Appendix B

Using MATLAB for adaptive filtering and subband adaptive filtering

This appendix provides an introduction to using MATLAB for digital signal processing (DSP), adaptive filtering and subband adaptive filtering. New MATLAB users are encouraged to read the first section of Appendix A for a quick hands-on guide to MATLAB fundamentals.

B.1 Digital signal processing

This section introduces the basic concepts of digital signal processing and how to represent digital signals in MATLAB.

B.1.1 Discrete-time signals and systems

Signals can be divided into three broad categories: (i) continuous-time (or analog) signals, (ii) discrete-time signals and (iii) digital signals. Signals that we encounter daily are mostly analog, which are defined continuously in time and amplitude. Discrete-time signals are defined only at a set of time instances with a continuous range of amplitudes. This is different from digital signals, which have discrete values in both time and amplitude.

Discrete-time and digital signals share a common characteristic; i.e. both can be treated as sequences of numbers. However, each discrete-time signal sample needs an infinite number of bits (or word-length) to represent its continuous-amplitude value. Signals processed on digital hardware (or computers) are represented in the form of digital signals, which are obtained by quantization of discrete-time signals in amplitude using a finite number of bits. Analysis of quantization (of signal samples and system parameters) effects is known as the finite word-length analysis. For development and analysis of DSP systems

Subband Adaptive Filtering: Theory and Implementation Kong-Aik Lee, Woon-Seng Gan and Sen M. Kuo
© 2009 John Wiley & Sons, Ltd. ISBN: 978- 0-470-51694- 2

and algorithms, we use discrete-time signals for mathematical simplicity. In most cases, it is acceptable to represent digital signals with a sufficient number of bits; i.e. the quantization and arithmetic errors are negligible. For example, MATLAB uses 64 bits (by default) to represent numbers. See Section A.1.4 for other supported data types in MATLAB. Remember that, in practice, only digital signals can be represented and processed by digital systems.

A DSP system is a digital hardware (or software) that implements some computational operations to process digital signals. Given one or more digital signals as inputs, a DSP system transforms, manipulates, extracts information or modifies those signals and produces outputs for some predefined purposes. Figure 1.1 depicts a block diagram of an adaptive digital system. As described in Chapter 1, and briefly discussed in this appendix, the output signal is generated by filtering the input signal such that the output is an approximation to the reference (desired) signal in a statistical sense.

B.1.2 Signal representations in MATLAB

In DSP systems, signals are sequences of numbers (or samples) created, for example, by sampling or measurement of some physical events. In MATLAB, we represent signals as finite-duration sequences in vectors due to finite memory limitations (i.e. it is not possible to save, store or load an infinite number of samples into memory on a computer). Assuming that all signals are causal (start at the time index $n = 0$), we can represent a signal sequence as a row vector in MATLAB, as in the following example:

```
un = [6,-2,-21,1,16,10,-16,-1,-7,-10,-12,3,-4,1,-4];
```

We may also define a time-index vector `n`, which has the same number of samples as `un`, to indicate the position (in discrete time) of each sample in the vector `un` as follows:

```
n = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14];
```

Note that this time-index vector can be easily generated by the MATLAB command as follows:

```
n = [0:length(un)-1];
```

The time-index vector is crucial, for example, when plotting the signal `un` (see Table A.3 for details).

A discrete-time signal is defined only for integer values of the time index n and is left undefined for other time values in between those integers. Therefore, a more precise way of plotting a discrete-time signal is shown in Figure B.1, generated by the MATLAB function `stem` (see Section A.1.8) as `stem(n,un)`. We also superimpose a dotted-continuous line generated by the `plot` function, which connects samples of the digital signal. In some cases, it would be difficult to see the shape of the signal by just plotting sample points using `stem`. We can also use the style options supported by the `plot` function to indicate those discrete sample values. For example, we can use the following statement to generate a similar plot as shown in Figure B.1:

```
plot(n,un,':o');
```

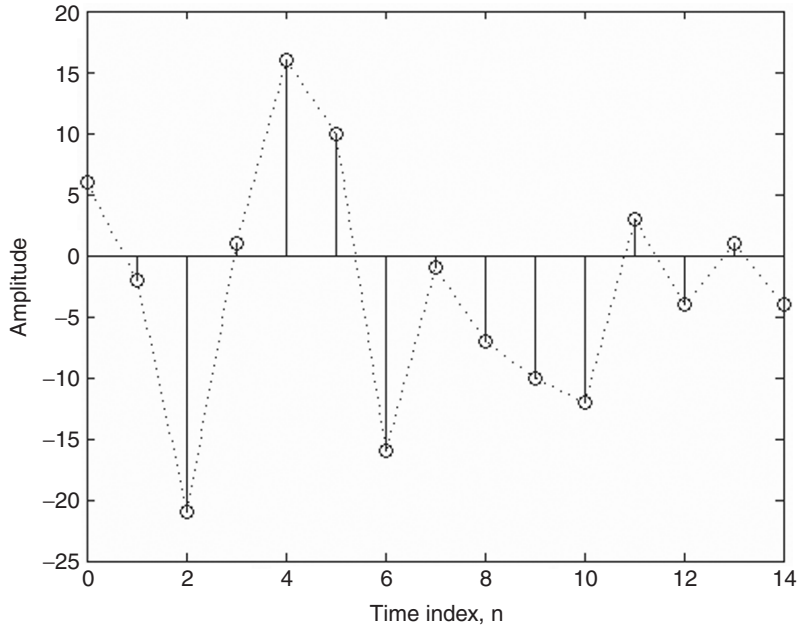

Figure B.1 Graphical representation of a discrete-time signal

Nevertheless, we should also be aware of the pitfall that connecting the sample points may give the impression that the signal is defined for continuous values of the time index n, which is not the case for discrete-time and digital signals. The MATLAB code for plotting the signal in Figure B.1 is given below:

```
un = [6,-2,-21,1,16,10,-16,-1,-7,-10,-12,3,-4,1,-4];
n = [0:length(un)-1];% Time index for un
stem(n,un);          % Plot digital signal
hold  on;
plot(n,un,':');      % Superimpose with continuous curve
```

As mentioned earlier, MATLAB is a powerful tool for simulation of DSP algorithms and applications. The data files (such as `engine_1, engine_2, Timit`, etc., in the Common folder) used for simulation are best coming from data acquisition in real experimental setups, but many users use MATLAB to generate digital signals for computer simulations. Two most frequently used digital signals are sinusoidal and random signals, which can be generated, for example, as follows:

```
fs = 8000;              % Define sampling rate
f = 500;                % Frequency of sinewave
n = 0:1:1023;           % Time index covers 1024
                        % samples
un = sin(2*pi*f*n/fs);  % Generate sinewave
vn = randn(size(n));    % Generate random signal
```

B.2 Filtering and adaptive filtering in MATLAB

An adaptive filter, as shown in Figure 1.2, is a time-varying system that uses a recursive (adaptive) algorithm to continuously adjust its tap weights for operation in an unknown environment. An adaptive filter typically consists of two functional blocks: (i) a digital filter to perform the desired filtering and (ii) an adaptive algorithm to adjust the tap weights of the filter.

As shown in Figure 1.1, an adaptive filter takes two inputs and produces one output. Given an input signal $u(n)$ and desired response $d(n)$, the task of an adaptive filter is to compute the output $y(n)$ from the input signal $u(n)$ such that $y(n)$ approximates the desired response $d(n)$ in some statistical sense. For example, the LMS algorithm minimizes the mean-squared difference between the two signals. The difference between the desired response $d(n)$ and the input signal $y(n)$ is called the error signal $e(n)$. The error signal is used by the adaptive algorithm to adjust the tap weights of the digital filter. Note that for some applications, such as adaptive noise cancellation, the error signal contains an enhanced signal that becomes the desired output of the adaptive system.

Figure 1.3 shows the most commonly used FIR filter structure for adaptive filtering. Its counterpart, the infinite impulse response (IIR) filter, complicates the design of adaptive algorithms due to stability issues. Thus the FIR filter is widely used in practical applications. We restrict the discussion here to the class of adaptive FIR filters, either in the time, subband, frequency or transform domain.

B.2.1 FIR filtering

Figure B.2 shows the block diagram of an adaptive FIR filter using the LMS algorithm. In this section, we focus on the implementation of the filtering block. As depicted in the figure, the FIR filtering block consists of a tapped-delay line and a set of tap weights. The tapped-delay line gathers a block of M samples represented mathematically in vector form as

$$\mathbf{u}(n) \equiv [u(n), u(n-1), \ldots, u(n-M+1)]^T. \tag{B.1}$$

Similarly, the tap weights can be represented in vector form as

$$\mathbf{w}(n) \equiv [w_0(n), w_1(n), \ldots, w_{M-1}(n)]^T. \tag{B.2}$$

Both $\mathbf{u}(n)$ and $\mathbf{w}(n)$ are defined as M-by-1 column vectors, where the index n in the parenthesis represents the time index. The parameter M is called the length of the FIR filter. It indicates the number of adjustable coefficients (i.e. the tap weights) in the filter. Note that an FIR filter of length M has the order of $M-1$. For a causal system, the time index n increases with a step of unity from 0, 1, 2, ... to the end of operation.

Assuming that we use a four-tap FIR filter (i.e. $M = 4$), Figure B.3 illustrates the tapped-delay line operations at the time index `n = 1, 2, 3, 4, 5` and `6` (recall that the MATLAB index starts from 1). At every time instance, the tapped-delay line gathers a block of $M = 4$ samples consisting of one new samples and three (i.e. $M - 1 = 3$) old

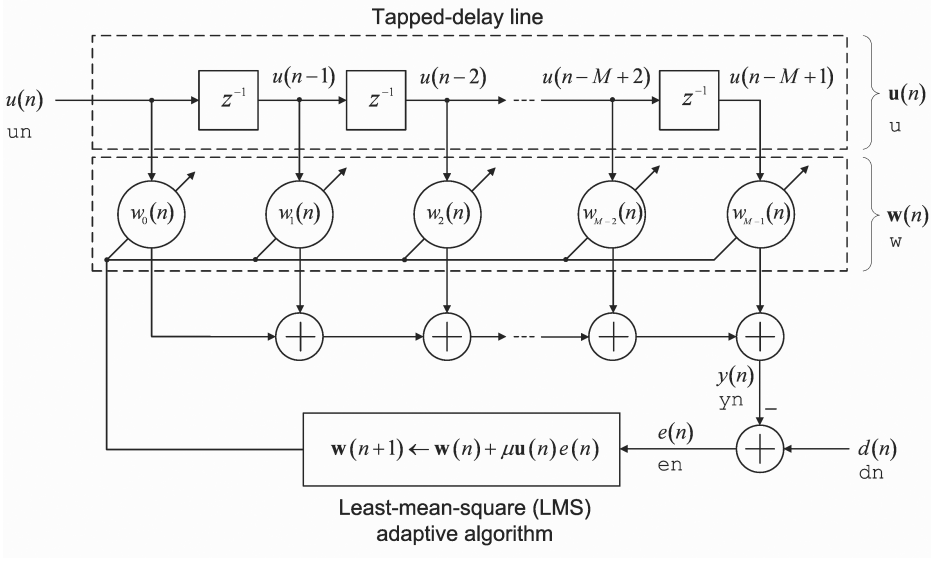

Figure B.2 The LMS algorithm is built based on the FIR filter. The signals $u(n), d(n), y(n)$ and $e(n)$ are represented by the row vectors `un`, `dn`, `yn` and `en` in MATLAB. The column vectors $\mathbf{u}(n)$ and $\mathbf{w}(n)$ of the adaptive filter are represented as column vectors `u` and `w`, respectively

```
Initialize,u = [ 0   0   0   0]'
     n = 1,u = [ 6   0   0   0]'
     n = 2,u = [-2   6   0   0]'
     n = 3,u = [-21 -2   6   0]'
     n = 4,u = [ 1 -21  -2   6]'
     n = 5,u = [16   1 -21  -2]'
     n = 6,u = [10  16   1 -21]'
```

Figure B.3 Tapped-delay line refreshing operation of a four-tap FIR filter

samples. The oldest sample was pushed out from the vector. The following MATLAB code implements the tapped-delay line refreshing operation:

```
un = [6,-2,-21,1,16,10,-16,-1,-7,-10,-12,3,-4,1,-4];
ITER = length(un);
u = zeros(4,1);
for n = 1:ITER
    u = [un(n); u(1:end-1)];
   disp(sprintf('%d\t\t',u));
end
```

The input signal consists of 15 samples represented as a row vector `un` (we used the same vector earlier in Section B.1.2). We check the length of the input signal using the

`length` function. The index `n` starts from 1 as the MATLAB index always starts from 1 instead of 0. At each iteration of the `for` loop, the input vector `u` is updated with a new sample `un(n)`. The oldest sample `un(end)` was pushed out from the vector `u` (recall that the keyword `end` indicates the end of a vector; see Section A.1.3 for details). We initialize the elements of the input vector to zeros in the third line of the code. For this reason, the first vector in Figure B.3 consists of zero-valued elements.

As shown in Figure B.2, the output $y(n)$ of the FIR filter is computed as the weighted sum of the M input samples, where the weights are the adjustable coefficients $w_0(n), w_1(n), \ldots, w_{M-1}(n)$ of the filter. The output $y(n)$ can be expressed in both scalar and vector notations as follows:

$$y(n) = \sum_{m=0}^{M-1} w_m(n)u(n-m) = \mathbf{w}^T(n)\mathbf{u}(n). \tag{B.3}$$

To compute one output sample $y(n)$, the input vector $\mathbf{u}(n)$ is updated with a new input sample (as described above) and the output is given by the inner product of the tap-weight and input vectors. The filtering operation can be implemented in MATLAB by inserting two additional lines into the tapped-delay line code as follows:

```
un = [6,-2,-21,1,16,10,-16,-1,-7,-10,-12,3,-4,1,-4];
ITER = length(un);   % No. of iterations
u = zeros(4,1);      % Clear signal buffer
w = randn(4,1);
for n = 1:ITER
    u = [un(n); u(1:end-1)]; % Update signal buffer
    yn(n) = w'*u;            % Compute filter output
end
```

In this example, we initialize the elements of the tap-weight vector to some random values and the tap weights are held constant throughout the process. As we shall see in the next section, the tap weights of an adaptive filter are updated at every iteration by the adaptive algorithm.

B.2.2 The LMS adaptive algorithm

The coefficients of a digital filter determine the filter characteristics. Based on some specifications (e.g. cut-off frequency and stopband ripples), we can design digital filters (that contain the set of coefficients conforming to the given specifications) and plug the coefficients into the filter structure. Nowadays, filter design can be easily done using software packages such as the Signal Processing Tool (SPTool) and the Filter Design and Analysis Tool (FDATool) presented in Appendix A.

In many practical applications, filter specifications are unknown at the design time. For example, one might need a digital filter that could model the acoustic response of a room. The acoustic responses are different from one room to another. Furthermore, acoustic responses change with time. A better solution for solving this problem is to use a digital filter with time-varying coefficients (i.e. an adaptive filter) to track the unknown yet changing environments. Different from the fixed-coefficient filters, designing an adaptive

filter requires some considerations, such as the filter length, step size, regularization parameters and the adaptive algorithm to be used.

One of the most widely used adaptive algorithms is the LMS algorithm introduced in Section 1.4. The LMS algorithm updates the filter coefficients (i.e. tap weights) as follows:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \underbrace{\mu \mathbf{u}(n) e(n)}_{\text{adjustment}}, \tag{B.4}$$

where μ is the step size that determines the stability and the convergence rate of the algorithm. The time-varying nature of the weight vector $\mathbf{w}(n)$ is indicated by having an index n to show its value as a function of time. Equation (B.4) can be understood as follows. At the current iteration n, the amount of adjustment to the current tap weights $\mathbf{w}(n)$ is determined by the input vector $\mathbf{u}(n)$ (formed with signal samples in the tapped-delay line) scaled with the estimation error $e(n)$ and the step size μ. As shown in Figure B.2, the estimation error (or error signal) $e(n)$ measures the difference between the adaptive filter output $y(n)$ and the desired response $d(n)$. The following MATLAB code implements the LMS algorithm:

```
for n = 1:ITER
    u = [un(n); u(1:end-1)]; % Update signal buffer
    yn(n) = w'*u             % Compute filter output
    en(n) = dn(n) - yn(n);   % Compute error signal
    w = w + (mu*en(n))*u;    % Update weight vector
end
```

At each iteration of the `for` loop, the input vector `u` is updated with the new sample `un(n)`. The filter output `yn(n)` and estimation error `en(n)` are computed using the current weight vector `w`. We then update the tap weights by adding the update term `(mu*en(n))*u` to `w` and assign the result back to `w` for the next iteration. The same operation is repeated until the end of the `for` loop. Notice that `yn` and `en` are defined as vectors. We also keep an account of the changes in the filter output and estimation error for further analysis.

B.2.3 Anatomy of the LMS code in MATLAB

Available on the companion CD, the script and function M-files implementing the LMS algorithm are `LMSinit.m`, `LMSadapt.m` and `LMSdemo.m`. The first function, `LMSinit`, initializes the parameters of the LMS algorithm. The second function, `LMSadapt`, performs the actual computation of the LMS algorithm. The third M-file, `LMSdemo`, is provided as an example of using `LMSinit` and `LMSdapt`.

The function M-files `LMSinit.m` and `LMSadapt.m` are kept in the folder called `Common`, while the demo script `LMSdemo.m` can be found in the folder '`Chapter1`' on the companion CD. See Appendix C for a brief description of the organization of M-files. Complete listings of the code for these three M-files are shown in Tables B.1, B.2 and B.3. In these tables, we have divided the programs into smaller segments. An explanation of the individual segment is given in the following subsection.

B.2.3.1 The `LMSdemo` script

In this script M-file, the LMS algorithm is demonstrated using an FIR filter for adaptive system identification (see Figure 1.5). The complete M-file is listed in Table B.1. A brief description for each statement is given below:

(a) The `addpath` command adds the folder 'Common' to the MATLAB search path since the initialization and adaptation functions (`LMSinit` and `LMSadapt`) are located in the `Common` folder. The `clear all` command removes all the variables from the workspace.

Table B.1 List of `LMSdemo` script M-file

`% LMSdemo  Lest Mean Square (LMS) Algorithm Demo`	
`addpath '..\Common';` `clear all;`	(a)
`mu = 0.001;` `M = 256;`	(b)

`iter = 8.0*80000;`
`b = load('h1.dat');`
`b = b(1:M);`
`[un,dn] = GenerateResponses(iter,b);`
`stepcheck_lms(un,M,mu);`
	(c)

`tic;`
`S = LMSinit(zeros(M,1),mu);`
`S.unknownsys = b;`
`[yn,en,S] = LMSadapt(un,dn,S);`
`EML = S.eml.^2;`
`err_sqr = en.^2;`
`disp(sprintf('Total time = %.3f mins',toc/60));`
	(d)

`figure;`
`q = 0.99; MSE = filter((1-q),[1 -q],err_sqr);`
`hold on; plot((0:length(MSE)-1)/1024,10*log10(MSE));`
`axis([0 iter/1024 -60 10]);`
`xlabel('Number of iterations (\times 1024 input samples)');`
`ylabel('Mean-square error (with delay)');`
`title('LMSdemo');`
`grid on;`
	(e)

`figure;`
`hold on; plot((0:length(EML)-1)/1024,10*log10(EML));`
`xlabel('Number of iterations (\times 1024 input samples)');`
`ylabel('Misalignment (dB)');`
`title('LMSdemo');`
`grid on;`
	(f)

(b) Set the length `M` of the FIR filter to `256` and step-size `mu` to `0.001`. The length of the FIR filter is selected so that it is sufficiently long to cover the impulse response of the unknown plant. The step size `mu` has to be bounded within a certain range according to (1.12).

(c) In practical applications, the unknown system `b` is a physical plant with both input and output connected to the adaptive filter. For experimental purposes, we simulate the unknown system with an FIR filter. The coefficients of the FIR filter are taken from the `h1.dat` data file using the `load` command. We further truncate the length of vector `b` to `M` so that the unknown plant and the adaptive filter have the same length of impulse response. This useful shortcut (only applicable in a controlled simulation) allows us to investigate the convergence behavior of the adaptive filter without considering the over- or undermodeling problem.

Given the coefficients of the unknown plant in vector `b`, the `GenerateResponse` function generates two signal vectors: `un` and `dn`. The input signal `un` is a zero-mean unit-variance random noise. The desired response `dn` is obtained by filtering `un` through the unknown system defined as vector `b`. By default, white noise is added to `dn` as a plant noise with a 40 dB signal-to-noise ratio (SNR). With reference to Figure 1.5, we can now completely simulate the adaptive system identification using `un` as the input signal and `dn` as the desired response (i.e. the reference signal) to the adaptive filter. The variable `iter` determines the duration of the simulation (i.e. number of samples of the signal used for simulation). The `stepcheck_lms` function checks whether a given step size `mu` is appropriate based on the variance of `un`. See Section 1.4 for a discussion on choosing the step size.

(d) The `LMSinit` function performs the initialization of the LMS algorithm. The argument `zeros(M,1)` sets the initial tap weights to zero. The `LMSadapt` function performs the actual computation of the LMS algorithm, where `un` and `dn` are the input and desired signal vectors, respectively. The structure array `S`, from the `LMSinit` function stores the initial state of the LMS algorithm. The same structure array is passed to the `LMSadapt` function. An output signal `yn`, an error signal `en` and the final state `S` are returned by the `LMSadapt` function at the end of the adaptation. The convergence behavior of the adaptive filter can now be analyzed by looking at the squared error `err_sqr` and the misalignment error `EML` (see Equation (4.12)). The last statement displays the total execution time for all operations between the `tic` and `toc` commands. Note that `tic` and `toc` functions work together to measure elapsed time. The `tic` saves the current time that `toc` uses later to measure the elapsed time. The sequence of commands

```
tic;
...
toc;
```

measures the amount of time MATLAB takes to complete the operations and displays the time in seconds.

(e) The `err_sqr` is smoothed along the time axis to approximate the mean-square error (MSE). The plot of the MSE versus time is called the learning curve. For the case where convergence is achieved, the learning curve shows that the MSE gradually

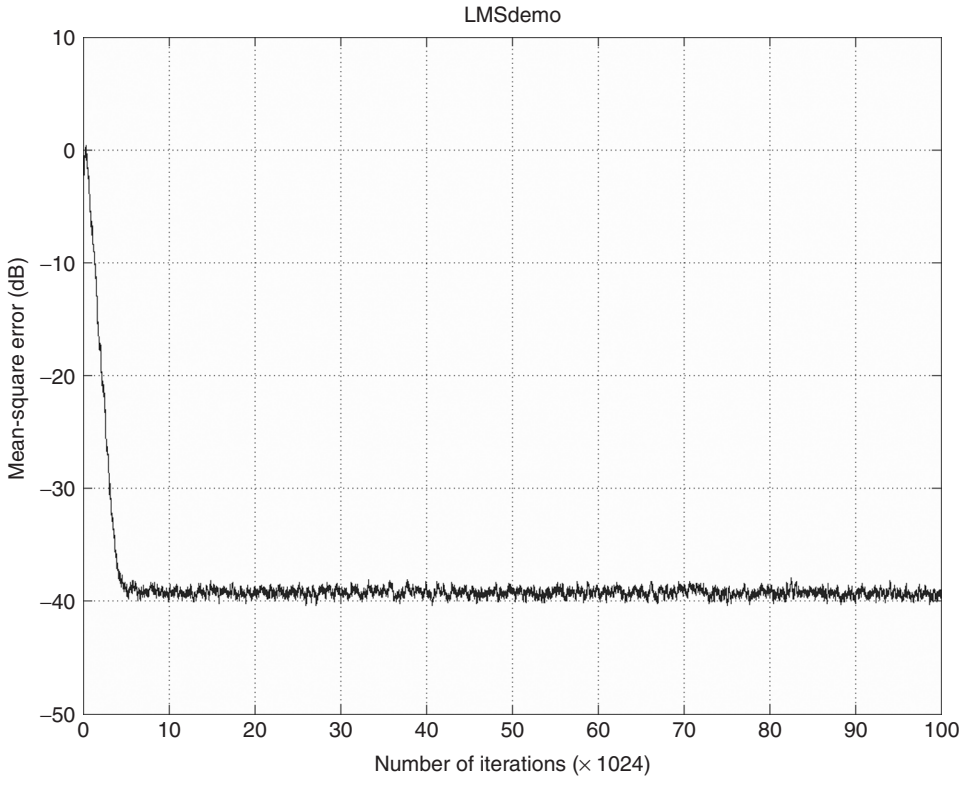

Figure B.4 Mean-square error curve showing the convergence of the LMS algorithm

converges (or decays) to a constant value. We contract the time axis of the MSE plot by 1024 samples in order to analyze the convergence behavior for a longer duration. The resulting MSE plot is shown in Figure B.4.

(f) The misalignment plot indicates the difference between the unknown plant and the adaptive filter at every iteration. The misalignment curve is shown in Figure B.5.

B.2.3.2 The `LMSinit` function

The `LMSinit` function uses the structure array `S` to set up parameters such as the step size, leaky factor and initial weights. These parameters have to be set accordingly in order for the LMS algorithm to estimate the unknown system `b` with sufficient accuracy. The `LMSinit` function is listed in Table B.2. A brief description of each statement is given below:

(a) The `LMSinit` function accepts three input arguments: the initial weights `w0`, the step size `mu` and the leaky factor `leak`. The leaky factor is set to a default value of zero (i.e. the normal form of LMS without leakage) if only two input arguments are given. The variable `nargin` indicates the number of input arguments given to a

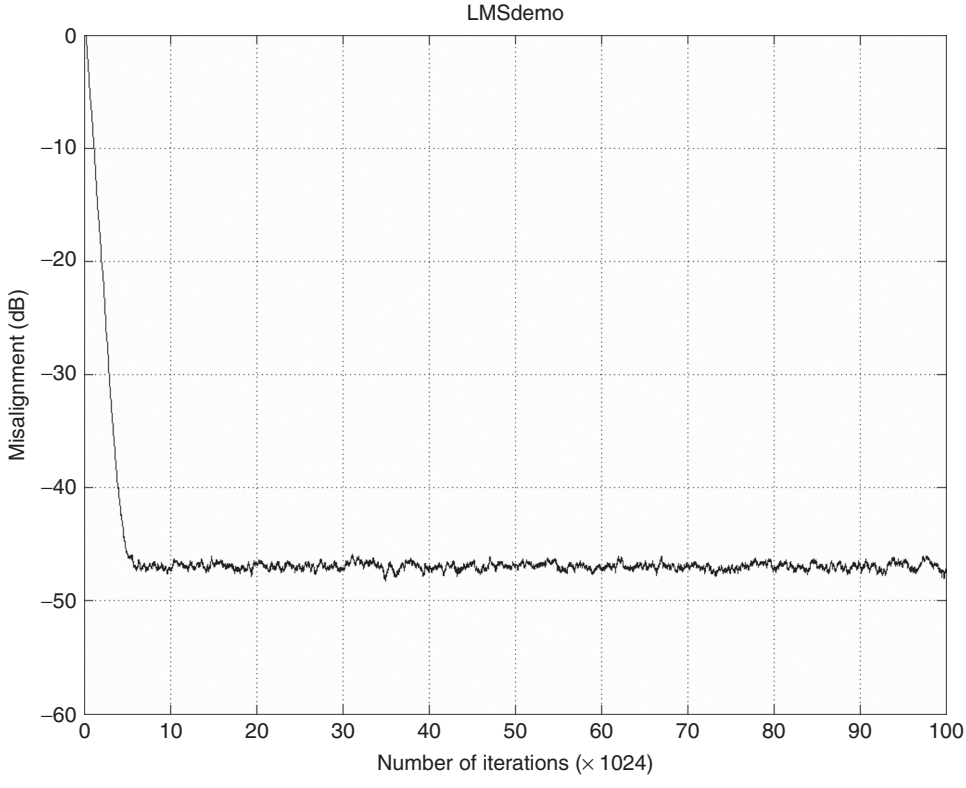

Figure B.5 Misalignment curve showing the convergence of the LMS algorithm

Table B.2 List of `LMSinit` function M-file

```
% function S = LMSinit(w0,mu,leak)                                    (a)

if nargin < 3
        leak = 0;
end
```

```
S.coeffs          = w0(:);
S.step            = mu;
S.leakage         = leak;                                            (b)
S.iter            = 0;
S.AdaptStart      = length (w0);
```

MATLAB function. The `LMSinit` function returns the structure array S containing all the parameters required by the `LMSadapt` function, which performs the system identification.

(b) `S` is a structure array with five fields: `coeffs, step, leakage, iter` and `AdaptStart`. The first three fields are set according to the value passed to the function. Notice that the length `M` of the adaptive filter is automatically determined by the length of the initial vector `w0` given to the `LMSinit` function. The `iter` field is used to record the number of iterations of the weight adaptation. The last field, `AdaptStart`, is used in the `LMSadapt` function to start the weight adaptation when the iteration number exceeds the value assigned to `AdaptStart`. The initial value of `AdaptStart` is set equal to the length `M` of the adaptive filter. By doing so, the weight adaptation is performed after the FIR filter has completely 'run on to' the input sequence `un`.

B.2.3.3 The `LMSadapt` function

The `LMSadapt` function, as shown in Table B.3, accepts three input arguments: the input signal `un`, the desired response `dn` and the parameter array `S`. In the `LMSdemo` script presented earlier, `dn` is generated by filtering `un` using the 'unknown' plant `b`. Given `un` and `dn` as input arguments, the `LMSadapt` function adjusts the coefficients of its FIR filter such that the filter output `yn` is close to the desired response `dn` in an MSE sense. The adaptive tap weights `w` are updated iteratively toward the unknown plant `b` using samples of `un` and `dn` in the adaptation. A brief description for each statement is given below:

(a) The structure array, `S`, provides a convenient way of passing the parameters between the `LMSinit` and `LMSadapt` functions. Instead of a bunch of MATLAB variables, the parameters of the LMS filter are initialized and packed into the structure array `S`, which are then unpacked and used in the `LMSadapt` function.

(b) The LMS algorithm given in Equation (B.4) is a sample-based algorithm, where the weight adaptation is performed sample by sample. The number of samples contained in the input signal vector `un` determines the number of iterations, `ITER`. We also pre-allocate two row vectors, `yn` and `en`, to store the output and error signals generated during the adaptation process. The column vector `u` represents the input signal vector $\mathbf{u}(n)$ defined in Equation (B.1), the length of which is the same as the weight vector $\mathbf{w}(n)$. Notice that we use `un` to represent the input signal $u(n)$, as shown in Figure B.2.

(c) The normalized misalignment is a measure of the difference between the unknown plant and the adaptive filter at each iteration. If `b` is assigned to the structure array `S` prior to calling the `LMSadapt` function (see statement (d) of Table B.1), the `ComputeEML` flag is set to 1 and the misalignment `EML` will be computed during the adaptation process.

(d) The LMS adaptation is implemented using a `for` loop. At each iteration, the following operations are performed in sequence:

(i) Update the tap-input vector `u` with a new sample `un(n)`; the oldest sample `u(end)` is pushed out from the vector.

Table B.3 List of function M-file `LMSadapt`

```
% function [yn,en,S] = LMSadapt(un,dn,S)
```

(a)

```
M = length(S.coeffs);
mu = S.step;
leak = S.leakage;
AdaptStart = S.AdaptStart;
w = S.coeffs;
```

(b)

```
u = zeros(M,1);
ITER = length(un);
yn = zeros(1,ITER);
en = zeros(1,ITER);
```

(c)

```
if isfield(S,'unknownsys')
     b = S.unknownsys;
     norm_b = norm(b);
     eml = zeros(1,ITER);
     ComputeEML = 1;
else
     ComputeEML = 0;
end
```

(d)

```
for n = 1:ITER
     u = [un(n); u(1:end-1)];
     yn(n) = w'*u;
     en(n) = dn(n) - yn(n);
     if ComputeEML == 1;
          eml(n) = norm(b-w)/norm_b;
     end
     if n >= AdaptStart
          w = (1-mu*leak)*w + (mu*en(n))*u;
          S.iter = S.iter + 1;
     end
end
```

(e)

```
S.coeffs = w;
if ComputeEML == 1;
     S.eml = eml;
end
```

(ii) Compute the FIR filter output `yn`, which is given by the inner product between the current adaptive weight vector `w` and the tap-input vector `u`.

(iii) Calculate the error `en` between the filter output and the desired response.

(iv) Update the adaptive filter coefficients. Equation (B.4) gives the adaptation equation for the case where the leaky factor `leak` $= 0$. The filter coefficients are updated by adding `(mu*en(n))*u` to `w` and assign the result back to `w` for the next iteration. By setting `AdaptStart` equal to the length of the adaptive filter `M`, the weight adaptation will only be performed after the filter has completely 'run on to' the input sequence `un`.

(e) Finally, the updated filter coefficients `w` and the misalignment error `EML` are packed on to the structure array `S` and passed back to the calling script.

B.3 Multirate and subband adaptive filtering

In this section, we describe the MATLAB implementation of multirate filter banks and subband adaptive filters. Readers are encouraged to refer to Chapter 2 for a more detailed description of subband and multirate systems and Chapter 4 for the fundamental principles of subband adaptive filtering.

B.3.1 Implementation of multirate filter banks

B.3.1.1 Decimator and interpolator

Digital signal processing systems that use more than one sampling rate are referred to as multirate systems. Two basic sampling rate conversion devices employed in multirate systems are the decimator and interpolator (see Section B.1 for more details).

A decimator with a decimation factor D, where D is a positive integer, reduces the sampling rate of an input sequence by a factor of D. The decimator is implemented by keeping every Dth sample of the input sequence, while removing other samples. On the other hand, an interpolator with an interpolation factor of I, where I is a positive integer, increases the sampling rate of the input signal by inserting $(I - 1)$ zero samples between an adjacent pair of input samples. The decimation and interpolation can be easily implemented in MATLAB using the colon ':' operator. The MATLAB scripts `Example_2P1.m` and `Example_2P2.m` demonstrate the operations of decimation and interpolation using a sinusoid as the input signal. Table B.4 lists these two files. The explanation for each code segment is given below:

(a) We set the decimation and interpolation factors, `D` and `I`, to be the same. In the following, we first decimate the signal `x` with a factor of `2` and then perform interpolation on the decimated sequence with the same factor. `L` is the length of the lowpass filter `h_LP`, which removes unwanted high-frequency components resulting from zero-insertions.

(b) The input signal `x` is a 50 Hz (`f = 50`) sinusoid with a sampling rate of 1 kHz (`fs = 1000`). The term `2*pi*(freq/fs)` in the `sin` function gives a normalized frequency of 0.1π.

Table B.4 List of MATLAB scripts from `Example_2P1` and `Example_2P2`

```
% Example_2P1 and Example_2P2
```

```
clear all;
D = 2;
I = 2;                                                    (a)
L = 17;
h_LP = firl(L-1,1/2);
```

```
fs = 1000; freq = 50; N = 100;
n = 0:N-1;                                                (b)
x = sin(2*pi*(freq/fs)*n);
```

```
x_D = x(1:D:end);
x_I = zeros(1,length(x));
x_I(1:I:end) = x_D;                                       (c)
y = filter(h_LP,1,x_I);
```

```
figure;
subplot(4,1,1); stem(0:20, x(1:21));
xlabel('Time index, n');
subplot(4,1,2); stem(0:10, x_D(1:11));
xlabel('Time index, k');                                  (d)
subplot(4,1,3); stem(0:20, x_I(1:21));
xlabel('Time index, n');
subplot(4,1,4); stem(0:20, y((1:21)+(L-1)/2)*I);
xlabel('Time index, n');
```

(c) For the decimation, every Dth samples of `x` is selected using `[1:D:end]`, where `D` indicates the incremental step and the keyword `end` indicates the end of the row vector `x`. The signal `x_D` is then interpolated with the same factor of `2` by inserting a zero sample between an adjacent pair of samples in `x_D`. The original, decimated and interpolated sequences are shown in the first three panels of Figure B.5. The following observations can be made:

(i) The sampling rate of `x_D` (in the second panel) is half of the sampling rate of the original sequence `x` and the interpolated sequence `x_I`.

(ii) The sampling rate is recovered in `x_I` by inserting zeros. However, interpolation alone does not reconstruct the original signal `x`.

(iii) The original input `x` can be reconstructed with interpolation followed by lowpass filtering. The lowpass filter `h_LP` removes the high-frequency images resulting from inserting zero-valued samples. The reconstructed signal `y` is shown in the last panel of Figure B.6.

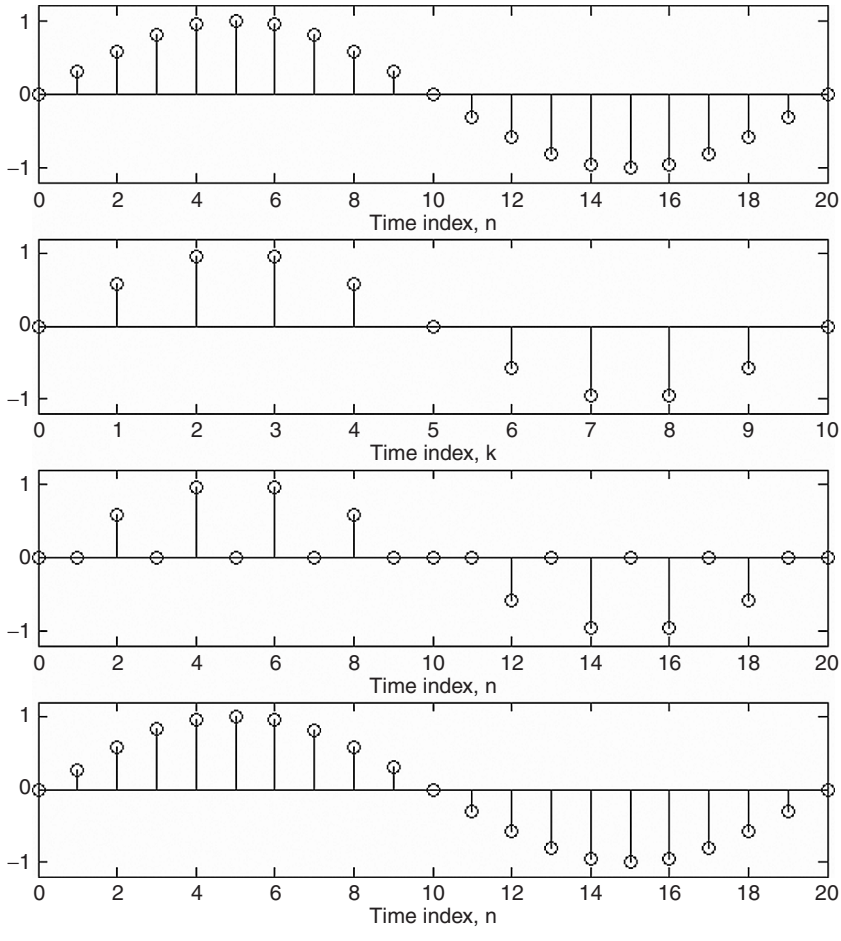

Figure B.6 The input signal `x` (first panel) is decimated with a factor of `2`. The decimated signal `x_D` (second panel) is then interpolated with a factor of `2`. The interpolated signal `x_I` (third panel) is finally lowpass filtered to obtain the output signal `y` (fourth panel), which is the reconstructed signal after sampling rate conversion

(d) We plot the first cycle of the sinusoids. The output signal `y` shows that there is a delay of `(L-1)/2` samples caused by the lowpass filter. We compensate this delay in the `plot` function.

B.3.1.2 DFT Filter bank as a block transform

Figure 2.3 shows an N-channel filter bank. The left side of the figure is the analysis filter bank consisting of a set of bandpass filters with a common input. The right side of the figure is the synthesis filter bank with a summed output from another set of bandpass filters. Filter banks can be implemented as a block transform with memory. The idea

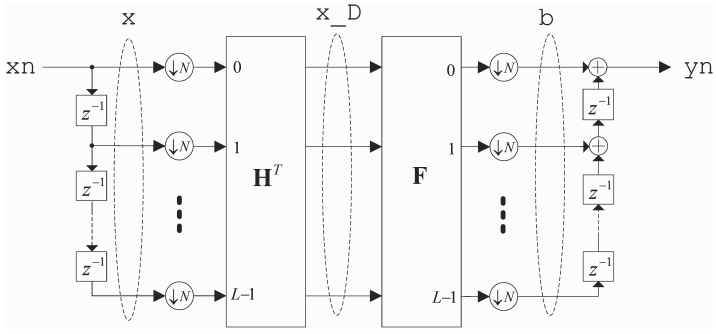

Figure B.7 A DFT filter bank can be implemented as a block transform with memory

is illustrated in Figure B.7 and Table B.5. In the following, we give a step-by-step explanation of the MATLAB implementation for the DFT filter bank:

(a) Design a prototype lowpass filter `hopt` using the `fir1` function available from the Signal Processing Toolbox. The cutoff frequency (second argument in the `fir1` function) is set at `1/N`, which is determined by the number of subbands `N = 8`. The DFT filter bank is then generated via complex modulation (see Section 2.6) of the prototype filter using the `make_bank_DFT` function. The length of the analysis filter is `L = 256`. The magnitude response of the filter bank is shown in Figure 2.14.

The coefficients of the analysis and synthesis are stored in matrices `H` and `F`, respectively. Both `H` and `F` are `256`-by-`8` matrices. Each column of `H` holds the coefficients of an analysis filter, with the first column corresponding to the first analysis filter and so on. Similar representation applies for the synthesis filter bank `F`. The scaling factor `sqrt(N)` ensures a unity gain for the combined analysis–synthesis filter bank.

(b) The input signal `xn` to the filter bank is a sinusoid with normalized frequency of 0.02π. With this frequency (see Figure 2.14), the signal `xn` essential passes through only the first subband while attenuated in other subbands. We pre-allocate another row `yn` for the synthesized output of the filter bank. We also define two row vectors, `x` and `b`, as the signal buffers at the analysis and synthesis sections, respectively. The length of the buffers is determined by the length `L` of the prototype filter.

(c) The forward and inverse transformations (i.e. the analysis and synthesis filter banks) of the input signal `xn` is implemented by a `for` loop. The number of `ITER` is given by the number of samples in `xn`. At each iteration, the following operations are performed in sequence:

(i) Update the input buffer `x` with a new input sample `xn(n)`. The oldest sample in the buffer `xn(end)` is discarded.

(ii) Block transformations are performed once for every `N` input samples. This can be accomplished using the `mod(n,N)` function, which returns a `0` when the iteration index `n` is multiples of `N`. The result of the forward transform `H.'*x` is assigned to an `N`-by-1 column vector `x_D`. Elements of the transformed vector

Table B.5 DFT filter bank as a block transform

```
addpath '..\common';
clear all;
```

(a)

```
N = 8;
k = 16;
L = 2*K*N;
hopt = firl(L-1,1/N);
[H,F] = make_bank_DFT(hopt,N);
H = sqrt(N)*H; F = sqrt(N)*F;
```

(b)

```
xn = sin(2*pi*0.01*(0:1000));
ITER = length(xn);
yn = zeros(ITER,1);
x = zeros(length(H),1);
b = zeros(length(F),1);
```

(c)

```
for n = 1:ITER
        x = [xn(n); x(1:end-1)];
        if mod(n,N)==0
           x_D = H.'*x;
           %============================
           % Insert subband processing
           % algorithm here
           %============================
           b = F*x_D + b;
        end
    yn(n) = real(b(1));
    b = [b(2:end); 0];
end
```

(d)

```
figure;
plot(0:length(xn)-1,xn); hold on;
plot(0:length(yn)-1,yn,'r:');
xlabel('Time index, n'); ylabel('Amplitude');
```

`x_D` are in fact the subband signals, which serve as inputs to subband processing algorithms. The inverse transform is given by `F*x_D`. The result is added to the output buffer `b`.

(d) The fullband output sample `yn(n)` is given by the first element `b(1)` of the output signal buffer. After pushing out the first element, the output buffer `b` is then updated with a `0` sample (recall that an interpolator inserts zero samples) to be used in the next iteration. The input and output signals are shown in Figure B.8. Notice that the output signal is delayed by `L` $- 1 = 255$ samples due to the filter bank.

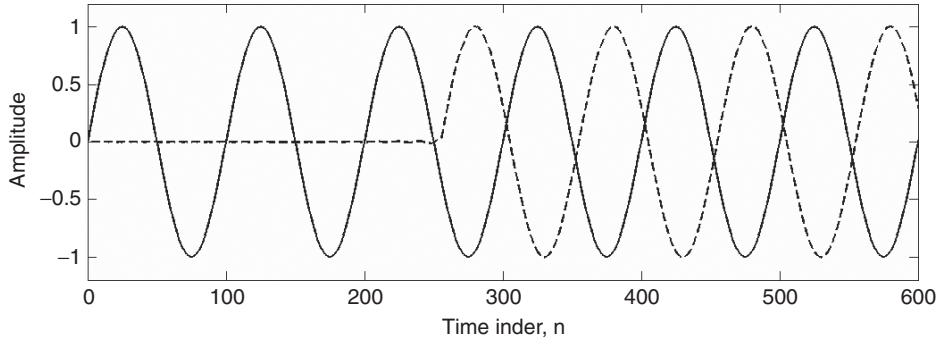

Figure B.8 The output signal (dashed line) is a delayed version of the input signal (solid line)

B.3.2 Implementation of a subband adaptive filter

An analysis filter bank decomposes a fullband signal into multiple subbands. The fullband signal can be reconstructed via a synthesis filter bank, as we have seen in Section B.3.1. In its conventional form, a subband adaptive filter (SAF) is constructed by inserting N adaptive subfilters (N is the number of subbands or channels), one between each of the analysis–synthesis filter pairs. Recall that an adaptive filter requires a reference signal $d(n)$ (or desired response) in addition to the input signal $u(n)$. Therefore, we need two analysis filter banks and one synthesis filter bank. The conventional form of the SAF is shown in Figure 4.1.

Available on the companion CD, the script and function M-files implementing the SAF are `SAFinit.m`, `SAFadapt.m` and `SAFdemo.m`. The first function, `SAFinit`, initializes the parameters (e.g. step size, initial weights, number of subbands, filter banks, etc.) of the SAF. The second function, `SAFadapt`, performs the actual computation of the SAF algorithm. The third M-file, `SAFdemo`, is provided as an example of using functions `SAFinit` and `SAFdapt`.

The function M-files `SAFinit.m` and `SAFadapt.m` are kept in the folder '`Common`', while the demo script `SAFdemo.m` can be found in the folder '`Chapter4`' on the companion CD. The structures of the `SAFdemo` and `SAFinit` are similar to that of the LMS algorithm presented in Section B.2.3. We will only focus on the `SAFadapt` function in this section. Partial listing of the MATLAB function is shown in Table B.6, where we focus only on the adaptation loop. The explanation for each block is given below. Figure B.9 shows the location of various signal vectors:

(a) Similar to that in Section B.3.1.2, `x` and `y` are column vectors representing the signal buffers of analysis filter banks for the input signal `un` and desired response `dn`, respectively. The length of these buffers is determined by the length `L` of the analysis and synthesis filters. For each iteration in the `for` loop, the signal buffers, `x` and `y`, are updated with new samples.

(b) For the input signal, the transformed vector `x'*H` is used to update the matrix `U`. Each column of `U` represents the signal vector for an adaptive subfilter. Similarly,

Table B.6 Partial listing of the `SAFadapt` function M-file

```
function [en,S] = SAFadapt (un,dn,S)

for n = 1:ITER                                                    (a)
  x = [un(n); x(1:end-1)];
  y = [dn(n); y(1:end-1)];

  if (mod(n-1,D)==0)                                              (b)
    U = [x'*H; U(1:end-1,:)];
    dD = y'*H;
    eD = dD - sum(U.*w);

    if n >= AdaptStart                                            (c)
      W = W + conj(U)*diag(eD./(sum(U.*conj(U))+alpha))*mu;
      S.iter = S.iter + 1;
    end

    eD = [eD, conj(fliplr(eD(2:end-1)))].';                       (d)
    z = F*eD + z;
  end
  en(n) = real(z(1));
  z = [z(2:end); 0];
end
```

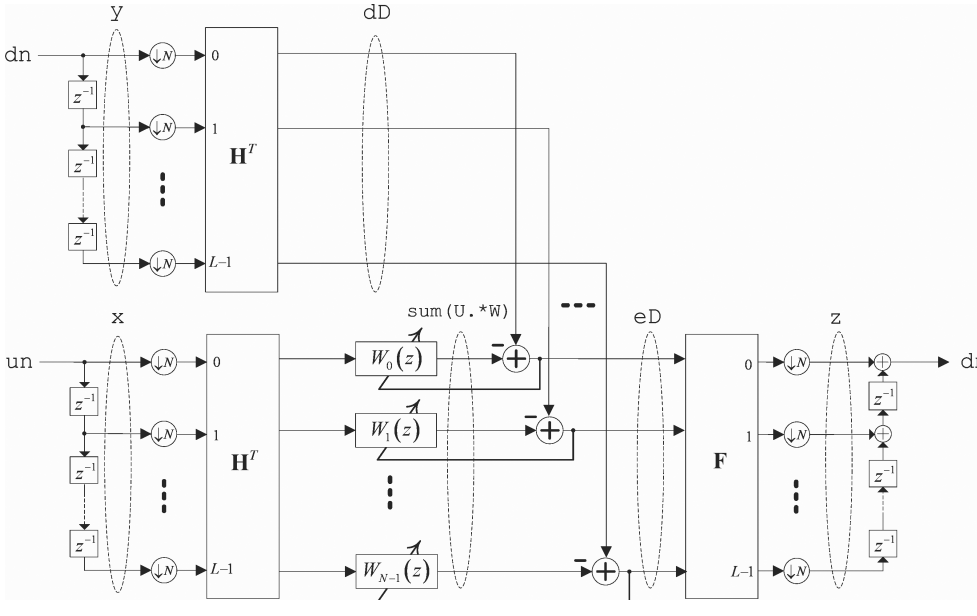

Figure B.9 Subband adaptive filter using DFT filter banks

each column of the matrix `W` represents the weight vector of an adaptive subfilter. The subband estimation errors `eD` are computed once in every `D` iterations.

(c) Tap-weight adaptation is performed using the NLMS algorithm (see Section 4.4.4). Remember that each column of `W` represents an adaptive subfilter.

(d) Inverse transformation of the subband estimation errors gives rise to the fullband error signal `en`. Notice that we only process the first $N/2+1$ subbands (instead of N subbands) by means of the complex-conjugate relation between subbands (see Section 4.1.1 for details). The estimation errors in the remaining subbands are obtained from other subbands using the complex-conjugate properties.