DSP Applications Using C and the TMS320C6x DSK. Rulph Chassaing
Copyright © 2002 John Wiley & Sons, Inc.
ISBNs: 0-471-20754-3 (Hardback); 0-471-22112-0 (Electronic)

7 Adaptive Filters

- Adaptive structures
- The least mean squares (LMS) algorithm
- Programming examples for noise cancellation and system identification using C code

Adaptive filters are best used in cases where signal conditions or system parameters are slowly changing and the filter is to be adjusted to compensate for this change. The least mean squares (LMS) criterion is a search algorithm that can be used to provide the strategy for adjusting the filter coefficients. Programming examples are included to give a basic intuitive understanding of adaptive filters.

7.1 INTRODUCTION

In conventional FIR and IIR digital filters, it is assumed that the process parameters to determine the filter characteristics are known. They may vary with time, but the nature of the variation is assumed to be known. In many practical problems, there may be a large uncertainty in some parameters because of inadequate prior test data about the process. Some parameters might be expected to change with time, but the exact nature of the change is not predictable. In such cases it is highly desirable to design the filter to be self-learning, so that it can adapt itself to the situation at hand.

The coefficients of an adaptive filter are adjusted to compensate for changes in input signal, output signal, or system parameters. Instead of being rigid, an adaptive system can learn the signal characteristics and track slow changes. An adaptive filter can be very useful when there is uncertainty about the characteristics of a signal or when these characteristics change.

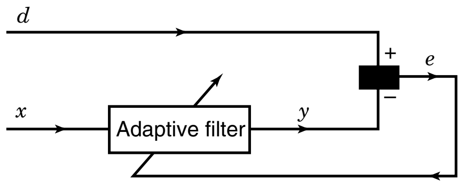

FIGURE 7.1. Basic adaptive filter structure.

Figure 7.1 shows a basic adaptive filter structure in which the adaptive filter's output y is compared with a desired signal d to yield an error signal e, which is fed back to the adaptive filter. The coefficients of the adaptive filter are adjusted, or optimized, using a least mean squares (LMS) algorithm based on the error signal.

We discuss here only the LMS searching algorithm with a linear combiner (FIR filter), although there are several strategies for performing adaptive filtering. The output of the adaptive filter in Figure 7.1 is

$$y(n) = \sum_{k=0}^{N-1} w_k(n)x(n-k) \tag{7.1}$$

where $w_k(n)$ represent N weights or coefficients for a specific time n. The convolution equation (7.1) was implemented in Chapter 4 in conjunction with FIR filtering. It is common practice to use the terminology of weights w for the coefficients associated with topics in adaptive filtering and neural networks.

A performance measure is needed to determine how good the filter is. This measure is based on the error signal,

$$e(n) = d(n) - y(n) \tag{7.2}$$

which is the difference between the desired signal $d(n)$ and the adaptive filter's output $y(n)$. The weights or coefficients $w_k(n)$ are adjusted such that a mean squared error function is minimized. This mean squared error function is $E[e^2(n)]$, where E represents the expected value. Since there are k weights or coefficients, a gradient of the mean squared error function is required. An estimate can be found instead using the gradient of $e^2(n)$, yielding

$$w_k(n+1) = w_k(n) + 2\beta e(n)x(n-k) \qquad k = 0, 1, \ldots, N-1 \tag{7.3}$$

which represents the LMS algorithm [1–3]. Equation (7.3) provides a simple but powerful and efficient means of updating the weights, or coefficients, without the need for averaging or differentiating, and will be used for implementing adaptive filters. The input to the adaptive filter is $x(n)$, and the rate of convergence and accuracy of the adaptation process (adaptive step size) is β.

For each specific time n, each coefficient, or weight, $w_k(n)$ is updated or replaced by a new coefficient, based on (7.3), unless the error signal $e(n)$ is zero. After the filter's output $y(n)$, the error signal $e(n)$ and each of the coefficients $w_k(n)$ are updated for a specific time n, a new sample is acquired (from an ADC) and the adaptation process is repeated for a different time. Note that from (7.3), the weights are not updated when $e(n)$ becomes zero.

The linear adaptive combiner is one of the most useful adaptive filter structures and is an adjustable FIR filter. Whereas the coefficients of the frequency-selective FIR filter discussed in Chapter 4 are fixed, the coefficients, or weights, of the adaptive FIR filter can be adjusted based on a changing environment such as an input signal. Adaptive IIR filters (not discussed here) can also be used. A major problem with an adaptive IIR filter is that its poles may be updated during the adaptation process to values outside the unit circle, making the filter unstable.

The programming examples developed later will make use of equations (7.1)–(7.3). In (7.3) we simply use the variable β in lieu of 2β.

7.2 ADAPTIVE STRUCTURES

A number of adaptive structures have been used for different applications in adaptive filtering.

1. *For noise cancellation.* Figure 7.2 shows the adaptive structure in Figure 7.1 modified for a noise cancellation application. The desired signal d is corrupted by uncorrelated additive noise n. The input to the adaptive filter is a noise n' that is correlated with the noise n. The noise n' could come from the same source as n but modified by the environment. The adaptive filter's output y is adapted to the noise n. When this happens, the error signal approaches the desired signal d. The overall output is this error signal and not the adaptive filter's output y. This structure will be further illustrated with programming examples using C code.
2. *For system identification.* Figure 7.3 shows an adaptive filter structure that can be used for system identification or modeling. The same input is to an unknown system in parallel with an adaptive filter. The error signal e is the difference between the response of the unknown system d and the response of the adaptive filter y. This error signal is fed back to the adaptive filter and

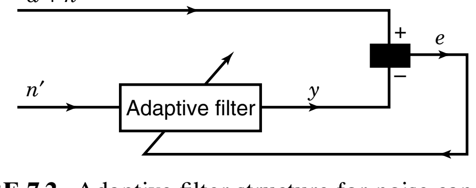

FIGURE 7.2. Adaptive filter structure for noise cancellation.

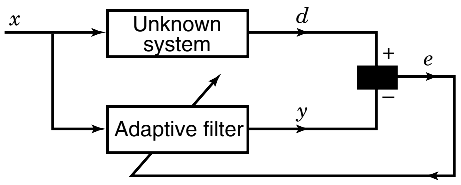

FIGURE 7.3. Adaptive filter structure for system identification.

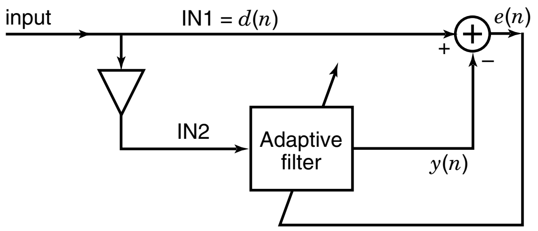

FIGURE 7.4. Adaptive predictor structure.

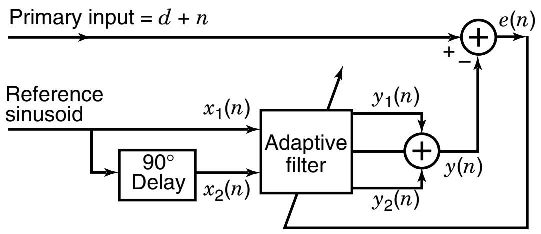

FIGURE 7.5. Adaptive notch structure with two weights.

is used to update the adaptive filter's coefficients until the overall output $y = d$. When this happens, the adaptation process is finished, and e approaches zero. In this scheme, the adaptive filter models the unknown system. This structure is illustrated later with three programming examples.

3. *Adaptive predictor.* Figure 7.4 shows an adaptive predictor structure which can provide an estimate of an input. This structure is illustrated later with a programming example.
4. Additional structures have been implemented, such as:
   (a) *Notch with two weights*, which can be used to notch or cancel/reduce a sinusoidal noise signal. This structure has only two weights or coefficients. This structure is shown in Figure 7.5 and is illustrated in Refs. 1, 3, and 4 using the C31 processor.

   (b) Adaptive channel equalization, used in a modem to reduce channel distortion resulting from the high speed of data transmission over telephone channels.

The LMS is well suited for a number of applications, including adaptive echo and noise cancellation, equalization, and prediction.

Other variants of the LMS algorithm have been employed, such as the sign-error LMS, the sign-data LMS, and the sign-sign LMS.

1. For the sign-error LMS algorithm, (7.3) becomes

$$w_k(n+1) = w_k(n) + \beta \operatorname{sgn}[e(n)]x(n-k) \tag{7.4}$$

   where sgn is the signum function,

$$\operatorname{sgn}(u) = \begin{cases} 1 & \text{if } u \geqslant 0 \\ -1 & \text{if } u < 0 \end{cases} \tag{7.5}$$

2. For the sign-data LMS algorithm, (7.3) becomes

$$w_k(n+1) = w_k(n) + \beta e(n) \operatorname{sgn}[x(n-k)] \tag{7.6}$$

3. For the sign-sign LMS algorithm, (7.3) becomes

$$w_k(n+1) = w_k(n) + \beta \operatorname{sgn}[e(n)] \operatorname{sgn}[x(n-k)] \tag{7.7}$$

   which reduces to

$$w_k(n+1) = \begin{cases} w_k(n) + \beta & \text{if } \operatorname{sgn}[e(n)] = \operatorname{sgn}[x(n-k)] \\ w_k(n) - \beta & \text{otherwise} \end{cases} \tag{7.8}$$

   which is more concise from a mathematical viewpoint because no multiplication operation is required for this algorithm.

The implementation of these variants does not exploit the pipeline features of the TMS320C6x processor. The execution speed on the TMS320C6x for these variants can be slower than for the basic LMS algorithm, due to additional decision-type instructions required for testing conditions involving the sign of the error signal or the data sample.

The LMS algorithm has been quite useful in adaptive equalizers, telephone cancelers, and so forth. Other methods, such as the recursive least squares (RLS) algorithm [4], can offer faster convergence than the basic LMS but at the expense of more computations. The RLS is based on starting with the optimal solution and then using each input sample to update the impulse response in order to maintain that optimality. The right step size and direction are defined over each time sample.

Adaptive algorithms for restoring signal properties can also be found in Ref. 4. Such algorithms become useful when an appropriate reference signal is not available. The filter is adapted in such a way as to restore some property of the signal lost before reaching the adaptive filter. Instead of the desired waveform as a template, as in the LMS or RLS algorithms, this property is used for the adaptation of the filter. When the desired signal is available, the conventional approach such as the LMS can be used; otherwise, a priori knowledge about the signal is used.

7.3 PROGRAMMING EXAMPLES FOR NOISE CANCELLATION AND SYSTEM IDENTIFICATION

The following programming examples illustrate adaptive filtering using the least mean squares (LMS) algorithm. It is instructive to read the first example even though it does not use the DSK, since it illustrates the steps in the adaptive process.

Example 7.1: Adaptive Filter Using C Code Compiled with Borland C/C++ (`Adaptc`)

This example applies the LMS algorithm using a C-coded program compiled with Borland C/C++. It illustrates the following steps for the adaptation process using the adaptive structure in Figure 7.1:

1. Obtain a new sample for each, the desired signal *d* and the reference input to the adaptive filter *x*, which represents a noise signal.
2. Calculate the adaptive FIR filter's output *y*, applying (7.1) as in Chapter 4 with an FIR filter. In the structure of Figure 7.1, the overall output is the same as the adaptive filter's output *y*.
3. Calculate the error signal applying (7.2).
4. Update/replace each coefficient or weight applying (7.3).
5. Update the input data samples for the next time *n*, with a data move scheme used in Chapter 4. Such a scheme moves the data instead of a pointer.
6. Repeat the entire adaptive process for the next output sample point.

Figure 7.6 shows a listing of the program `adaptc.c`, which implements the LMS algorithm for the adaptive filter structure in Figure 7.1. A desired signal is chosen as $2\cos(2n\pi f/F_s)$, and a reference noise input to the adaptive filter is chosen as $\sin(2n\pi f/F_s)$, where f is 1 kHz and $F_s = 8$ kHz. The adaptation rate, filter order, number of samples are 0.01, 22, and 40, respectively.

The overall output is the adaptive filter's output *y*, which adapts or converges to the desired cosine signal *d*.

The source file was compiled with Borland's C/C++ compiler. Execute this program. Figure 7.7 shows a plot of the adaptive filter's output (`y_out`) converging to the desired cosine signal. Change the adaptation or convergence rate β to 0.02 and verify a faster rate of adaptation.

```
//Adaptc.c Adaptation using LMS without TI’s compiler

#include <stdio.h>
#include <math.h>
#define beta 0.01                       //convergence rate
#define N 21                            //order of filter
#define NS 40                           //number of samples
#define Fs 8000                         //sampling frequency
#define pi 3.1415926
#define DESIRED 2*cos(2*pi*T*1000/Fs)   //desired signal
#define NOISE sin(2*pi*T*1000/Fs)       //noise signal

main()
{
 long I, T;
 double D, Y, E;
 double W[N+1] = {0.0};
 double X[N+1] = {0.0};
 FILE *desired, *Y_out, *error;
 desired = fopen (“DESIRED”, “w++”);   //file for desired samples
 Y_out = fopen (“Y_OUT”, “w++”);       //file for output samples
 error = fopen (“ERROR”, “w++”);       //file for error samples
 for (T = 0; T < NS; T++)              //start adaptive algorithm
  {
   X[0] = NOISE;                        //new noise sample
   D = DESIRED;                         //desired signal
   Y = 0;                               //filter’output set to zero
   for (I = 0; I <= N; I++)
    Y += (W[I] * X[I]);                 //calculate filter output
   E = D - Y                            //calculate error signal
   for (I = N; I >= 0; I--)
    {
     W[I] = W[I] + (beta*E*X[I]);       //update filter coefficients
     if (I != 0)
     X[I] = X[I-1];                     //update data sample
    }
   fprintf (desired, “\n%10g   %10f”, (float) T/Fs, D);
   fprintf (Y_out, “\n%10g   %10f”, (float) T/Fs, Y);
   fprintf (error, “\n%10g   %10f”, (float) T/Fs, E);
   }
   fclose (desired);
   fclose (Y_out);
   fclose (error);
}
```

FIGURE 7.6. Adaptive filter program compiled with Borland C/C++ (`adaptc.c`).

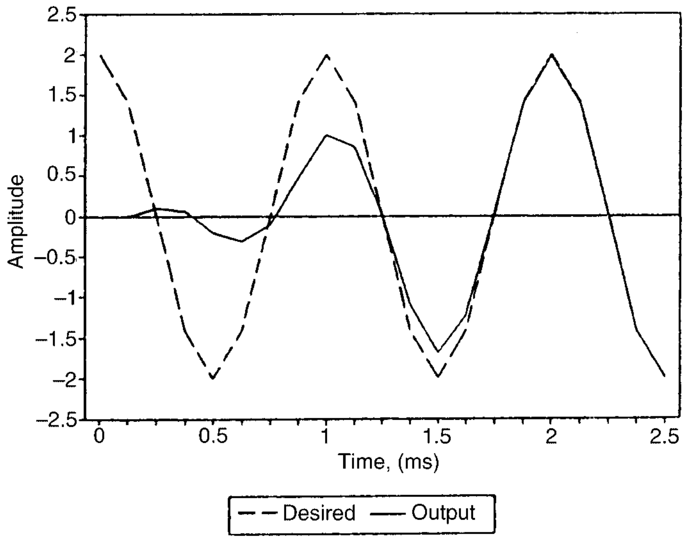

FIGURE 7.7. Plot of adaptive filter's output converging to cosine signal desired.

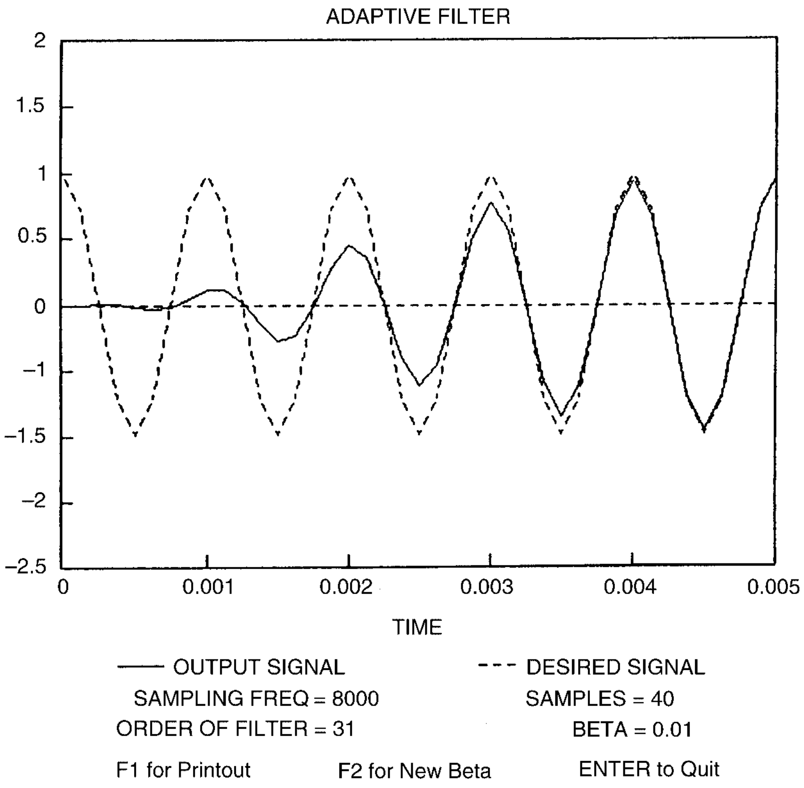

FIGURE 7.8. Plot of adaptive filter's output converging to cosine signal desired using interactive capability with progam `adaptive.c`.

Interactive Adaptation

A version of the program `adaptc.c` in Figure 7.6, with graphics and interactive capabilities to plot the adaptation process for different values of β is on the accompanying disk as `adaptive.c`, compiled with Turbo or Borland C/C++. It uses a desired cosine signal with an amplitude of 1 and a filter order of 31. Execute this program, enter a β value of 0.01, and verify the results in Figure 7.8. Note that the output converges to the desired cosine signal. Press `F2` to execute this program again with a different beta value.

Example 7.2: Adaptive Filter for Noise Cancellation (`adaptnoise`)

This example illustrates the application of the LMS criterion to cancel an undesirable sinusoidal noise. Figure 7.9 shows a listing of the program *`adaptnoise.c`*, which implements an adaptive FIR filter using the structure in Figure 7.1. This program uses a float data format. An integer format version is included on the accompanying disk as *`adaptnoise_int.c`*.

A desired sine wave of 1500 Hz with an additive (undesired) sine wave noise of 312 Hz forms one of two inputs to the adaptive filter structure. A reference (template) cosine signal, with a frequency of 312 Hz, is the input to a 30-coefficient adaptive FIR filter. The 312-Hz reference cosine signal is correlated with the 312-Hz additive sine noise but not with the 1500-Hz desired sine signal.

For each time n, the output of the adaptive FIR filter is calculated and the 30 weights or coefficients are updated along with the delay samples. The "error" signal E is the overall desired output of the adaptive structure. This error signal is the difference between the desired signal and additive noise (*`dplusn`*), and the adaptive filter's output, `y(n)`.

All signals used are from a lookup table generated with MATLAB. No external inputs are used in this example. Figure 7.10 shows a MATLAB program *`adaptnoise.m`* (a more complete version is on the disk) that calculates the data values for the desired sine signal of 1500 Hz, the additive noise as a sine of 312 Hz, and the reference signal as a cosine of 312 Hz. The appropriate files generated (on the disk) are:

1. *`dplusn`*: sine(1500 Hz) + sine(312 Hz)
2. *`refnoise`*: cosine(312 Hz)

Figure 7.11 shows the file *`sin1500.h`* with sine data values that represent the 1500-Hz sine-wave signal desired. The frequency generated associated with `sin1500.h` is

$$f = F_s\,(\#\text{ of cycles})/(\#\text{ of points}) = 8000(24)/128 = 1500\text{ Hz}$$

The constant *`beta`* determines the rate of convergence.

```
//Adaptnoise.c  Adaptive FIR filter for noise cancellation

#include <refnoise.h>                //cosine 312Hz
#include <dplusn.h>                  //sin(1500) + sin(312)
#define beta 1E-9                    //rate of convergence
#define N 30                         //# of weights (coefficients)
#define NS 128                       //# of output sample points
float w[N];                          //buffer weights of adapt filter
float delay[N];                      //input buffer to adapt filter
short output;                        //overall output
short out_type = 1;                  //output type for slider

interrupt void c_int11()             //ISR
{
 short i;
 static short buffercount=0;         //init count of # out samples
 float yn, E;                        //output filter/"error" signal

 delay[0] = refnoise[buffercount]; //cos(312Hz) input to adapt FIR
 yn = 0;                             //init output of adapt filter

 for (i = 0; i < N; i++)             //to calculate out of adapt FIR
    yn += (w[i] * delay[i]);         //output of adaptive filter

 E = dplusn[buffercount] - yn;       //"error" signal=(d+n)-yn

 for (i = N-1; i >= 0; i--)          //to update weights and delays
   {
    w[i] = w[i] + beta*E*delay[i];//update weights
    delay[i] = delay[i-1];           //update delay samples
   }
 buffercount++;                      //increment buffer count
 if (buffercount >= NS)              //if buffercount=# out samples
    buffercount = 0;                 //reinit count

 if (out_type == 1)                  //if slider in position 1
    output = ((short)E*10);          //"error" signal overall output
 else if (out_type == 2)
    output=dplusn[buffercount]*10; //desired(1500)+noise(312)

 output_sample(output);              //overall output result
 return;                             //return from ISR
}

void main()
{
 short T=0;
 for (T = 0; T < 30; T++)
   {
    w[T] = 0;                        //init buffer for weights
    delay[T] = 0;                    //init buffer for delay samples
   }
 comm_intr();                        //init DSK, codec, McBSP
 while(1);                           //infinite loop
}
```

FIGURE 7.9. Adaptive FIR filter program for noise cancellation (adaptnoise.c).

```
%Adaptnoise.m Generates: dplusn.h, refnoise.h, sin1500.h

for i=1:128
  desired(i) = round(100*sin(2*pi*(i-1)*1500/8000)); %sin(1500)
  addnoise(i) = round(100*sin(2*pi*(i-1)*312/8000)); %sin(312)
  refnoise(i) = round(100*cos(2*pi*(i-1)*312/8000)); %cos(312)
end

dplusn = addnoise + desired;                %sin(312) + sin(1500)

fid=fopen('sin1500.h','w');                 %desired sin(1500)
fprintf(fid,'short sin1500[128]={');
fprintf(fid,'%d, ' ,desired(1:127));
fprintf(fid,'%d' ,desired(128));
fprintf(fid,'};\n');
fclose(fid);

% fid=fopen('dplusn.h','w');                %desired + noise
% fid=fopen('refnoise.h','w');              %reference noise
```

FIGURE 7.10. MATLAB program to generate data values for `sine(1500)`, `sine(1500) + sine(312)`, and `cosine(312)` (`adaptnoise.m`).

```
short sin1500[128]={0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92};
```

FIGURE 7.11. MATLAB's header file generated for `sine(1500Hz)` with 128 points (`sin1500.h`).

Build and run this project as **`adaptnoise`**. Verify the following output result: The undesired 312-Hz sinusoidal signal is being gradually reduced (canceled), while the desired 1500-Hz signal remains. Note that in this application the output desired is the error signal *E*, which adapts (converges) to the desired signal. A faster rate of cancellation can be observed with a larger value of `beta`. However, if `beta` is too large, the adaptation process will not be observed since the output would be shown as the 1500-Hz signal. With the slider is position 2, the output is (*`dplusn`*), the desired 1500-Hz sinusoidal signal with the additive 312-Hz noise signal.

Example 7.3: Adaptive FIR Filter for System ID of Fixed FIR (`adaptIDFIR`)

Figure 7.12 shows a listing of the program *adaptIDFIR.c*, which models or identifies an unknown system. See also Example 7.2, which implements an adaptive FIR for noise cancellation.

To test the adaptive scheme, the unknown system to be identified is chosen as an FIR bandpass filter with 55 coefficients centered at $F_s/4 = 2$ kHz. The coefficients of this fixed FIR filter are in the file *bp55.cof*, introduced in Chapter 4. A 60-coefficient adaptive FIR filter models the fixed unknown FIR bandpass filter.

A pseudorandom noise sequence is generated within the program (see Examples 2.16 and 4.4) and becomes the input to both the fixed (unknown) and the adaptive FIR filters. This input signal represents a training signal. The adaptation process continues until the error signal is minimized. This feedback error signal is the difference between the output of the fixed unknown FIR filter and the output of the adaptive FIR filter.

An extra memory location is used in each of the two delay sample buffers (fixed and adaptive FIR). This is used to update the delay samples (see method B in Example 4.8).

Build and run this project as **adaptIDFIR** (using the C67x floating-point tools). Verify that the output (*adaptfir_out*) of the adaptive FIR filter is a bandpass filter centered at 2 kHz (with the slider in position 1 by default). With the slider in position 2, verify the output (*fir_out*) of the fixed FIR bandpass filter centered at 2 kHz and represented by the coefficient file *bp55.cof*. It can be observed that this output is practically identical to the adaptive filter's output.

Edit the main program to include the coefficient file *BS55.cof* (introduced in Example 4.4), which represents an FIR bandstop filter with 55 coefficients centered at 2 kHz. The FIR bandstop filter represents the unknown system to be identified.

Rebuild/run and verify that the output of the adaptive FIR filter (with the slider in position 1) is practically identical to the FIR bandstop filter (with the slider in position 2). Increase (decrease) *beta* by a factor of 10 to observe a faster (slower) rate of convergence. Change the number of weights (coefficients) from 60 to 40 and verify a slight degradation of the identification process.

Example 7.4: Adaptive FIR for System ID of Fixed FIR with Weights of Adaptive Filter Initialized as an FIR Bandpass (`adaptIDFIRw`)

The program *adaptIDFIR.c* in Example 7.3 is modified slightly to create the program *adaptIDFIRW.c* (on the accompanying disk). This new program initializes the weights of the adaptive FIR filter with the coefficients of an FIR bandpass filter centered at 3 kHz and represented by the coefficient file *bp3000.cof* (on the disk). The weights *w[i]* within the function *main* are initialized with the coefficients in the file *bp3000.cof* in lieu of zero.

```
//AdaptIDFIR.c Adaptive FIR for system ID of an FIR (uses C67 tools)

#include "bp55.cof"                    //fixed FIR filter coefficients
#include "noise_gen.h"                 //support noise generation file
#define beta 1E-13                     //rate of convergence
#define WLENGTH 60                     //# of coefffor adaptive FIR
float w[WLENGTH+1];                    //buffer coeff for adaptive FIR
int dly_adapt[WLENGTH+1];              //buffer samples of adaptive FIR
int dly_fix[N+1];                      //buffer samples of fixed FIR
short out_type = 1;                    //output for adaptive/fixed FIR
int fb;                                //feedback variable
shift_reg sreg;                        //shift register

int prand(void)                        //pseudo-random sequence {-1,1}
{
  int prnseq;
  if(sreg.bt.b0)
      prnseq = -8000;                  //scaled negative noise level
  else
      prnseq = 8000;                   //scaled positive noise level
  fb =(sreg.bt.b0)^(sreg.bt.b1);       //XOR bits 0,1
  fb^=(sreg.bt.b11)^(sreg.bt.b13);     //with bits 11,13 -> fb
  sreg.regval<<=1;
  sreg.bt.b0=fb;                       //close feedback path
  return prnseq;                       //return noise sequence
}

interrupt void c_int11()               //ISR
{
 int i;
 int fir_out = 0;                      //init output of fixed FIR
 int adaptfir_out = 0;                 //init output of adapt FIR
 float E;                              //error=diff of fixed/adapt out

 dly_fix[0] = prand();                 //input noise to fixed FIR
 dly_adapt[0]=dly_fix[0];              //as well as to adaptive FIR

 for (i = N-1; i>= 0; i--)
  {
   fir_out +=(h[i]*dly_fix[i]);        //fixed FIR filter output
   dly_fix[i+1] = dly_fix[i];          //update samples of fixed FIR
  }
```

FIGURE 7.12. Program to implement adaptive FIR filter that models (identifies) a fixed FIR filter (`adaptIDFIR.c`).

```
 for (i = 0; i < WLENGTH; i++)
   adaptfir_out +=(w[i]*dly_adapt[i]);    //adaptive FIR filter output

 E = fir_out - adaptfir_out;              //error signal

 for (i = WLENGTH-1; i >= 0; i--)
  {
   w[i] = w[i]+(beta*E*dly_adapt[i]);     //update weights of adaptive FIR
   dly_adapt[i+1] = dly_adapt[i];         //update samples of adaptive FIR
  }

 if (out_type == 1)                       //slider position for adapt FIR
   output_sample(adaptfir_out);           //output of adaptive FIR filter
 else if (out_type == 2)                  //slider position for fixed FIR
   output_sample(fir_out);                //output of fixed FIR filter
 return;
}

void main()
{
 int T=0, i=0;
 for (i = 0; i < WLENGTH; i++)
  {
   w[i] = 0.0;                            //init coeff for adaptive FIR
   dly_adapt[i] = 0;                      //init buffer for adaptive FIR
  }
 for (T = 0; T < N; T++)
   dly_fix[T] = 0;                        //init buffer for fixed FIR

 sreg.regval=0xFFFF;                      //initial seed value
 fb = 1;                                  //initial feevack value
 comm_intr();                             //init DSK, codec, McBSP
 while (1);                               //infinite loop
}
```

FIGURE 7.12. (*Continued*)

Build this project as **adaptIDFIRw** (using the C67x floating-point tools). Initially, the spectrum of the output of the adaptive FIR filter shows the FIR bandpass filter centered at 3kHz. Then, gradually, the output spectrum adapts (converges) to the fixed (unknown) FIR bandpass filter centered at 2kHz (represented by *bp55.cof*), while the reference filter gradually phases out. As the adaptation process takes place, one can observe at some time the two bandpass filters. You may wish to increase slightly the rate of adaptation (*beta*).

The adaptation process is illustrated with the CCS plots in Figure 7.13. Figure 7.14 illustrates the real-time adaptation process using an HP dynamic signal analyzer.

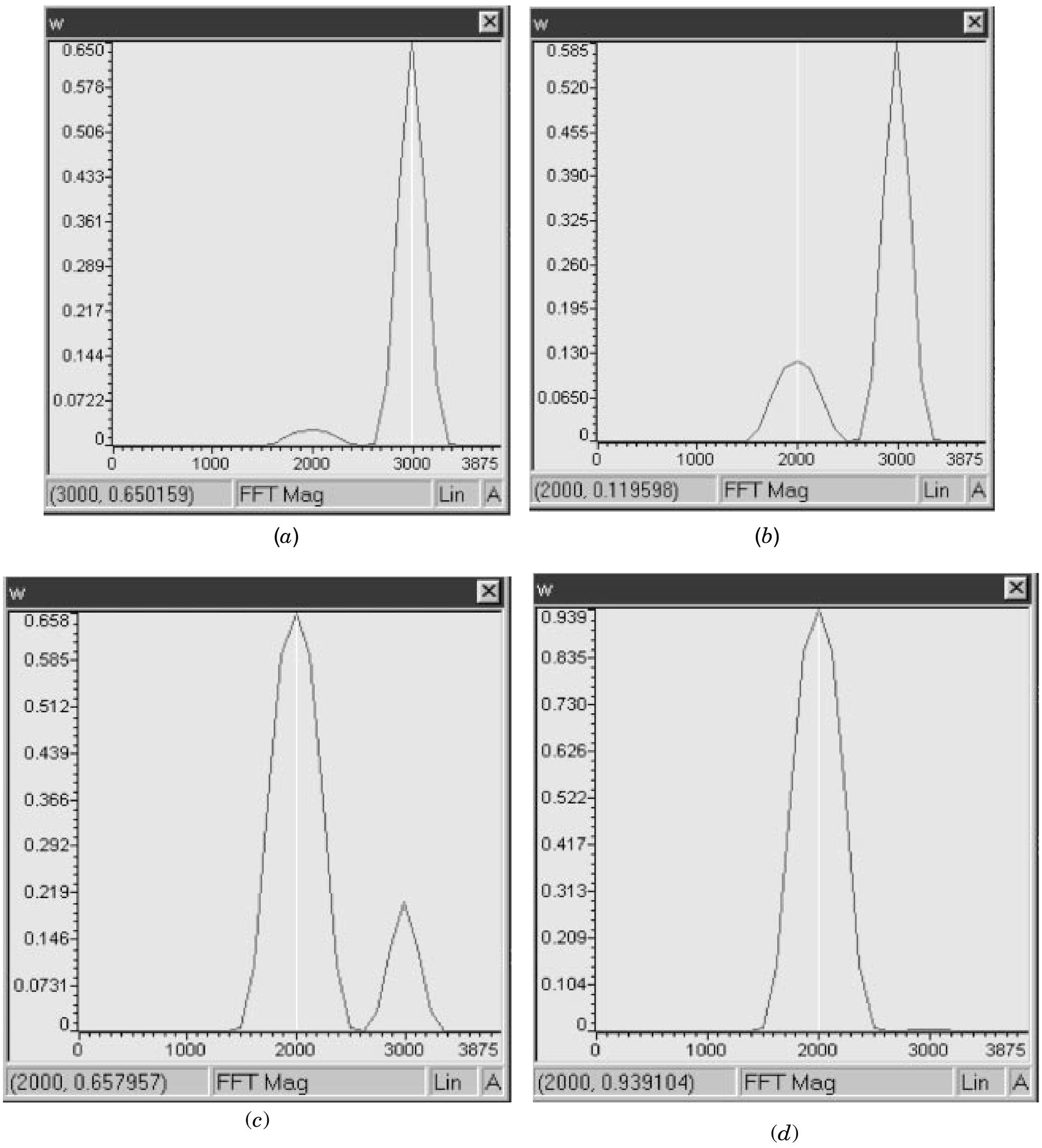

FIGURE 7.13. CCS plots to illustrate adaptation process of adaptive filter: (*a*) weights set initially as a 3-kHz bandpass filter; (*b*) weights starting to converge to a 2-kHz filter; (*c*) weights almost converged to 2 kHz with the 3-kHz filter reduced; (*d*) adaptation completed with convergence to the 2-kHz bandpass filter.

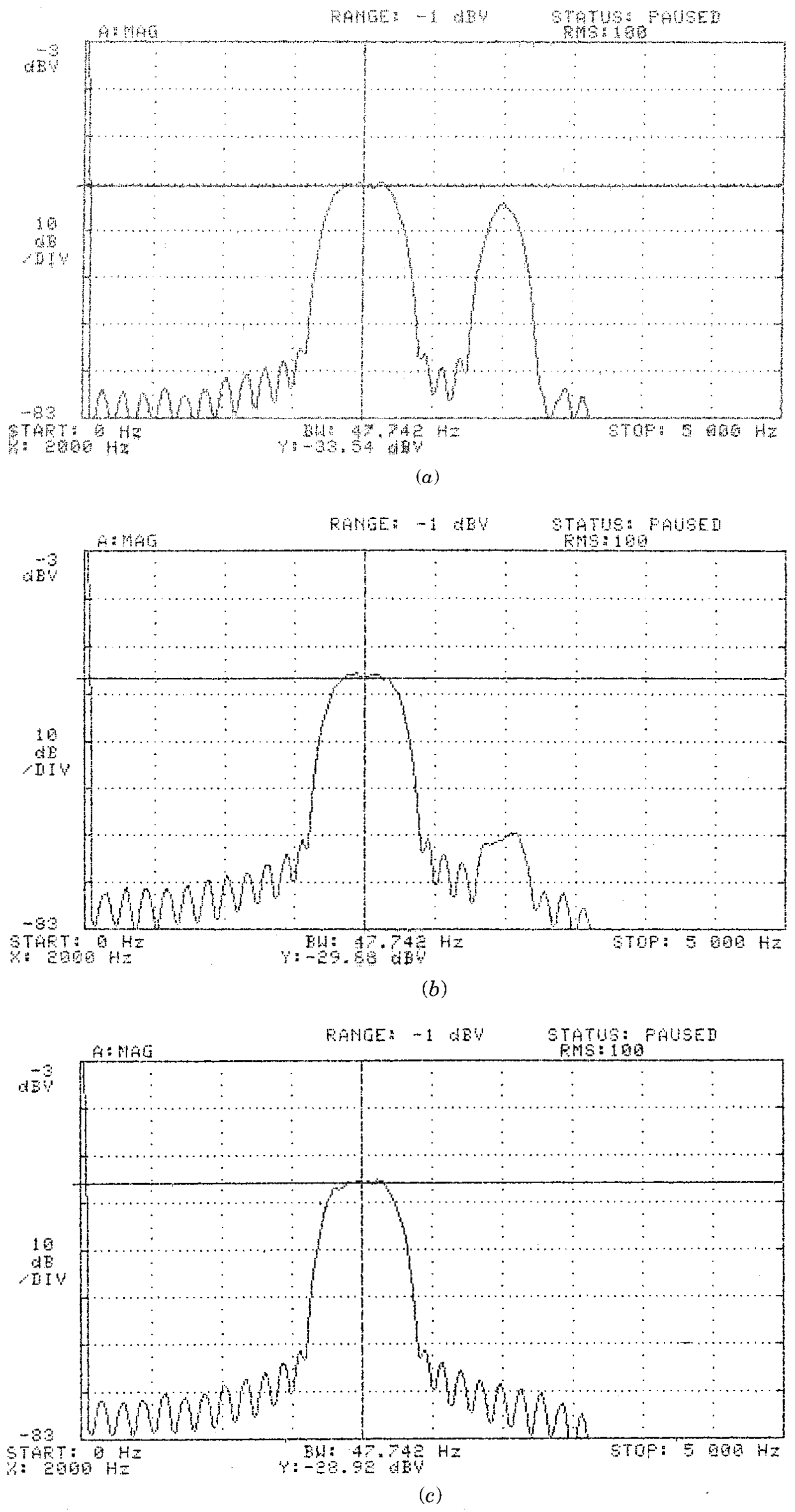

FIGURE 7.14. Real time adaptation process with adaptive filter converging to 2 kHz, obtained with an HP dynamic signal analyzer: (*a*) showing both the 3- and 2-kHz filters; (*b*) converging further to the 3-kHz filter; (*c*) adapted to the 3-kHz fixed filter.

Example 7.5: Adaptive FIR for System ID of Fixed IIR (`adaptIDIIR`)

Figure 7.15 shows a listing of the program *adaptIDIIR.c*, which uses an adaptive FIR filter to model or identify a system (fixed unknown IIR). See Example 5.1, which implements an IIR filter, and Examples 7.3 and 7.4, which implement an adaptive FIR filter to model a fixed FIR filter.

To test the adaptive scheme, the unknown system to be identified is chosen as a 36th-order IIR bandpass filter with *eighteen* second-order stages centered at 2 kHz. The coefficients of this fixed IIR filter are in the file *bp2000.cof*, introduced in Example 5.1. A 200-coefficient adaptive FIR filter is to model the fixed unknown IIR bandpass filter. A larger number of coefficients or weights than for the adaptive FIR filter are necessary for a good model of the IIR filter.

A pseudorandom noise sequence is generated (see Example 2.16) and becomes the input to both the fixed IIR filter and the adaptive FIR filter. The adaptation process continues until the error signal is minimized. This feedback error signal is the difference between the output of the fixed unknown IIR filter and the output of the adaptive FIR filter.

Build and run this project as **adaptIDIIR** (using the C67x floating-point tools). Verify that the output (*adaptfir_out*) converges to (models) the IIR bandpass filter centered at 2 kHz (with the slider initially in position 1). Verify that the output (*iir_out*) is the fixed IIR bandpass filter with the slider in position 2.

Include the coefficient file *lp2000.cof* in lieu of *bp2000.cof*. The coefficient file *lp2000.cof* represents an eighth-order (four second-order stages) IIR lowpass filter with a cutoff frequency of 2 kHz, introduced in Example 5.1. Verify that the adaptive FIR filter now adapts to the IIR lowpass filter with a cutoff frequency of 2 kHz.

Example 7.6: Adaptive Predictor for Cancellation of Narrowband Interference Added to Desired Wideband Signal (`adaptpredict`)

The program *adaptpredict.c*, shown in Figure 7.16, implements an adaptive FIR predictor for the cancellation of a narrowband interference in the presence of a wideband signal. The desired wideband signal with an additive narrowband interference is delayed and becomes the input to a 60-coefficient adaptive FIR filter.

The desired wideband signal is generated with a MATLAB program *wbsignal.m*, shown in Figure 7.17. This MATLAB program generates a 256-point lookup table in the file *wbsignal.h* (on the disk). A random sequence {-1,1} is generated, scaled, and written into the file *wbsignal.h*. Since the random sequence is for a length of 128 with a bit rate of 4 kHz, it is up-sampled to a 256-point sequence with a bit rate of 8 kHz. The wideband random sequence generated (with the file `wbsignal.h`) represents the signal desired.

The narrowband interference is an external signal. The bandwidth of the interference is narrow compared with the bandwidth of the random sequence generated

```
//AdaptIDIIR.c Adaptive FIR for system ID of fixed IIR using C67x tools

#include "bp2000.cof"                  //BP @ 2kHz fixed IIR coeff
#include "noise_gen.h"                 //support file noise sequence
#define beta 1E-11                     //rate of convergence
#define WLENGTH 200                    //# of coeff for adaptive FIR
float w[WLENGTH+1];                    //buffer coeff for adaptive FIR
int dly_adapt[WLENGTH+1];              //buffer samples of adaptive FIR
int dly_fix[stages][2] = {0};          //delay samples of fixed IIR
int a[stages][3], b[stages][2];        //coefficients of fixed IIR
short out_type = 1;                    //slider adaptive FIR/fixed IIR
int fb;                                //feedback variable for noise
shift_reg sreg;                        //shift register for noise

int prand(void)                        //pseudo-random sequence {-1,1}
{
  int prnseq;
  if(sreg.bt.b0)
      prnseq = -4000;                  //scaled negative noise level
  else
      prnseq= 4000;                    //scaled positive noise level
  fb =(sreg.bt.b0)^(sreg.bt.b1);       //XOR bits 0,1
  fb^=(sreg.bt.b11)^(sreg.bt.b13);     //with bits 11,13 ->fb
  sreg.regval<<=1;
  sreg.bt.b0=fb;                       //close feedback path
  return prnseq;                       //return noise sequence
}

interrupt void c_int11()               //ISR
{
 int i, un, input, yn;
 int iir_out=0;                        //init output of fixed IIR
 int adaptfir_out=0;                   //init output of adaptive FIR
 float E;                              //error signal

 dly_fix[0][0] = prand();              //input noise to fixed IIR
 dly_adapt[0] = dly_fix[0][0];         //same input to adaptive FIR
 input = prand();                      //noise as input to fixed IIR

 for (i = 0; i < stages; i++)          //repeat for each stage
  {
  un=input-((b[i][0]*dly_fix[i][0])>>15)-((b[i][1]*dly_fix[i][1])>>15);

  yn=((a[i][0]*un)>>15)+((a[i][1]*dly_fix[i][0])>>15)
     +((a[i][2]*dly_fix[i][1])>>15);
```

FIGURE 7.15. Program to implement adaptive FIR that models (identifies) a fixed IIR filter (adaptIDIIR.c).

```
  dly_fix[i][1] = dly_fix[i][0];          //update delays of fixed IIR
  dly_fix[i][0] = un;                     //update delays of fixed IIR
  input = yn;                             //in next stage=out previous
  }

 iir_out = yn;                            //output of fixed IIR

 for (i = 0; i < WLENGTH; i++)
   adaptfir_out +=(w[i]*dly_adapt[i]);    //output of adaptive FIR

 E = iir_out - adaptfir_out;              //error as difference of outputs
 for (i = WLENGTH; i > 0; i--)
  {
   w[i] = w[i]+(beta*E*dly_adapt[i]);     //update weights of adaptive FIR
   dly_adapt[i] = dly_adapt[i-1];         //update samples of adaptive FIR
  }

 if (out_type == 1)                       //slider adaptive FIR/fixed IIR
   output_sample(adaptfir_out);           //output of adaptive FIR
 else if (out_type == 2)
   output_sample(iir_out);                //output of fixed IIR
 return;                                  //return to main
}

void main()
{
 int i=0;
 for (i = 0; i < WLENGTH; i++)
  {
   w[i] = 0.0;                            //init coeff of adaptive FIR
   dly_adapt[i] = 0.0;                    //init samples of adaptive FIR
  }
 sreg.regval=0xFFFF;                      //initial seed value
 fb = 1;                                  //initial feedback value
 comm_intr();                             //init DSK, codec, McBSP
 while (1);                               //infinite loop
}
```

FIGURE 7.15. (*Continued*)

(the wideband signal desired). As a result, the samples of the interference are highly correlated. On the other hand, the samples of the wideband signal are relatively uncorrelated.

The characteristics of the narrowband interference permits the estimation of the narrowband interference from past samples of `splusn` in the program. The signal `splusn`, which represents the desired wideband signal with an additive narrowband

```
//Adaptpredict.C Adaptive predictor to cancel interference

#include “wbsignal.h”               //wide-band signal table look-up
#define beta 1E-14                  //rate of convergence
#define N 60                        //# of coefficients of adapt FIR
const short bufferlength = NS;      //buffer length for wideband signal
short splusn[N+1];                  //buffer wideband signal+interference
float w[N+1];                       //buffer for weights of adapt FIR
float delay[N+1];                   //buffer for input to adapt FIR

interrupt void c_int11()            //ISR
{
 static short buffercount=0;        //init buffer
 int i;
 float yn, E;                       //yn=out adapt FIR, error signal
 short wb_signal;                   //wideband desired signal
 short noise;                       //external interference

 wb_signal=wbsignal[buffercount];   //wideband signal from look-up table
 noise = input_sample();            //external input as interference
 splusn[0] = wb_signal + noise;     //wideband signal+interference
 delay[0] = splusn[3];              //delayed input to adaptive FIR
 yn = 0;                            //init output of adaptive FIR

 for (i = 0; i < N; i++)
   yn += (w[i] * delay[i]);         //output of adaptive FIR filter
 E = splusn[0] - yn;                //(wideband+noise)-out adapt FIR

 for (i = N-1; i >= 0; i--)
  {
   w[i] = w[i]+(beta*E*delay[i]);   //update weights of adapt FIR
   delay[i+1] = delay[i];           //update buffer delay samples
   splusn[i+1] = splusn[i];         //update buffer corrupted wideband
  }

 buffercount++;                     //incr buffer count of wideband
 if (buffercount >= bufferlength)   //if buffer count=length of buffer
   buffercount = 0;                 //reinit count
 output_sample((short)E);           //overall output
 return;
}

void main()
{
 int T = 0;
 for (T = 0; T < N; T++)            //init variables
  {
   w[T] = 0.0;                      //buffer for weights of adaptive FIR
   delay[T] = 0.0;                  //buffer for delay samples
   splusn[T] = 0;                   //buffer for wideband+interference
  }
 comm_intr();                       //init DSK, codec, McBSP
 while(1);                          //infinite loop
}
```

FIGURE 7.16. Adaptive predictor program for cancellation of narrowband interference in the presence of a wideband signal (adaptpredict.c).

```
%wbsignal.m Generates wideband random sequence. Represents one info bit

len_code = 128;                         %length of random sequence
code = 2*round(rand(1,len_code))-1;     %generates random sequence {1,-1}
sample_rate = 2;                        %up-sampling from 4 to 8kHz
NS = len_code * sample_rate;            %length of up-sampled sequence
sig = zeros(1,NS);                      %initialize random sequence
for i = 1:len_code                      %obtain up-sampled random sequence
  sig((i-1)*sample_rate + 1:i*sample_rate) = code(i);
end;
wbsignal = sig*5000;                    %scale for p-p amplitude of 500mV

fid=fopen('wbsignal.h','w');            %open file for wideband signal
fprintf(fid,'#define NS 256 //number of output sample points\n\n');
fprintf(fid,'short wbsignal[256]={');
fprintf(fid,'%d, ' ,wbsignal(1:NS-1));
fprintf(fid,'%d' ,wbsignal(NS));
fprintf(fid,'};\n\n');
fclose(fid);
return;
```

FIGURE 7.17. MATLAB program to generate a desired wideband random sequence (wbsignal.m).

interference, is delayed before becoming the input to the adaptive FIR filter. The delay is sufficiently long so that the delayed wideband signal is uncorrelated with the undelayed sample.

The output of the adaptive FIR filter is an estimate of the correlated narrowband interference. As a result, the error signal E is an estimate of the wideband signal desired.

Build and run this project as **adaptpredict** (using the C67x floating-point tools). Apply a sinusoidal input signal between 1 and 3kHz, representing the narrowband interference. Run the program and verify that the output spectrum of the error signal E adapts (converges) to the desired wideband signal, showing the input interference being gradually reduced.

Change the frequency of the input sinusoidal external interference and observe the adaptation process repeated to cancel the undesirable external interference. A faster rate of convergence can be observed by increasing `beta` by 10.

The wideband signal desired can be observed by outputting *`wb_signal`* (in lieu of E). Furthermore, the wideband signal with additive interference can be observed using `output_sample(`*`splusn[0]`*`)`. Better results are obtained when the amplitude of the external sinusoidal interference is about three times the amplitude of the wideband signal desired.

REFERENCES

1. B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1985.
2. B. Widrow and M. E. Hoff, Jr., Adaptive switching circuits, *IRE WESCON*, 1960, pp. 96–104.
3. B. Widrow, J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, R. H. Hearn, J. R. Zeidler, E. Dong, Jr., and R. C. Goodlin, Adaptive noise cancelling: principles and applications, *Proceedings of the IEEE*, Vol. 63, 1975, pp. 1692–1716.
4. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
5. D. G. Manolakis, V. K. Ingle, and S. M. Kogon, *Statistical and Adaptive Signal Processing*, McGraw-Hill, New York, 2000.
6. S. Haykin, *Adaptive Filter Theory*, Prentice Hall, Upper Saddle River, NJ, 1986.
7. J. R. Treichler, C. R. Johnson, Jr., and M. G. Larimore, *Theory and Design of Adaptive Filters*, Wiley, New York, 1987.
8. S. M. Kuo and D. R. Morgan, *Active Noise Control Systems*, Wiley, New York, 1996.
9. K. Astrom and B. Wittenmark, *Adaptive Control*, Addison-Wesley, Reading, MA, 1995.
10. J. Tang, R. Chassaing, and W. J. Gomes III, Real-time adaptive PID controller using the TMS320C31 DSK, *Proceedings of the 2000 Texas Instruments DSPS Fest Conference*, 2000.
11. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
12. R. Chassaing et al., Student projects on applications in digital signal processing with C and the TMS320C30, *Proceedings of the 2nd Annual TMS320 Educators Conference*, Texas Instruments, Dallas, TX, 1992.
13. C. S. Linquist, *Adaptive and Digital Signal Processing*, Steward and Sons, 1989.
14. S. D. Stearns and D. R. Hush, *Digital Signal Analysis*, Prentice Hall, Upper Saddle River, NJ, 1990.
15. J. R. Zeidler, Performance analysis of LMS adaptive prediction filters, *Proceedings of the IEEE*, Vol. 78, 1990, pp. 1781–1806.
16. S. T. Alexander, *Adaptive Signal Processing: Theory and Applications*, Springer-Verlag, New York, 1986.
17. C. F. Cowan and P. F. Grant, eds., *Adaptive Filters*, Prentice Hall, Upper Saddle River, NJ, 1985.
18. M. L. Honig and D. G. Messerschmitt, *Adaptive Filters: Structures, Algorithms and Applications*, Kluwer Academic, Norwell, MA, 1984.
19. V. Solo and X. Kong, *Adaptive Signal Processing Algorithms: Stability and Performance*, Prentice Hall, Upper Saddle River, NJ, 1995.
20. S. Kuo, G. Ranganathan, P. Gupta, and C. Chen, Design and implementation of adaptive filters, *IEEE 1988 International Conference on Circuits and Systems*, June 1988.

21. M. G. Bellanger, *Adaptive Digital Filters and Signal Analysis*, Marcel Dekker, New York, 1987.

22. R. Chassaing and B. Bitler, Adaptive filtering with C and the TMS320C30 digital signal processor, *Proceedings of the 1992 ASEE Annual Conference*, June 1992.

23. R. Chassaing, D. W. Horning, and P. Martin, Adaptive filtering with the TMS320C25, *Proceedings of the 1989 ASEE Annual Conference*, June 1989.