example 8.6 (p. 511) pipeline with an optimal number of stages, i.e. DLMS with 8 pipeline stages, 3 for each multiplier and one stage each for $e[n]$ and $y[n]$.
(b) Make a functional compilation (with the Quartus II compiler) of the HDL code for the filter.
(c) Perform a functional simulation the filter with the inputs $d[n]$ and $x[n]$.
(d) Compare the results with the simulation in Exercise 8.17d.
(e) Determine the Registered Performance and the used resources (LEs, multipliers, and M4Ks) of your D=8 design using the device EP2C35F672C6 from the Cyclone II family.
(f) Repeat (e) for the EPF10K70RC240-4 from the Flex 10K family.