Adaptive Filters 6

6.1 INTRODUCTION

Fixed-frequency-response digital filters were discussed in the two previous chapters. This chapter looks at filters with a frequency response, or transfer function, that can change over time to match desired system characteristics.

Many computationally efficient algorithms for adaptive filtering have been developed within the past twenty years. They are based on either a statistical approach, such as the least-mean square (LMS) algorithm, or a deterministic approach, such as the recursive least-squares (RLS) algorithm. The major advantage of the LMS algorithm is its computational simplicity. The RLS algorithm, conversely, offers faster convergence, but with a higher degree of computational complexity.

The adaptive filter algorithms discussed in this chapter are implemented with FIR filter structures. Since adaptive FIR filters have only adjustable zeros, they are free of stability problems that can be associated with adaptive IIR filters where both poles and zeros are adjustable. Of the various FIR filter structures available, the direct form (transversal), the symmetric transversal form, and the lattice form are the ones often employed in adaptive filtering applications.

6.1.1 Applications Of Adaptive Filters

Adaptive filters are widely used in telecommunications, control systems, radar systems, and in other systems where minimal information is available about the incoming signal.

Due to the variety of implementation options for adaptive filters, many aspects of adaptive filter design, as well as the development of some of the adaptive algorithms, are governed by the applications themselves. Several applications of adaptive filters based on FIR filter structures are described below.

6.1.1.1 System Identification

You can design controls for a dynamic system if you have a model that describes the system in motion. Modeling is not easy with complex physical phenomena, however. You can get information about the system to be controlled from collecting experimental data of system responses to given excitations. This process of constructing models and estimating the best values of unknown parameters from experimental data is called *system identification.*

Figure 6.1 shows a block diagram of the system identification model. The unknown system is modeled by an FIR filter with adjustable coefficients. Both the unknown time-variant system and FIR filter model are excited by an input sequence *u(n).* The adaptive FIR filter output *y(n)* is compared with the unknown system output *d(n)* to produce an estimation error *e(n).* The estimation error represents the difference between the unknown system output and the model (estimated) output. The estimation error *e(n)* is then used as the input to an adaptive control algorithm which corrects the individual tap weights of the filter. This process is repeated through several iterations until the estimation error *e(n)* becomes sufficiently small in some statistical sense. The resultant FIR filter response now represents that of the previously unknown system.

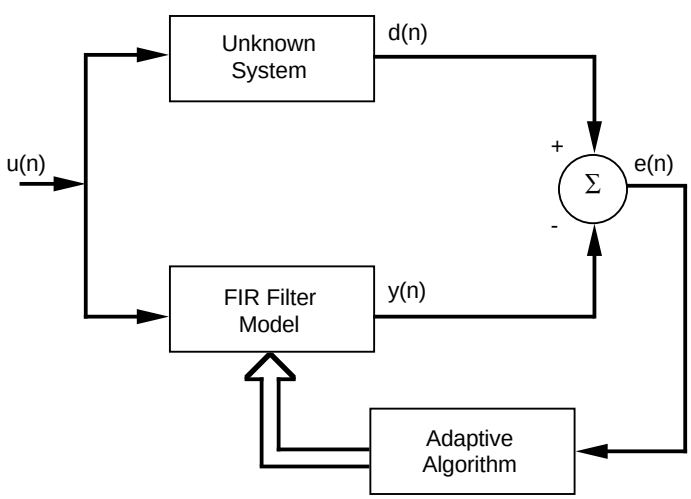

Figure 6.1 System Identification Model

6.1.1.2 Adaptive Equalization For Data Transmission

Adaptive filters are used widely to provide equalization in data modems that transmit data over speech-band and wider bandwidth channels. An adaptive equalizer is employed to compensate for the distortion caused by the transmission medium. Its operation involves a training mode followed by a tracking mode.

Initially, the equalizer is trained by transmitting a known test data sequence $u(n)$. By generating a synchronized version of the test signal in the receiver, the adaptive equalizer is supplied with a desired response $d(n)$. The equalizer output $y(n)$ is subtracted from this desired response to produce an estimation error, which is in turn used to adaptively adjust the coefficients of the equalizer to their optimum values. When the initial training period is completed, the adaptive equalizer tracks possible time variations in channel characteristics during transmission by using a receiver estimate of the transmitted sequence as a desired response. The receiver estimate is obtained by applying the equalizer output $y(n)$ to a decision device.

6.1.1.3 Echo Cancellation For Speech-Band Data Transmission

Dial-up switched telephone networks are used for low-volume infrequent data transmission. A device called a "hybrid" provides full-duplex operation, transmit and receive channels, from a two-wire telephone line. Due to impedance mismatch between the hybrid and the telephone channel, an "echo" is generated which can be suppressed by adaptive echo cancellers installed in the network in pairs. The cancellation is achieved by making an estimate of the echo signal components using the transmitted sequence $u(n)$ as input data, and then subtracting the estimate $y(n)$ from the sampled received signal $d(n)$. The resulting error signal can be minimized, in the least-squares sense, to adjust optimally the weights of the echo canceller.

Similar applications include the suppression of narrowband interference in a wideband signal, adaptive line enhancement, and adaptive noise cancellation.

6.1.1.4 Linear Predictive Coding of Speech Signals

The method of linear predictive coding (LPC) is an example of a source coding algorithm used for the digital representation of speech signals. So called source coders are model dependent: they use previous knowledge of how the speech signal was generated at its source. Source coders for speech are generally referred to as *vocoders* and can operate at data rates of 4.8 Kbits/s or below. In LPC, the source vocal tract is modeled as a linear all-pole filter whose parameters are determined adaptively from speech samples by means of linear prediction. The speech samples *u(n)* are, in this case, the desired response, while *u(n-1)* forms the inputs to the adaptive FIR filter known as a prediction error filter. The error signal between u(n) and the output of the FIR filter, *y(n)*, is then minimized in the least-squares sense to estimate the model parameters. The error signal and the model parameters are encoded into a binary sequence and transmitted to the destination. At the receiver, the speech signal is synthesized from the model parameters and the error signal.

6.1.1.5 Array Processing

Adaptive antenna arrays use processing techniques that are very similar to those of adaptive filters. They use the spatial separation between the antenna elements to provide a parallel set of signal samples rather than using the time-delayed or partly processed versions of a one-dimensional input signal. Their applications include bearing estimation and adaptive beamforming.

6.1.2 FIR Filter Structures

An FIR system has a finite-duration impulse response that is zero outside of some finite time interval. Thus, an FIR system has a finite memory of length-*N* samples. Three basic structures for realizing the FIR filter (transversal, symmetric, and lattice) are described below.

6.1.2.1 Transversal Structure

Figure 6.2 shows the structure of a transversal FIR filter with N tap weights (adjustable during the adaptation process) with values at time n denoted as

$$w_0(n),\ w_1(n),\ \ldots,\ w_{N-1}(n).$$

The tap-weight vector, w(n), is represented as

$$\underline{w}(n) = [w_0(n)\ w_1(n)\ \ldots\ w_{N-1}(n)]T$$

the tap-input vector, $\underline{u}$(n), as

$$\underline{u}(n) = [u(n)\ u(n-1)\ \ldots\ u(n-N+1)]T$$

The FIR filter output, $y(n)$, can then be expressed as

$$y(n) = \underline{w}^T(n)\ \underline{u}(n) = \sum_{i=0}^{N-1} w_i(n)\ u(n-i)$$

where T denotes transpose, n is the time index, and N is the order of the filter.

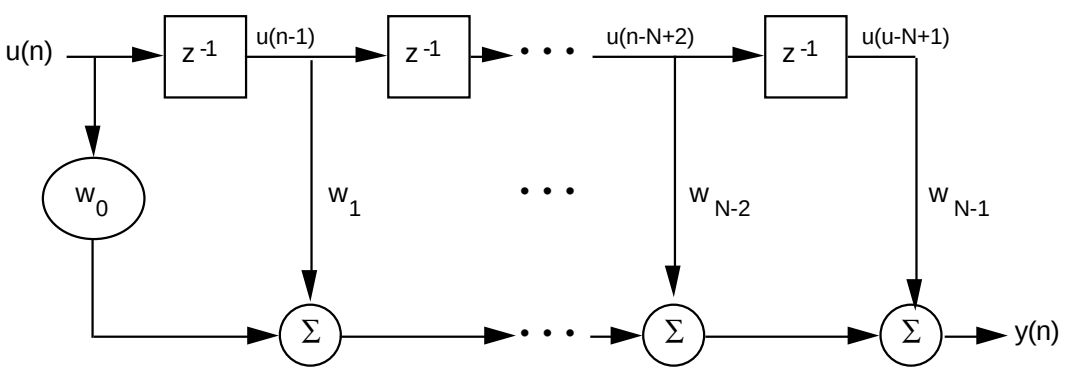

Figure 6.2 Transversal FIR Filter Structure

6.1.2.2 Symmetric Transversal Structure

The characteristic of linear phase response in a filter is sometimes desirable because it allows a system to reject or shape energy bands of the spectrum and still maintain the basic pulse integrity with a constant filter group delay. Imaging and digital communications are examples of applications where this characteristic is desirable.

An FIR filter with time domain symmetry, such as

$$w_0(n) = w_{N-1}(n),\ w_1(n) = w_{N-2}(n)...$$

has a linear phase response in the frequency domain. Consequently, the number of weights is reduced by a half in a transversal structure, as shown in Figure 6.3 with an even N tap weights. The tap-input vector becomes

$$u(n) = [u(n) + u(n-N+1),\ u(n-1) + u(n-N+2),\ ...,\ u(n-N/2+1) + u(n-N/2)]\ T$$

As a result, the filter output $y(n)$ becomes

$$y(n) = \sum_{i=0}^{N/2} w_i(n)[u(n-i) + u(n-N+1+i)]$$

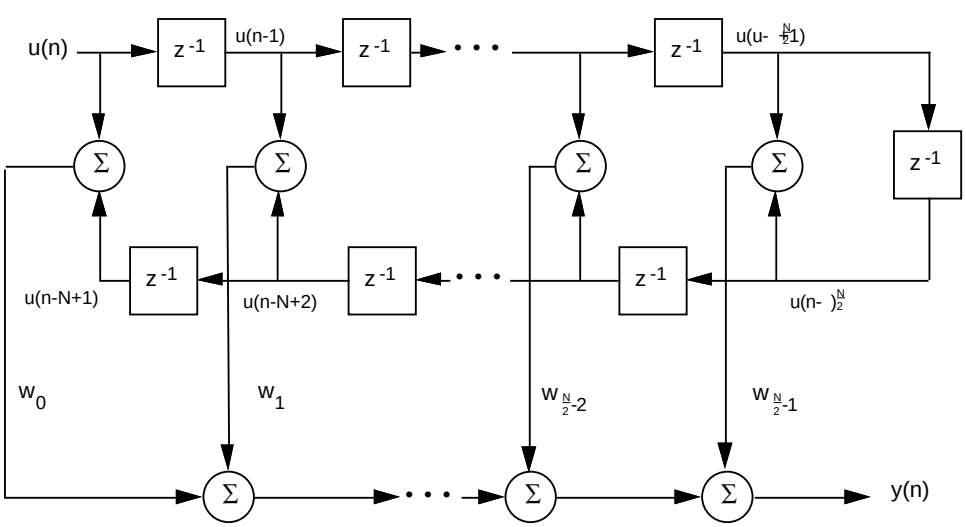

Figure 6.3 Symmetric Transversal Filter Structure

6.1.2.3 Lattice Structure

The lattice filter has a modular structure with cascaded identical stages. Figure 6.4 shows one stage of a lattice FIR structure.

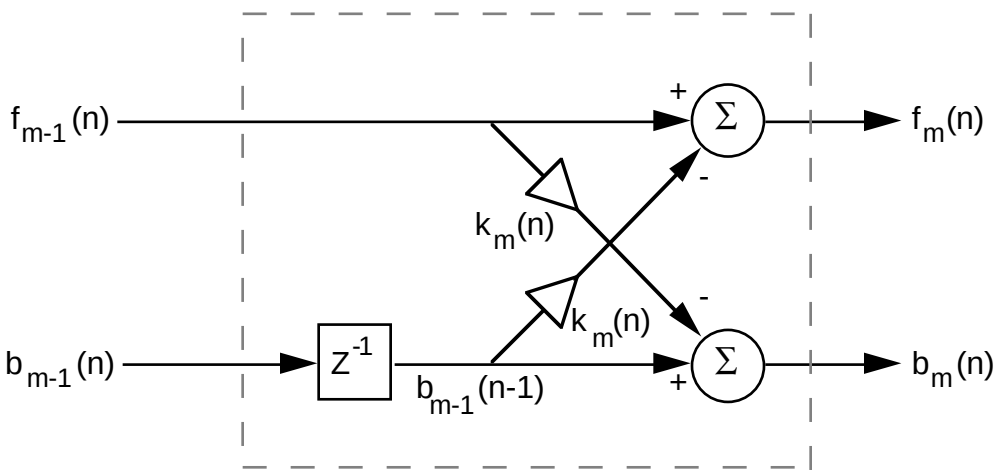

Figure 6.4 One Stage Of Lattice FIR

The lattice structure offers several advantages over the transversal structure:

- The lattice structure has good numerical round-off characteristics that make it less sensitive than the transversal structure to round-off errors and parameter variations.
- The lattice structure orthogonalizes the input signal stage-by-stage, which leads to fast convergence and efficient tracking capabilities when used in an adaptive environment.
- The various stages are decoupled from each other, so it is relatively easy to increase the prediction order if required.
- The lattice filter (predictor) can be interpreted as wave propagation in a stratified medium. This can represent an acoustical tube model of the human vocal tract, which is extremely useful in digital speech processing.

These advantages, however, come at the expense of an increased number of multiplies and adds for a given transfer function realization.

The following equations represent the dynamics of the *m*th stage of an order *M* lattice structure as derived from Figure 6.4:

$$f_m(n)=f_{m-1}(n) - K_m(n)\, b_{m-1}(n-1),\ 0 < m < M$$
$$b_m(n)=b_{m-1}(n-1)\, K_m(n)\, f_{m-1}(n),\ 0 < m < M$$

where $f_m(n)$ represents the forward prediction error, $b_m(n)$ represents the backward prediction error, $K_m(n)$ is the reflection coefficient, m is the stage index, and M is the number of cascaded stages. $K_m(n)$ has a magnitude less than one. The terms $f_m(n)$ and $b_m(n)$ are initialized as

$$f_0(n) = b_0(n) = u(n)$$

where $u(n)$ is the input signal.

Speech analysis is usually performed by using the lattice structure and the reflection coefficients $K_m(n)$. Since the dynamic range of $K_m(n)$ is significantly smaller than that of the tap weights, $w(n)$, of a transversal filter, the reflection coefficients require fewer bits to represent them. Hence, $K_m(n)$ are transmitted over the channel.

6.1.3 Adaptive Filter Algorithms

Two types of adaptive algorithms are discussed in this section: least-mean square (LMS) and recursive least-squares (RLS). LMS algorithms are based on a gradient-type search for tracking time-varying signal characteristics. RLS algorithms provide faster convergence and better tracking of time-variant signal statistics than LMS algorithms, but are more complex computationally.

6.1.3.1 The LMS Algorithm

The LMS algorithm is initialized by setting all the weights to zero at time n=0. Tap weights are updated using the relationship

$$\underline{w}(n+1) = \underline{w}(n) + \mu e(n)\underline{u}(n)$$

where $w(n)$ represents the tap weights of the transversal filter, $e(n)$ is the error signal, $u(n)$ represents the tap inputs, and the factor μ is the adaptation parameter or step-size. To ensure convergence, μ must satisfy the condition

$$0 < \mu < (2 \,/\, \textit{total input power})$$

where the *total input power* refers to the sum of the mean-square values of the tap inputs u*(n), u(n-1), ..., u(n-N+1)*. Moreover, the LMS convergence time depends on the ratio of maximum to minimum eigenvalues of the autocorrelation matrix R of the input signal.

To insure that μ does not become sufficiently large to cause filter instability, a Normalized LMSalgorithm can be employed. The normalized LMS employs a time-varying *mu* defined as

$$mu = \frac{x}{\underline{u}T(n).\ \underline{u}(n))}$$

where x is the normalized step-size chosen between 0 and 2. Tap weights are updated according to the relationship

$$w(n+1) = w(n) + \frac{xe(n)\underline{u}(n)}{r+\underline{u}T(n)\underline{u}(n)}$$

The term *x* is the new normalized adaptation constant, while *r* is a small positive term included to ensure that the update term does not become excessively large when $u_T(n)u(n)$ temporarily becomes small.

A problem can occur when the autocorrelation matrix associated with the input process has one or more zero eigenvalues. In this case, the adaptive filter will not converge to a unique solution. In addition, some uncoupled coefficients (weights) may grow without bound until hardware overflow or underflow occurs. This problem can be remedied by using coefficient leakage. This "leaky" LMS algorithm can be written as

$$\underline{w}(n+1) = (1-\mu r)\underline{w}(n) + \mu e(n)\underline{u}(n)$$

where the adaptation constant μ and the leakage coefficient *r* are a small positive values.

6.1.3.2 The RLS Algorithm

The LMS algorithm has many advantages (due to its computational simplicity), but its convergence rate is slow. The LMS algorithm has only one adjustable parameter that affects convergence rate, the step-size parameter μ, which has a limited range of adjustment in order to insure stability.

6 Adaptive Filters

For faster rates of convergence, more complex algorithms with additional parameters must be used. The RLS algorithm uses a least-squares method to estimate correlation directly from the input data. The LMS algorithm uses the statistical mean-squared-error method, which is slower.

The RLS algorithm uses a transversal FIR filter implementation. The order of operations the algorithm takes is

1. Compute the filter output (tap weights initialized to zero)
2. Find the error signal
3. Compute the Kalman Gain Vector (defined below)
4. Update the inverse of the correlation matrix
5. Update the tap weights

The *Kalman Gain Vector* is based on input-data autocorrelation results, the input data itself, and a factor called the *forgetting factor*. The forgetting factor ranges between zero and one and provides a time-weighting of the input data such that the most recent data points are weighted more heavily than past data. This allows the filter coefficients to adapt to time-varying statistical characteristics of the input data.

The tap weight update is based on the error signal and the Kalman Gain Vector and is expressed as

$$w(n) = w(n-1) + Ke(n)$$

where K is the Kalman Gain Vector and e(n) represents the error signal.

6.2 IMPLEMENTATIONS

The adaptive filter routines have two inputs

- input sample
- desired response

and three outputs

- filter output
- filter error signal
- filter weights.

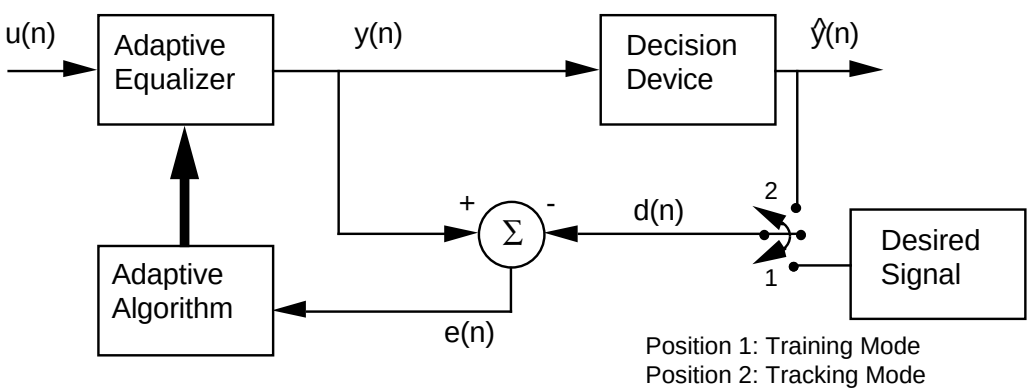

Figure 6.5 Generic Adaptive Filter

6.2.1 Transversal Filter Implementation

The transversal and symmetric transversal FIR filter structures require the use of a delay line of input samples *u(n).* The samples *u(n), u(n-1), ..., u(n-N+1)* are stored in a circular Data Memory buffer in a reverse order. The filter weights $w_0, w_1,...,w_{N-1}$ are placed in a circular Program Memory Data buffer in forward order. Note that *u(n + 1)* replaces *u(n– N + 1)* at time *n + 1*; therefore, the index of the delay line buffer must point to *u(n - N + 1)* before the next filter iteration.

6.2.2 LMS (Transversal FIR Filter Structure)

The routine accepts two inputs, the input sample, u(n), and the desired output, d(n).

The filter output, *y(n)*, is calculated in terms of the difference equation

$$y(n) = \sum_{i=0}^{N-1} w_i(n)\, u(n-i)$$

The calculation is made in the single-instruction `macs` loop, which stores the *u(n)* value in a delay line when input. Once the output *y(n)* is calculated, its value is subtracted from *d(n)* to yield the error signal, *e(n)*. The tap weights are updated according to the relationship

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * e(n) * u(n-i)$$

where $0 \le i \le N-1$.The `weights` buffer is then updated with the new value. `STEPSIZE` is a constant set to 0.005. Increasing this value decreases adaptation time, but has a negative effect on the system's steady-state mean-squared error.

6.2.2.1 Code Listing—lms.asm

```
/
*****************************************************************************

File Name
   LMS.ASM

Version
   April 2 1991

Purpose
   Performs LMS algorithm implemented with a transversal FIR filter structure.

Equations Implemented
   ********************************************************************
   * 1)  y(n)= w.u  ( . = dot_product), y(n)= FIR filter output      *
   * where  w= [w0(n) w1(n) ... wN-1(n)]= filter weights             *
   * and  u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
   * n= time index, N= number of filter weights (taps)              *
   * 2)  e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
   * 3)  wi(n+1)= wi(n)+STEPSIZE*e(n)*u(n-i),  0 =<i<= N-1          *
   ********************************************************************

Calling Parameters
   f0= u(n) = input sample
   f1= d(n) = desired output

Return Values
   f13= y(n)= filter output
   f6= e(n)= filter error signal
   i8 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f4, f6, f7, f8, f12, f13

Cycle Count
   lms_alg: 3N+8 per iteration, lms_init: 12+N

# PM Locations
   pm code= 29 words, pm data= N words

# DM Locations
   dm data= N words

*****************************************************************************/

#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   lms_init, lms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;
```

(listing continues on next page)

6 Adaptive Filters

```
.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
lms_init: b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f0=0.0;
          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f0, pm(i8,m8)=f0;
          rts;            /* clear delay line & weights */



lms_alg:  dm(i0,m0)=f0, f4=pm(i8,m8);
                                  /* store u(n) in delay line, f4=w0(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8); /*f8= u(n)*w0(n)
                                     f0= u(n-1), f4= w1(n) */
          f12=f0*f4, f0=dm(i0,m0), f4= pm(i8,m8);
                        /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n) */
          lcntr=TAPS-3, do macs until lce;
macs:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4= pm(i8,m8);
                                    /* f12= u(n-i)*wi(n), f8= sum of prod,
                                          f0= u(n-i-1), f4= wi+1(n)   */
          f12=f0*f4, f8=f8+f12;   /* f12= u(n-N+1)*wN-1(n) */
          f13=f8+f12;             /* f13= y(n) */
          f6=f1-f13;              /* f6= e(n) */
          f1=f6*f7, f4=dm(i0,m0); /* f1= STEPSIZE*e(n), f4= u(n) */
          f0=f1*f4, f12=pm(i8,m8);/* f0= STEPSIZE*e(n)*u(n),
                                     f12= w0(n) */

          lcntr=TAPS-1, do update_weights until lce;
            f8=f0+f12, f4=dm(i0,m0), f12=pm(i8,m8); /* f8= wi(n+1) */
                                   /* f4= u(n-i-1), f12= wi+1(n) */

update_weights:   f0=f1*f4, pm(i9,m8)=f8;     /* f0= STEPSIZE*e(n)*u(n-i-1) */
                                   /* store wi(n+1) */
          rts(db);
          f8=f0+f12, f0=dm(i0,1); /* f8= wN-1(n+1) */
                       /* i0 -> u(n+1) location in delay line */
          pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.1 lms.asm

6.2.3 llms.asm—Leaky LMS Algorithm (Transversal)

Implementation of this algorithm is similar to the standard LMS, except the tap weight update calculation takes into account the constant LEAK_COEF , such that

$$w_i(n+1) = \text{LEAK\_COEF} * w_i(n) + \text{STEPSIZE} * e(n) * u(n-i)$$

6.2.3.1 Code Listing

```
/
*****************************************************************************

File Name
    LLMS.ASM

Version
    April 2 1991

Purpose
    Performs the “leaky” LMS algorithm implemented with a
    transversal FIR filter structure


Equations Implemented
    ***************************************************************
    * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights           *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
    * n= time index, N= number of filter weights (taps)           *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
    * 3) wi(n+1)= LEAK_COEF*wi(n)+STEPSIZE*e(n)*u(n-i), 0 =<i<= N-1 *
    ***************************************************************

Calling Parameters
    f0= u(n)= input sample
    f1= d(n)= desired output

Return Values
    f13= y(n)= filter output
    f6= e(n)= filter error signal
    f8 -> Program Memory Buffer of the filter weights

Registers Affected
    f0, f1, f2, f4, f6, f7, f8, f9, f12, f13

Cycle Count
    llms_alg: 3N+8 per iteration, llms_init: 13+N

# PM Locations
    pm code= 30 words, pm data= N words

# DM Locations
    dm data= N words
```

(listing continues on next page)

6 Adaptive Filters

```
*****************************************************************************/

#define   TAPS         5
#define   STEPSIZE     0.0045
#define LEAK_COEF       0.9995

.GLOBAL   llms_init, llms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
llms_init:       b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f2=LEAK_COEF;
          f0=0.0;

          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f0, pm(i8,m8)=f0; /* clear delay line & weights */
          rts;

llms_alg: f9= pass f1, dm(i0,m0)=f0, f4=pm(i8,m8);
                   /* store u(n) in delay line, f4= w0(n), f9= d(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8); /* f8= u(n)*w0(n) */
                                        /* f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                          /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n) */

          lcntr=TAPS-3, do macs until lce;
macs:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                                 /* f12= u(n-i)*wi(n), f8= sum of prod,
                                    f0= u(n-i-1), f4= wi+1(n) */

          f12=f0*f4, f8=f8+f12;               /* f12=u(n-N+1)*wN-1(n) */
          f13=f8+f12, f4=pm(i8,m8);           /* f13= y(n), f4= w0(n) */
          f12=f2*f4, f6=f9-f13, f4=dm(i0,m0); /* f12= LEAK_COEF*w0(n)
                                                  f6= e(n), f4= u(n)*/
          f0=f6*f7;                           /* f0= STEPSIZE*e(n) */
          f9=f0*f4, f4=dm(i0,m0);             /* f9= STEPSIZE*e(n)*u(n),
                                                  f4= u(n-1) */
```

```
          lcntr=TAPS-1, do update_weights until lce;
f9=f0*f4, f8=f9+f12, f4=dm(i0,m0), f12=pm(i8,m8); /*
f9=STEPSIZE*e(n)*u(n-i-1), f8=wi(n+1),
          f4=u(n-i-2),f12=wi+1(n) */
update_weights:   f12=f2*f12, pm(i9,m8)=f8;
/* f12= LEAK_COEF*wi+1(n), store wi(n+1) */ rts(db);
          f8=f9+f12, f0=dm(i0,2);
/* f8= wN-1(n+1) i0 -> u(n+1) location in delay line */
          pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.2 llms.asm

6.2.4 Normalized LMS Algorithm (Transversal)

This implementation is similar to the standard LMS of `lms.asm`, but adds calculation for the normalized stepsize. The normalized stepsize is calculated using the constants `ALPHA` and `GAMMA` and the value *E(n)*, which represents the energy in the delay line. *E(n)*is calculated recursively in the outer `nlms_alg` loop, just prior to the main filter calculation loop, `macs`.

6.2.4.1 Code Listing—nlms.asm

```
/*************************************************************************

    File Name
       NLMS.ASM

    Version
       April 2 1991

    Purpose
       Performs the normalized LMS algorithm implemented
       with a transversal FIR filter structure.


    Equations Implemented
       ***************************************************************
       * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
       * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights            *
       * and u=[ u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
       * n= time index, N= number of filter weights (taps)            *
       * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
       * 3) wi(n+1)= wi(n)+normalized_stepsize*e(n)*u(n-i), 0 =<i<= N-1*
       * 4) normalized_stepsize= ALPHA/(GAMMA+E(n))                   *
       * where E(n)= u.u = energy in delay line                       *
       * E(n) is computed recursively as follows                      *
       * 5) E(n)= E(n-1)+u(n)**2-u(n-N)**2                            *
       ***************************************************************

    Calling Parameters
       f0= u(n) = input sample
       f1= d(n) = desired output

    Return Values
       f2= y(n)= filter output
       f6= e(n)= filter error signal
       i8 -> Program Memory Data buffer of the filter weights

    Registers Affected
       f0, f1, f2, f4, f5, f6, f7, f8, f9, f11, f12, f13, f14

    Cycle Count
       nlms_alg: 3N+16 per iteration, nlms_init: 14+N

    # PM Locations
       pm code= 32 words, pm data= N words

    # DM Locations
       dm data= N words

    *************************************************************************/
```

```
#define   TAPS         5
#define   ALPHA        0.1
#define   GAMMA        0.1

#include “a:\global\macros.h”

.GLOBAL   nlms_init, nlms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;



.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
nlms_init:       b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=weights;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=TAPS;
          f5=ALPHA;
          f11=GAMMA;      /* f11= E(0)= GAMMA */
          f9= 2.0;        /* f9= 2.0  for DIVIDE_ macro */
          f13=0.0;
          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f13, pm(i8,m8)=f13;
                                /* clear delay line & weights */
          rts;

nlms_alg: f14=f0*f0, dm(i0,m0)=f0, f4=pm(i8,m8);
                /* f14= u(n)**2, store u(n) in delay line, f4= w0(n) */
   f8=f0*f4, f11=f11+f14, f0=dm(i0,m0), f4=pm(i8,m8);
                /* f11= E(n-1)+u(n)**2, f8=u(n)*w0(n) */
          f12=f0*f4, f11=f11-f13, f0=dm(i0,m0), f4=pm(i8,m8);
                /* f12= u(n-1)*w1(n), f11= E(n), f0= u(n-2),
                    f4= w2(n) */

          lcntr=TAPS-3, do macs until lce;
macs:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1),
                         f4= wi+1(n) */
```

(listing continues on next page)

```
            f12=f0*f4, f8=f8+f12, f14=f11; /* f12= u(n-N+1)*wN-1(n),
                                              f14= E(n) */
            f2=f8+f12;                 /* f2= y(n) */
            f6=f1-f2, f4=dm(i0,m0);     /* f6= e(n), f4= u(n) */
            f7=f6*f5;                  /* f7= ALPHA*e(n) */
            DIVIDE(f1,f7,f14,f9,f0); /* f1= normalized_stepsize*e(n) */
            f0=f1*f4, f12=pm(i8,m8); /* f0= f1*u(n), f12= w0(n) */

            lcntr=TAPS-1, do update_weights until lce;
              f8=f0+f12, f4=dm(i0,m0), f12=pm(i8,m8); /* f8= wi(n+1)
                                                f4= u(n-i-1), f12= wi+1(n) */
update_weights:   f0=f1*f4, pm(i9,m8)=f8;
                                           /* f0=normalized_stepsize*e(n)*u(n-i-1)
                                                 store wi(n+1)*/
            rts(db);
            f8=f0+f12, f0=dm(i0,1); /* f8= wN-1(n+1)
                                    i0 -> u(n+1) location in delay line */
            f13=f4*f4, pm(i9,m8)=f8; /* f13= u(n-N)**2, store wN-1(n+1) */

.ENDSEG;
```

Listing 6.3 nlms.asm

6.2.5 Sign-Error LMS (Transversal)

The sign-error LMS, along with sign-data and sign-sign implementations, represent attempts to simplify the computational requirements of the LMS by reducing the number of multiplies required. While this approach has benefits for discrete IC or VLSI implementations, there is no computational benefit for programmable DSP processor implementations. The filter implementations are similar to the standard LMS, but the tap weight updates use information related to the sign of the error signal and/or the input data.

For the sign-error algorithm, taps are updated according to the relationship

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * sgn\{e(n)\} * u(n-i)$$

where

$sgn\{x\} = 1$ for $x \geq 0$
$sgn\{x\} = -1$ for $x < 0$

This function is implemented after the main filter calculation loop, `macs`, and prior to the `update_weights` loop, as the error signal is used across all weights *i*, for any given time, *n*.

6.2.5.1 Code Listing—selms.asm

```
/
*************************************************************************

File Name
     SELMS.ASM

Version
    April 2 1991

Purpose
    Performs the sign-error LMS algorithm implemented
    with a transversal FIR filter structure

Equations Implemented
    ********************************************************************
    * 1) y(n)= w.u  ( . = dot_product) , y(n)= FIR filter output       *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights               *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line   *
    * n= time index, N= number of filter weights (taps)               *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output    *
    * 3) wi(n+1)= wi(n)+STEPSIZE*sgn[e(n)]*u(n-i), 0 =<i<= N-1         *
    * where sgn[e(n)]= +1 if e(n) >= 0 and -1 if e(n) < 0              *
    ********************************************************************

Calling Parameters
    f0= u(n)= input sample
    f1= d(n)= desired output

Return Values
     f13= y(n)= filter output
     f1= e(n)= filter error signal
     i8 -> Program Memory Data buffer of the filter weights

Registers Affected
    f0, f1, f2, f4, f7, f8, f12, f13

Cycle Count
    selms_alg: 3N+8 per iteration, selms_init: 12+N

# PM Locations
    pm code: 29 words, pm data= N words

# DM Locations
    dm data= N words
```

(listing continues on next page)

6 Adaptive Filters

```
*****************************************************************************/

#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   selms_init;
.GLOBAL selms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
selms_init:      b0=deline_data;
          m0=-1;
          l0=TAPS;                /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;                /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f0=0.0;

          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:        dm(i0,m0)=f0, pm(i8,m8)=f0;
                                  /* clear delay line & weights */
          rts;


selms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
                                  /* store u(n) in delay line, f4= w0(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                  /* f8= u(n)*w0(n)f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                  /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n)
*/
```

```
           lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);

            /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4= wi+1(n) */
            f12=f0*f4, f8=f8+f12;   /* f12=u(n-N+1)*wN-1(n) */
            f13=f8+f12, f2=f7;      /* f13= y(n), f2= step-size */
            f1=f1-f13, f4=dm(i0,m0);/* f1= e(n), f4= u(n) */
            if lt f2=-f7;
                                    /* if e(n) < 0 then f2= -step-size */
           f0=f2*f4, f12=pm(i8,m8);/* f0= f2*u(n), f12= w0(n) */

           lcntr=TAPS-1, do update_weights until lce;
             f8=f0+f12, f4=dm(i0,m0), f12=pm(i8,m8);
                                   /* f8= wi(n+1), f4= u(n-i-1), f12= wi+1(n) */
update_weights:   f0=f2*f4, pm(i9,m8)=f8;
                                    /* store wi(n+1) */

            rts(db);
            f8=f0+f12, f0=dm(i0,1); /* f8= wN-1(n+1) */
                                    /* i0 -> u(n+1) location in delay line */
            pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.4 selms.asm

6.2.6 Sign-Data LMS (Transversal)

Another sign variation, this time relying on the sign of the data input, where

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * e(n) * sgn\{u(n - i)\}.$$

The sign function here is implemented in the `update_weights` loop, as

previous input values have influence across the taps, *i*, for a given time, *n*.

6.2.6.1 Code Listing—sdlms.asm

```
/
*************************************************************************

File Name
     SDLMS.ASM

Version
    April 2 1991

Purpose
    Performs the sign-data LMS algorithm implemented with
    a transversal FIR filter structure

Equations Implemented
    ***************************************************************
    * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights            *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
    * n= time index, N= number of filter weights (taps)            *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
    * 3) wi(n+1)= wi(n)+STEPSIZE*e(n)*sgn[u(n-i)], 0 =<i<= N-1      *
    * where sgn[u(n)]= +1 if u(n) >= 0 and -1 if u(n) < 0           *
    ***************************************************************

Calling Parameters
    f0= u(n)= input sample
    f1= d(n)= desired output

Return Values
    f13= y(n)= filter output
    f6= e(n)= filter error signal
    i8 -> Program Memory Data buffer of the filter weights

Registers Affected
    f0, f1, f4, f5, f6, f7, f8, f9, f12, f13

Cycle Count
    sdlms_alg: 4N+8 per iteration, sdlms_init: 13+N

# PM Locations
    pm code= 32 words, pm data= N words

# DM Locations
```

```
   dm data= N words
*******************************************************************************/


#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   sdlms_init, sdlms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
sdlms_init:      b0=deline_data;
          m0=-1;
          l0=TAPS;                  /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;                  /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f5=1.0;
          f0=0.0;
          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:        dm(i0,m0)=f0, pm(i8,m8)=f0;
                                    /* clear delay line & weights */
          rts;


sdlms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
                                    /* f4=w0(n),store u(n) in delay line */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                    /* f8= u(n)*w0(n) f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                   /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n)
*/
```

(listing continues on next page)

```
          lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
              /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4= wi+1(n)
*/

          f12=f0*f4, f8=f8+f12;     /* f12=u(n-N+1)*wN-1(n) */
          f13=f8+f12, f12=pm(i8,m8);/* f13= y(n), f12= w0(n) */
          f6=f1-f13, f8=dm(i0,m0);  /* f6= e(n), f8= u(n) */
          f1=f6*f7, f4=dm(i0,m0);   /* f1= STEPSIZE*e(n),f4=u(n-1) */
          f0=pass f8, f9=f1;        /* set ALU flags for sign of u(n) */

          lcntr=TAPS-1, do update_weights until lce;
            if lt f9=-f1;
                                     /* if u(n-i) < 0 then f9= -STEPSIZE*e(n) */
            f9=f1*f5, f8=f9+f12, f12=pm(i8,m8);
                                   /* f8= wi-1(n+1) f12= wi(n), f9= STEPSIZE*e(n)
*/
update_weights:   f0=pass f4, f4=dm(i0,m0), pm(i9,m8)=f8;
               /*store wi-1(n+1) set ALU flags for sign of u(n-i), f4= u(n-i-1)
*/

          if lt f9=-f1;
          rts(db);
          f8=f9+f12, f0=dm(i0,2);
```

```
                              /* f8= wN-1(n+1) i0 -> u(n+1) location in delay line
*/
          pm(i9,m8)=f8;             /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.5 sdlms.asm

6.2.7 Sign-Sign LMS (Transversal)

This sign variation uses sign information from both the error signal and input value to calculate the tap weight updates, using the relationship

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * sgn\{e(n)\} * sgn\{u(n-i)\}$$

The error value, *e(n)*, and the input value, *u(n)*, are multiplied together in the `update_weights` loop in order to set the appropriate multiplier sign flags. The resultant flags determine whether the `STEPSIZE` value is added or subtracted from the past tap weight.

6.2.7.1 Code Listing—sslms.asm

```
/
*************************************************************************

File Name
     SSLMS.ASM

Version
    April 2 1991

Purpose
    Performs the sign-sign LMS algorithm implemented with
    a transversal FIR filter structure

Equations Implemented
    ***************************************************************
    * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights            *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
    * n= time index, N= number of filter weights (taps)            *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
    * 3) wi(n+1)= wi(n)+STEPSIZE*sgn[e(n)]*sgn[u(n-i)], 0 =<i<= N-1 *
    * where sgn[x]= +1 if x >= 0 and -1 if x < 0                    *
    ***************************************************************

Calling Parameters
    f0= u(n)= input sample
```

(listing continues on next page)

```
   f1= d(n)= desired output

Return Values
   f13= y(n)= filter output
   f1= e(n)= filter error signal
   i8 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f2, f4, f7, f8, f9, f12, f13

Cycle Count
   sslms_alg: 4N+7 per iteration, sslms_init: 13+N

# PM Locations
   pm code= 31 words, pm data= N words

# DM Locations
   dm data= N words

*****************************************************************************/


#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   sslms_init, sslms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
sslms_init:      b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f2=1.0;
          f0=0.0;

          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f0, pm(i8,m8)=f0; /* clear delay line & weights */
          rts;


sslms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
```

```
                                    /* f4=w0(n), store u(n) in delay line */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                    /* f8= u(n)*w0(n), f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                   /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n)
*/

          lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
               /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4= wi+1(n)
*/

          f12=f0*f4, f8=f8+f12;     /* f12=u(n-N+1)*wN-1(n) */
          f13=f8+f12, f8=dm(i0,m0), f12=pm(i8,m8);
                                     /* f13= y(n), f12= w0(n) */
           f1=f1-f13, f9=f7;         /* f1= e(n), f8= u(n), f9= STEPSIZE */
          f0=f1*f8, f4=dm(i0,m0);
        /* f0= u(n)*e(n) to set multiplier flags for sign of product, f4=u(n-1)
*/

          lcntr=TAPS-1, do update_weights until lce;
            if ms f9=-f7;          /* if u(n)*e(n)<0 then f9=-STEPSIZE */
            f9=f2*f7, f8=f9+f12, f12=pm(i8,m8);
                            /* restore f9 to STEPSIZE f8= wi-1(n+1), f12= wi(n)
*/
update_weights:   f0=f1*f4, f4=dm(i0,m0), pm(i9,m8)=f8;
                         /* store wi-1(n+1),set multiplier flags, f4= u(n-i-1)
*/
          if ms f9=-f7;
          rts(db);
          f8=f9+f12, f0=dm(i0,2);
                         /* i0 -> u(n+1) location in delay line,.f8= wN-1(n+1)
*/
          pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.6 sslms.asm

6.2.8 Symmetric Transversal Filter Implementation LMS

In this routine, the filter output is defined as

$$N-1$$

$$y(n) = \sum_{i=0} w_i(n) * [u(n-i) + u(n-M+i+1)].$$

The filter calculation is broken up into two loops. The `macs` loop is similar to the one in `lms.asm`, but this algorithm adds a second calculation loop, `macs1`, to account for the second term in the sum-of-products. The `update_weights` loop has an added multiply to account for the extra term in the weight update equation

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * e(n) * [u(n-i) + u(n-M+i+1)]$$

6.2.8.1 Code Listing—sylms.asm

```
/
*******************************************************************************

File Name
   SYLMS.ASM

Version
   April 2 1991

Purpose
   Even order symmetric transversal filter structure implementation of the LMS
algorithm

Equations Implemented
   ***************************************************************
   * 1) y(n)= SUM[wi(n)*[u(n-i)+u(n-M+i+1)]] for 0 =<i<= N-1       *
   * where y(n)= FIR filter output, wi= filter weights             *
   *   u= input samples in delay line, N= number of weights (taps),*
   *   n= time index, and M= 2N= length of delay line              *
   * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
   * 3) wi(n+1)= wi(n)+STEPSIZE*e(n)*[u(n-i)+u(n-M+i+1)], 0=<i<=N-1*
    ***************************************************************

Calling Parameters
   Calling parameters (inputs):
   f0= u(n)= input sample
   f1= d(n)= desired output

Return Values
   f13= y(n)= filter output
   f6= e(n)= filter error signal
```

```
   i8 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f3, f4, f6, f7, f8, f9, f12, f13

Cycle Count
   sylms_alg: 2M+10 per iteration, sylms_init: 17+M

# PM Locations
   pm code= 39 words, pm data= N words

# DM Locations
   dm data= 2N

*************************************************************************/


#define   TAPS          4
#define   STEPSIZE     0.005

.GLOBAL   sylms_init, sylms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[2*TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
sylms_init:      b0=deline_data;
          b3=deline_data;
          m0=-1;
          m1=1;
          m2=TAPS+2;
          l0=2*TAPS;   /* circular delay line buffer of length M= 2N */
          l3=2*TAPS;
          b8=weights;
          b9=weights;
          m8=1;
          m9=-1;
          m10=-2;
          l8=TAPS;     /* circular weight buffer */
          l9=TAPS;
          f7=STEPSIZE;
          f0=0.0;
          lcntr=2*TAPS, do clear_bufs until lce;
clear_bufs:        dm(i0,m0)=f0, pm(i8,m8)=f0;
                        /* clear delay line & weights */
```

(listing continues on next page)

```
          rts;

sylms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
                        /* store u(n) in delay line, f4=w0(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                        /* f8= u(n)*w0(n), f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                      /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n) */

          lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
          /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4=wi+1(n) */

          f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f9=pm(i8,m10);
                       /* f12= u(n-N+1)*wN-1(n), i8 -> wN-2(n) */

          lcntr=TAPS-1, do macs1 until lce;
macs1:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m9);
                         /* f4=wi-1(n) */
          f12=f0*f4, f8=f8+f12, i3=i0;
          f13=f8+f12, f9=dm(i0,m0), f12=pm(i8,m8);
                        /* f13= y(n), i0 -> u(n-1), i8 -> w0(n) */
          f6=f1-f13, f4=dm(i3,m1);
                         /*f6= e(n), f4= u(n), i3 -> u(n-M+1)     */
          f1=f6*f7, f3=dm(i3,m1);  /* f1= STEPSIZE*e(n), f3= u(n-M+1) */
          f4=f3+f4, f3=dm(i3,m1), f12=pm(i8,m8);
                        /* f4= u(n)+u(n-M+1), f3= u(n-M+2), f12=w0(n) */
          f9=f1*f4, f0=dm(i0,m0);
                         /* f9= STEPSIZE*e(n)*[u(n)+u(n-M+1)] f0= u(n-1) */
          f4=f0+f3, f3=dm(i3,m1);
                        /* f4= u(n-1)+u(n-M+2), f3= u(n-M+3)*/
```

```
          lcntr=TAPS-1, do update_weights until lce;
            f9=f1*f4, f8=f9+f12, f0=dm(i0,m0), f12=pm(i8,m8)
                        /* f9= STEPSIZE*e(n)*[u(n-i-1)+u(n-M+2+i)], */
                        /* f8= wi(n+1), f0= u(n-2-i), f12= wi+1(n) */
update_weights:   f4=f0+f3, f3=dm(i3,m1), pm(i9,m8)=f8;
                       /* f4=u(n-2-i)+u(n-M+3+i), f3=u(n-M+4+i), store
wi(n+1) */

          rts(db);
          f8=f9+f12, f0=dm(i0,m2);
                       /* i0 ->u(n+1) location in delay line */
          pm(i9,m8)=f8;
.ENDSEG;
```

Listing 6.7 sylms.asm

6.2.9 Lattice Filter LMS With Joint Process Estimation

The LMS algorithm is implemented using a lattice structure with Joint Process estimation. The utility of a multistage lattice predictor is extended by using the resulting sequence of backward prediction errors, $b_m(n)$, as inputs to a corresponding set of tap coefficients $g_m(n)$, to produce the minimum mean-square estimate of some desired response $d(n)$. This is referred to as a Joint Process estimator since it provides simultaneous calculation of both forward prediction, $f_m(n)$, as well as backward prediction, providing the optimum estimation of the desired response. This joint process LMS technique allows very fast system adaptation for channel equalization and noise cancellation applications. This technique is also known as the gradient lattice-ladder algorithm.

The lattice parameters are described by the relationship

$$k_m(n+1) = K_m(n) + \mu[f_m(n)\, b_{m-1}(n-1) + b_m(n)\, f_{m-1}(n)]$$

where $0 < m \leq M$. The first-stage error is set as

$$e_0(n) = d(n) - b_0(n)\, g_0(n)$$

and subsequent stages set as

$$e_m(n) = e_m{-}1(n) - b_m(n)\, g_m(n)$$

where 0 < m < M. The tap coefficients are updated using the relationship

$$g_m(n+1) = g_m(n) + \mu e_m(n)\, b_m(n)$$

where $0 < m \leq M$.

The output value, $y(n)$, is calculated using the difference equation

$$y(n) = \sum^{M} g_m(n)\, b_m(n)$$

$m = 0$

The gradient lattice algorithm provides significantly faster convergence than the LMS algorithm. Unlike the LMS algorithm, the convergence rate of the gradient lattice algorithm does not depend on the eigenvalue spread of the autocorrelation matrix.

In addition to the filter weight, filter error, and filter output, this routine also provides reflection coefficient, forward prediction error, and backward prediction error as outputs. The lattice algorithm is implemented in the `update_coefs` loop. This loop is `STAGES` long, where `STAGES` is a constant set to the length of the lattice structure (`STAGES = 3` in this example).

6.2.9.1 Code Listing—latlms.asm

```
/
*******************************************************************************

File Name
   LATLMS.ASM

Version
   April 9 1991

Purpose
   Performs the LMS algorithm implemented with a lattice FIR filter structure
with Joint       Process estimation

Equations Implemented
   ****************************************************************
   * 1) f0(n)= b0(n)= u(n)                                        *
   * 2) fm(n)= fm-1(n) - km(n)*bm-1(n-1), 0 <m<= M                *
   * 3) bm(n)= bm-1(n-1) - km(n)*fm-1(n), 0 <m<= M                *
   * 4) km(n+1)= km(n) + STEPSIZE*[fm(n)*bm-1(n-1) + bm(n)*fm-1(n)],*
   *                                  0 <m<= M                    *
   * where u(n)= input sample, n= time index, M= Number of stages *
   *       fm(n)= forward prediction error of the mth stage       *
   *       bm(n)= backward prediction error of the mth stage      *
   *       km(n)= reflection coefficient of the mth stage         *
   *                                                              *
   * 5) e0(n)= d(n) - b0(n)*g0(n)                                 *
   * 6) em(n)= em-1(n) - bm(n)*gm(n), 0 <m<= M                    *
   * 7) gm(n+1)= gm(n) + STEPSIZE*em(n)*bm(n), 0 <=m<= M          *
   * 8) y(n)= SUM [gm(n)*bm(n)] for 0 <=m<= M                     *
   * where d(n)= desired output, y(n)= FIR filter output          *
   *       em(n)= output error at the mth stage                   *
```

(listing continues on next page)

6 Adaptive Filters

```
*       gm(n)= filter weight at the mth stage                    *
*****************************************************************


Calling Parameters
   f0= u(n)= input sample
   f9= d(n)= desired output

Return Values
   f0= bm(n)= mth backward prediction error
   f2= fm(n)= mth forward prediction error
   f6= em(n)= mth output error
   f12= y(n)= filter output
   i8 -> Program Memory Data buffer of the reflection coefficients
   i10 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f12, f13

Cycle Count
      LATLMS_ALG: 7+11M  per iteration    , LATLMS_INIT: 16+2M

# PM Locations
      pm code= 36 words, pm data= 2M+1

# DM Locations
   dm data= M+1

*******************************************************************************/

#define   STAGES       3
#define STEPSIZE 0.005

.GLOBAL   latlms_init, latlms_alg;

.SEGMENT/DM      dm_data;
.VAR      bpe_coef[STAGES+1];
.ENDSEG;

.SEGMENT/PM      pm_data;
.VAR      weights[STAGES+1];
.VAR      ref_coef[STAGES];
.ENDSEG;

.SEGMENT/PM      pm_code;
```

```
latlms_init:     b0=bpe_coef;
          m0=0;
          m1=1;
          l0=STAGES+1;
          b8=ref_coef;
          b10=weights;
          m8=0;
          m9=1;
          l8=STAGES;
          l10=STAGES+1;
          f7=STEPSIZE;
          f0=0.0;
          lcntr=STAGES, do clear_bufs until lce;
            dm(i0,m1)=f0, pm(i8,m9)=f0;
clear_bufs:        pm(i10,m9)=f0;
          rts(db);
          dm(i0,m1)=f0, pm(i10,m9)=f0;
          nop;

latlms_alg:      f10=pass f0, f1=dm(i0,m0), f4=pm(i10,m8);
                 /* f0= b0(n), f10= f0(n), f1= b0(n-1), f4= g0(n) */
          f12=f0*f4, dm(i0,m1)=f0, f5=pm(i8,m9);
                 /* f12= y0(n)= b0(n)*g0(n), store b0(n), f5= k1(n) */
          f13=f1*f5, f6=f9-f12;
                  /* f13= b0(n-1)*k1(n), f6= e0(n) */

          lcntr=STAGES, do update_coefs until lce;
            f3=f0*f7, f2=f10-f13, f8=f4;
                /* f2= fm(n), f3= bm-1(n)*stepsize, f8= gm-1(n) */
            f13=f3*f6, f3=f10;
                /* f13= stepsize*bm-1(n)*em-1(n), f3= fm-1(n) */
            f13=f3*f5, f4=f8+f13, f8=f5;
                /* f4= gm-1(n+1), f13= fm-1(n)*km(n), f8= km(n) */
            f0=f1-f13, pm(i10,m9)=f4;
                 /* f0= bm(n), store gm-1(n+1) */
            f10=f0*f3, f9=dm(i0,m0);
                 /* f10= bm(n)*fm-1(n), f9= bm(n-1) */
            f13=f1*f2, dm(i0,m1)=f0, f4=pm(i10,m8);
                /* f13= bm-1(n-1)*fm(n), store bm(n), f4= gm(n) */
            f13=f0*f4, f1=f10+f13, f10=f6;
                 /* f10= em-1(n)*/
                /* f13= bm(n)*gm(n), f1= fm(n)*bm-1(n-1)+bm(n)*fm-1(n) */
            f12=f12+f13, f5=pm(i8,-1);
```

```
                    /* f12= y(n) of mth stage, f5= km+1(n) */
                f13=f1*f7, f6=f10-f13, f1=f9;
                    /* f6= em(n), f13= stepsize*f1, f1= bm(n-1) */
                f13=f1*f5, f8=f8+f13, f10=f2;
                    /* f13= bm(n-1)*km+1(n), f8= km(n+1), f10= fm(n) */
update_coefs:     f3= f0*f6, pm(i8,2)=f8;
                   /* f3= bm(n)*em(n), store km(n+1) */

            rts(db), f3=f3*f7;
                   /* f3= stepsize*em(n)*bm(n) */
            f4=f3+f4, f3=pm(i8,-1);
                   /* f4= gm(n+1), i8 -> k1(n+1) */
            pm(i10,m9)=f4;
                   /* store gm(n+1) */
.ENDSEG;
```

Listing 6.8 latlms.asm

6.2.10 RLS (Transversal Filter)

The routine calculates the transversal filter output, *y(n)*, much like the lms.asm routine. The Kalman Gain Vector is computed in two steps. The first step computes an intermediate value, *x*. Since *x* deals with the *N*-by-*N* matrix of input autocorrelation data, it is calculated using a nested loop. The second step takes *x* and performs a dot product with the input vector, *u*, in the `mac3` loop, to create $k_0(n)$, the first element in the vector. The subsequent vector values, $k_i(n)$, along with the tap weight vector, $w_i(n)$, are computed in the loop `comp_kn_wn`. Finally, the autocorrelation matrix, *Z*, is updated in a nested loop, where the inner loop updates the rows, and the outer loop updates the columns.

During initialization, the *Z*-matrix is set up using the forgetting factor to create the time decay structure. The forgetting factor is set as a constant, `FORGET_FACT`.

6.2.10.1 Code Listing—rls.asm

```
/
*****************************************************************************

File Name
   RLS.ASM

Version
   April 18 1991

Purpose
   Performs the Recursive Least-Squares (RLS) algorithm
   implemented with a transversal FIR filter structure


Equations Implemented
   ********************************************************************
   * 1) x= s*Z(n-1).u  , where s= 1/forgetting factor, n= time index   *
   *        u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line *
   *        Z is an N-by-N matrix, N= number of filter weights         *
   * 2) k= x/[1+u.x] where k is an N-by-1 vector                       *
   * 3) Z(n)= s*Z(n-1) - k.xT , xT is the transpose of vector x        *
   * 4) e(n)= d(n) - w(N-1).u , e(n)= filter "a priori" error signal   *
   *        w= [w0 w1 ... wN-1]= filter weights, d(n)= desired output  *
   * 5) w(n)= w(n-1) + k*e(n)                                          *
   ********************************************************************

Calling Parameters
   f0, f1, f2, f3, f4, f5, f8, f9, f10, f11, f12, f13, f14

Return Values
```

(listing continues on next page)

6 Adaptive Filters

```
     f13= y(n)= filter output
     f1= e(n)= filter “a priori” error signal
     i0 -> Data Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f2, f4, f7, f8, f9, f12, f13

Cycle Count
   rls_alg: 3N**2+9N+20 per iteration, rls_init: N**2+3N+25

# PM Locations
   pm code= 81 words, pm data= 2N words

# DM Locations
   dm data= N**2+2N

*************************************************************************/

#define   TAPS         5
#define FORGET_FACT    0.9
#define INIT_FACT      1000.

#include “b:\global\macros.h”

.GLOBAL   rls_init, rls_alg;

.SEGMENT/DM      dm_data;
.VAR      weights[TAPS];
.VAR      zmatrix[TAPS*TAPS];
.VAR      xvector[TAPS];
.ENDSEG;

.SEGMENT/PM      pm_data;
.VAR      deline_data[TAPS];
.VAR      kvector[TAPS];
.ENDSEG;

.SEGMENT/PM      pm_code;
rls_init: b0=weights;
          l0=TAPS;
          b1=zmatrix;
          l1=TAPS*TAPS;
          b3=b1;
          l3=l1;
          b2=xvector;
          l2=TAPS;
          m0=1;
          m2=0;
          m3=-3;
          b8=deline_data;
          l8=TAPS;
          b9=kvector;
          l9=TAPS;
```

```
          m8=-1;
          m9=1;
          m11=3;
          f10=2.0;
          f14=1.0;
          f5=1/FORGET_FACT;
          f0=0.0;
          f1=INIT_FACT;
          lcntr=TAPS, do clear_bufs until lce;
            dm(i0,m0)=f0, pm(i8,m9)=f0;
clear_bufs:       dm(i2,m0)=f0, pm(i9,m9)=f0;
          dm(i1,m0)=f1;
          lcntr=TAPS-1, do init_zmatrix until lce;
            lcntr=TAPS, do clear_zel until lce;
clear_zel:         dm(i1,m0)=f0;
init_zmatrix:     dm(i1,m0)=f1;
          rts;

rls_alg:  f4=dm(i0,m0), pm(i8,m8)=f0;
                           /* f4=w0(n-1), store u(n) */
          f8=f0*f4, f4=dm(i0,m0), f0=pm(i8,m8);
                          /* f8=u(n)*w0(n-1), f4=w1(n-1), f0=u(n-1) */
          f12=f0*f4, f4=dm(i0,m0), f0=pm(i8,m8);
                         /* f12=u(n-1)*w1(n-1), f4=w2(n-1), f0=u(n-2) */

          lcntr=TAPS-3, do mac1 until lce;
mac1:              f12=f0*f4, f8=f8+f12, f4=dm(i0,m0), f0=pm(i8,m8);
               /* f12=u(n-i)*wi(n-1), f8=sum of prod,f4=wi+1(n-1), f0=u(n-i-1)
*/

          f12=f0*f4, f8=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
                /* f12=u(n-N+1)*wN-1(n-1), f4=Z0(0,n-1), f0=u(n) */
          f8=f0*f4, f13=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
                /* f13=y(n), f8= u(n)*Z0(0,n-1), f4=Z0(1,n-1), f0=u(n-1) */
          f12=f0*f4, f1=f9-f13, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f1=e(n), f12=u(n-1)*Z0(1,n-1), f4=Z0(2,n-1), f0=u(n-2) */

          lcntr=TAPS, do compute_xn until lce;
            lcntr=TAPS-3, do mac2 until lce;
mac2:                f12=f0*f4, f8=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
           /* f12=u(n-i)*Zk(i,n-1), f8=sum of prod,f4=Zk(i+1,n-1), f0=u(n-i-1)
*/

            f12=f0*f4, f8=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f12=u(n-N+1)*Zk(N-1,n-1), f4=Zk+1(0,n-1), f0=u(n) */
            f8=f0*f4, f2=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f2=xk(n),f8=u(n)*Zk+1(0,n-1),f4=Zk+1(1,n-1),
                                                  f0=u(n-1) */
            f12=f0*f4, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f12=u(n-1)*Zk+1(1,n-1), f4=Zk+1(2,n-1), f0=u(n-2) */
compute_xn:  dm(i2,m0)=f2;
               /* store xk(n) */

          f4=dm(i1,m3), f0=pm(i8,m11);
```

(listing continues on next page)

6 Adaptive Filters

```
                /* i1 -> Z0(0,n-1), i8 -> u(n) */
          f4=dm(i2,m0), f0=pm(i8,m8);
                /* f4= x0(n), f0= u(n) */
          f8=f0*f4, f4=dm(i2,m0), f0=pm(i8,m8);
               /* f8=x0(n)*u(n), f4=x1(n), f0=u(n-1) */
          f12=f0*f4, f8=f8+f14, f4=dm(i2,m0), f0=pm(i8,m8);
               /* f12=x1(n)*u(n-1), f8=1+x0(n)*u(n),f4=x2(n), f0=u(n-
2) */

          lcntr=TAPS-3, do mac3 until lce;
mac3:              f12=f0*f4, f8=f8+f12, f4=dm(i2,m0), f0=pm(i8,m8);
               /* f12=xi(n)*u(n-i), f8=1+sum of prod, f4=xi+1(n),
f0=u(n-i-1) */
          f12=f0*f4, f8=f8+f12, f4=f14;
                /* f12=u(n-N+1)*xN-1(n), f4=1.*/
          f12=f8+f12, f3=dm(i2,m0);
                /* f12=1+u.x, f3= x0(n) */
          DIVIDE(f2,f4,f12,f10,f0);
                /* f2=1/(1+u.x) */
          f0=f2*f3, modify(i8,m9);
                /* f0=k0(n), i8 -> u(n+1) location in delay line */

          lcntr=TAPS-1, do comp_kn_wn until lce;
            f8=f0*f1, f12=dm(i0,m2), pm(i9,m9)=f0;
               /* f8= ki(n)*e(n), f12=wi(n-1), store ki(n) */
            f8=f8+f12, f3=dm(i2,m0);
                /* f8=wi(n), f3=xi+1(n) */
comp_kn_wn:        f0=f2*f3, dm(i0,m0)=f8;
                /* f0=ki+1(n), store wi(n) */

          f8=f0*f1, f12=dm(i0,m2), pm(i9,m9)=f0;
               /* f8=kN-1(n)*e(n), f12=wN-1(n-1), store kN-1(n) */
          f11=f8+f12, f2=dm(i2,m0), f4=pm(i9,m9);
               /* f11= wN-1(n), f2= x0(n), f4= k0(n) */

          f12=f2*f4, f8=dm(i1,m0), f4=pm(i9,m9);
               /* f12= x0(n)*k0(n), f8=Z0(0,n-1), f4= k1(n) */
```

```
          lcntr=TAPS-1, do update_zn until lce;
            f12=f2*f4, f0=f8-f12, f8=dm(i1,m0);
            lcntr=TAPS-2, do update_zrow until lce;
              f3=f0*f5, f0=f8-f12, f8=dm(i1,m0), f4=pm(i9,m9);
update_zrow: f12=f2*f4, dm(i3,m0)=f3;
            f3=f0*f5, f0=f8-f12, f2=dm(i2,m0);
            f0=f0*f5, dm(i3,m0)=f3, f4=pm(i9,m9);
            f12=f2*f4, dm(i3,m0)=f0, f4=pm(i9,m9);
update_zn: f8=dm(i1,m0);
          f12=f2*f4, f0=f8-f12, f8=dm(i1,m0);
          lcntr=TAPS-2, do update_zlastrow until lce;
            f3=f0*f5, f0=f8-f12, f8=dm(i1,m0), f4=pm(i9,m9);
update_zlastrow:  f12=f2*f4, dm(i3,m0)=f3;
          f3=f0*f5, f0=f8-f12, dm(i0,m0)=f11;
          rts(db);
          f0=f0*f5, dm(i3,m0)=f3;
          dm(i3,m0)=f0;
.ENDSEG;
```

Listing 6.9 rls.asm

6.2.11 Testing Shell For Adaptive Filters

The program acts as the signal-generating plant, and can call any of the adaptive algorithms. This module must be edited for use with the specific routine it will call.

6.2.11.1 Code Listing—testafa.asm

```
/
*************************************************************************

File Name
   TESTAFA.ASM

Version
   April 2 1991

Purpose
   This is a testing shell for the Adaptive Filtering Algorithms.
```

(listing continues on next page)

6 Adaptive Filters

```
   This file has to be edited to conform with the algorithm employed.


Equations Implemented

************************************************************************
   *   This program generates a moving average sequence of real
samples, *
   * and employs the adaptive filter for System Identification based
on  *
   * a transversal FIR filter structure. The generating plant is
*
   * described by the following difference equation:                *
   *               y(n)= x(n-1)-0.5x(n-2)-x(n-3)
*
   * where the plant impulse response is 0, 1, -0.5, -1, 0, 0, ... .
*
   *   The filter is allowed five weight coefficients. The input data
*
   * sequence is a pseudonoise process with a period of 20.
*
   *   This program is also used to test adaptive filters based on
both  *
   * symmetric transversal FIR and lattice FIR structures. The output
*
   * filter error signal is stored in a data buffer named [flt_err]
*
   * for comparison.
*
   *   Place * s in this file with the corresponding code
*

************************************************************************



*************************************************************************/


#define   SAMPLES      **
   /* ** = 200 for RLS and 1000 for various LMS */

.EXTERN ***_init, ***_alg;
/* *** = lms, llms, nlms, selms, sdlms, sslms, */
                        /*       sylms, latlms, rls .
*/

.SEGMENT/DM     dm_data;
.VAR      flt_err[SAMPLES];
.VAR      input_data[20]= 0.038, -0.901, 0.01, -0.125, -1.275, 0.877,
```

```
-0.881,
                0.930, 1.233, -1.022, 1.522, -0.170, 1.489, -1.469,
                1.068, -0.258, 0.989, -2.891, -0.841, -0.355;
.GLOBAL   flt_err;
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      plant_hn[3]= -1.0, -0.5, 1.0;
.ENDSEG;

.SEGMENT/PM     rst_svc;
          dmwait=0x21;
          pmwait=0x21;
          jump begin;
.ENDSEG;

.SEGMENT/PM     pm_code;
begin:          b6=flt_err;
          l6=0;
          b7=input_data;
```

```
            l7=20;
            b15=plant_hn;
            l15=3;
            m15=1;
            m7=1;
            m6=-3;

            call ***_init;/* *** = algorithm codenames listed above */

            lcntr=SAMPLES, do adapt_filter until lce;
                           /* generate data through the plant */
              f0=dm(i7,m7), f4=pm(i15,m15); /* f0= x(n-3), f4= -1.0 */
              f8=f0*f4, f0=dm(i7,m7), f4=pm(i15,m15);
                            /* f8= -x(n-3), f0= x(n-2), f4= -0.5 */
              f12=f0*f4, f0=dm(i7,m7), f4=pm(i15,m15);
                            /* f12= -0.5x(n-2), f0= x(n-1), f4= 1.0 */
              f12=f0*f4, f8=f8+f12, f0=dm(i7,m7);
                           /* f12= x(n-1), f8= -x(n-3)-0.5x(n-2), f0=
    x(n)= u(n) */
              f*=f8+f12, modify(i7,m6);
                 /* f*= y(n)= d(n), i7 -> x(n-3) of next iteration */
                 /* f*= f1 for lms, nlms, llms, selms, sdlms,
    sslms,sylms */
                  /* f*= f9 for latlms and RLS */

              call ***_alg; /* *** = algorithm codenames listed above */

              dm(i6,m7)=f*; /* store filter error */
                 /* f*= f6 for lms, llms, nlms, sdlms, sylms, latlms */
                 /* f*= f1 for selms, sslms, rls */
              nop;
    adapt_filter:     nop;
              idle;
    .ENDSEG;
```

Listing 6.10 testafa.asm

6.3 CONCLUSION

Table 6.1 lists the number of instruction cycles and the memory usage relating to the various LMS algorithms implemented with a transversal FIR filter structure. It is interesting to note that the sign-LMS algorithms, originally designed to ease implementation in fixed-function silicon, require more instruction cycles in a programmable DSP due to their sign checking routines.

		Memory Usage		
Algorithm	**Cycles per Iteration**	**PM Code**	**PM Data**	**DM Data**
LMS	3N + 8	29	N	N
"Leaky" LMS	3N + 8	30	N	N
Normalized LMS	3N + 16	40	N	N
Sign-Error LMS	3N + 8	29	N	N
Sign-Data LMS	4N + 8	32	N	N
Sign-Sign LMS	4N + 8	31	N	N

Table 6.1 Transversal FIR LMS Performance & Memory Benchmarks For Filters Of Order N

Table 6.2 shows the performance of the LMS algorithm implemented with three different FIR filter structures. The final application will determine the structure used.

		Memory Usage		
Structure	**Cycles per Iteration**	**PM Code**	**PM Data**	**DM Data**
Transversal	3N + 8	29	N	N
Symmetric Transversal	2N + 10	39	0.5N	N
Lattice-Ladder	11N + 7	36	2N + 1	N + 1

Table 6.2 LMS Algorithm Benchmarks For Different Filter Structures

Finally, the performance of the RLS and LMS algorithms implemented with a transversal FIR filter structure are compared in Table 6.3. Evidently the fast convergence of the RLS occurs at the expense of instruction cycles.

Algorithm	**Cycles per Iteration**	**PM Code**	**PM Data**	**DM Data**
LMS	3N + 8	29	N	N
RLS	$3N^2 + 9N + 20$	89	2N	$N^2 + 2N$