CHAPTER 5

REAL-TIME DSP APPLICATIONS

This chapter combines the DSP principles described in the previous chapters with the specifications of real-time systems designed to solve real-world problems and provide complete software solutions for several DSP applications. Applications of FFT spectrum analysis are described in section 5.1. Speech and music processing are considered in sections 5.3 and 5.4. Adaptive signal processing methods are illustrated in section 5.2 (parametric signal modeling) and section 5.5 (adaptive frequency tracking).

5.1 FFT POWER SPECTRUM ESTIMATION

Signals found in most practical DSP systems do not have a constant power spectrum. The spectrum of radar signals, communication signals, and voice waveforms change continually with time. This means that the FFT of a single set of samples is of very limited use. More often a series of spectra are required at time intervals determined by the type of signal and information to be extracted.

Power spectral estimation using FFTs provides these power spectrum snapshots (called *periodograms*). The average of a series of periodograms of the signal is used as the estimate of the spectrum of the signal at a particular time. The parameters of the *average periodogram spectral estimate* are:

(1) Sample rate: Determines maximum frequency to be estimated
(2) Length of FFT: Determines the resolution (smallest frequency difference detectable)
(3) Window: Determines the amount of spectral leakage and affects resolution and noise floor

(4) Amount of overlap between successive spectra: Determines accuracy of the estimate, directly affects computation time
(5) Number of spectra averaged: Determines maximum rate of change of the detectable spectra and directly affects the noise floor of the estimate

5.1.1 Speech Spectrum Analysis

One of the common application areas for power spectral estimation is speech processing. The power spectra of a voice signal give essential clues to the sound being made by the speaker. Almost all the information in voice signals is contained in frequencies below 3,500 Hz. A common voice sampling frequency that gives some margin above the Nyquist rate is 8,000 Hz. The spectrum of a typical voice signal changes significantly every 10 msec or 80 samples at 8,000 Hz. As a result, popular FFT sizes for speech processing are 64 and 128 points.

Included on the MS-DOS disk with this book is a file called CHKL.TXT. This is the recorded voice of the author saying the words "chicken little." These sounds were chosen because of the range of interesting spectra that they produced. By looking at a plot of the CHKL.TXT samples (see Figure 5.1) the break between words can be seen and the

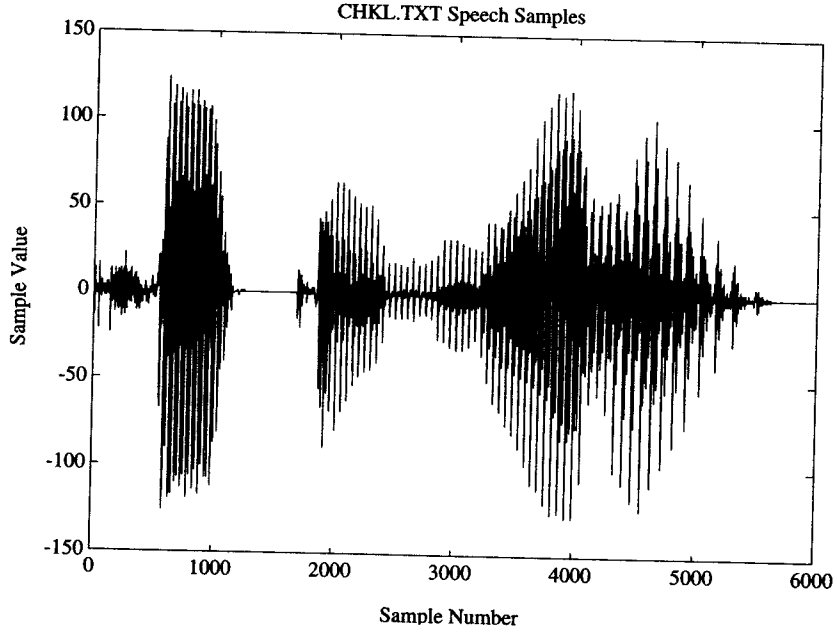

FIGURE 5.1 Original CHKL.TXT data file consisting of the author's words "chicken little" sampled at 8 kHz (6000 samples are shown).

relative volume can be inferred from the envelope of the waveform. The frequency content is more difficult to determine from this plot.

The program RTPSE (see Listing 5.1) accepts continuous input samples (using **getinput()**) and generates a continuous set of spectral estimates. The power spectral estimation parameters, such as FFT length, overlap, and number of spectra averaged, are set by the program to default values. The amount of overlap and averaging can be changed in real-time. RTPSE produces an output consisting of a spectral estimate every 4 input samples. Each power spectral estimate is the average spectrum of the input file over the past 128 samples (16 FFT outputs are averaged together).

Figure 5.2 shows a contour plot of the resulting spectra plotted as a frequency versus time plot with the amplitude of the spectrum indicated by the contours. The high fre-

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "rtdspc.h"

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

RTPSE.C - Real-Time Power spectral estimation using the FFT

This program does power spectral estimation on input samples.
The average power spectrum in each block is determined
and used to generate a series of outputs.

Length of each FFT snapshot: 64 points
Number of FFTs to average: 16 FFTs
Amount of overlap between each FFT: 60 points

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

/* FFT length must be a power of 2 */
#define FFT_LENGTH 64
#define M 6             /* must be log2(FFT_LENGTH) */

/* these variables global so they can be changed in real-time */
  int numav = 16;
  int ovlap = 60;

main()
{
  int           i,j,k;
  float         scale,tempflt;
```

LISTING 5.1 Program RTPSE to perform real-time power spectral estimation using the FFT. (*Continued*)


```
  static float   mag[FFT_LENGTH], sig[FFT_LENGTH], hamw[FFT_LENGTH];
  static COMPLEX samp[FFT_LENGTH];

/* overall scale factor */
  scale = 1.0f/(float)FFT_LENGTH;
  scale *= scale/(float)numav;

/* calculate hamming window */
  tempflt = 8.0*atan(1.0)/(FFT_LENGTH-1);
  for(i = 0 ; i < FFT_LENGTH ; i++)
    hamw[i] = 0.54 - 0.46*cos(tempflt*i);

/* read in the first FFT_LENGTH samples, overlapped samples read in loop */
  for(i = 0 ; i < FFT_LENGTH ; i++) sig[i] = getinput();

  for(;;) {

    for (k=0; k<FFT_LENGTH; k++) mag[k] = 0;

    for (j=0; j<numav; j++){

      for (k=0; k<FFT_LENGTH; k++){
        samp[k].real = hamw[k]*sig[k];
        samp[k].imag = 0;
      }

      fft(samp,M);

      for (k=0; k<FFT_LENGTH; k++){
        tempflt  = samp[k].real * samp[k].real;
        tempflt += samp[k].imag * samp[k].imag;
        tempflt = scale*tempflt;
        mag[k] += tempflt;
      }

/* overlap the new samples with the old */
      for(k = 0 ; k < ovlap ; k++) sig[k] = sig[k+FFT_LENGTH-ovlap];
      for( ; k < FFT_LENGTH ; k++) sig[k] = getinput();
    }

/*  Take log after averaging the magnitudes.  */
    for (k=0; k<FFT_LENGTH/2; k++){
      tempflt = mag[k];
      if(tempflt < 1.e-10f) tempflt = 1.e-10f;
      sendout(10.0f*log10(tempflt));
    }
  }
}
```

LISTING 5.1 (*Continued*)

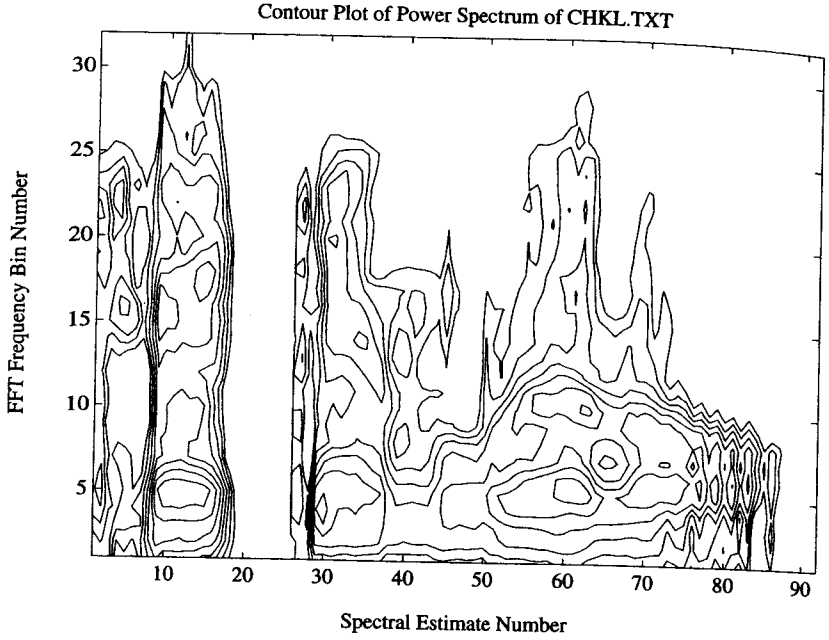

FIGURE 5.2 Contour plot of the power spectrum versus frequency and time obtained using the RTPSE program with the input file CHKL.TXT. Contours are at 5 dB intervals and the entire 2D power spectrum is normalized to 0 dB.

quency content of the "chi" part of "chicken" and the lower frequency content of "little" are clearly indicated.

5.1.2 Doppler Radar Processing

Radar signals are normally transmitted at a very high frequency (usually greater than 100 MHz), but with a relatively narrow bandwidth (several MHz). For this reason most radar signals are processed after mixing them down to baseband using a quadrature demodulator (see Skolnik, 1980). This gives a complex signal that can be processed digitally in real-time to produce a display for the radar operator. One type of display is a moving target indicator (MTI) display where moving targets are separated from stationary targets by signal processing. It is often important to know the speed and direction of the moving targets. Stationary targets are frequently encountered in radar because of fixed obstacles (antenna towers, buildings, trees) in the radar beam's path. The beam is not totally blocked by these targets, but the targets do return a large echo back to the receiver. These

targets can be removed by determining their average amplitude from a series of echoes and subtracting them from each received echo. Any moving target will not be subtracted and can be further processed. A simple method to remove stationary echoes is to simply subtract successive echoes from each other (this is a simple highpass filter of the Doppler signal). The mean frequency of the remaining Doppler signal of the moving targets can then be determined using the complex FFT.

Listing 5.2 shows the program RADPROC.C, which performs the DSP required to remove stationary targets and then estimate the frequency of the remaining Doppler signal. In order to illustrate the operation of this program, the test data file RADAR.DAT was generated. These data represent the simulated received signal from a stationary target

```
#include <stdlib.h>
#include <math.h>
#include "rtdspc.h"

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

RADPROC.C - Real-Time Radar processing

This program subtracts successive complex echo signals to
remove stationary targets from a radar signal and then
does power spectral estimation on the resulting samples.
The mean frequency is then estimated by finding the peak of the
FFT spectrum.

Requires complex input (stored real,imag) with 12
consecutive samples representing 12 range locations from
each echo.

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

/* FFT length must be a power of 2 */
#define FFT_LENGTH 16
#define M 4             /* must be log2(FFT_LENGTH) */
#define ECHO_SIZE 12

void main()
{
  int           i,j,k;
  float         tempflt,rin,iin,p1,p2;
  static float  mag[FFT_LENGTH];
  static COMPLEX echos[ECHO_SIZE][FFT_LENGTH];
  static COMPLEX last_echo[ECHO_SIZE];
```

LISTING 5.2 Program RADPROC to perform real-time radar signal processing using the FFT. (*Continued*)

```
/* read in the first echo */
  for(i = 0 ; i < ECHO_SIZE ; i++) {
     last_echo[i].real = getinput();
     last_echo[i].imag = getinput();
  }

  for(;;) {
    for (j=0; j< FFT_LENGTH; j++){

/* remove stationary targets by subtracting pairs (highpass filter) */
       for (k=0; k< ECHO_SIZE; k++){
         rin = getinput();
         iin = getinput();
         echos[k][j].real = rin - last_echo[k].real;
         echos[k][j].imag = iin - last_echo[k].imag;
         last_echo[k].real = rin;
         last_echo[k].imag = iin;
       }
     }
/* do FFTs on each range sample */
     for (k=0; k< ECHO_SIZE; k++) {

       fft(echos[k],M);

       for(j = 0 ; j < FFT_LENGTH ; j++) {
         tempflt  = echos[k][j].real * echos[k][j].real;
         tempflt += echos[k][j].imag * echos[k][j].imag;
         mag[j] = tempflt;
       }
/* find the biggest magnitude spectral bin and output */
       tempflt = mag[0];
       i=0;
       for(j = 1 ; j < FFT_LENGTH ; j++) {
         if(mag[j] > tempflt) {
           tempflt = mag[j];
           i=j;
         }
       }
/* interpolate the peak loacation */
       p1 = mag[i] - mag[i-1];
       p2 = mag[i] - mag[i+1];
       sendout((float)i + (p1-p2)/(2*(p1+p2+1e-30)));
     }
   }
}
```

LISTING 5.2 *(Continued)*

added to a moving target signal with Gaussian noise. The data is actually a 2D matrix representing 12 consecutive complex samples (real,imag) along the echo in time (representing 12 consecutive range locations) with each of 33 echoes following one after another. The sampling rates and target speeds are not important to the illustration of the program. The output of the program is the peak frequency location from the 16-point FFT in bins (0 to 8 are positive frequencies and 9 to 15 are –7 to –1 negative frequency bins). A simple (and efficient) parabolic interpolation is used to give a fractional output in the results. The output from the RADPROC program using the RADAR.DAT as input is 24 consecutive numbers with a mean value of 11 and a small standard deviation due to the added noise. The first 12 numbers are from the first set of 16 echoes and the last 12 numbers are from the remaining echoes.

5.2 PARAMETRIC SPECTRAL ESTIMATION

The parametric approach to spectral estimation attempts to describe a signal as a result from a simple system model with a random process as input. The result of the estimator is a small number of parameters that completely characterize the system model. If the model is a good choice, then the spectrum of the signal model and the spectrum from other spectral estimators should be similar. The most common parametric spectral estimation models are based on AR, MA, or ARMA random process models as discussed in section 1.6.6 of chapter 1. Two simple applications of these models are presented in the next two sections.

5.2.1 ARMA Modeling of Signals

Figure 5.3 shows the block diagram of a system modeling problem that will be used to illustrate the adaptive IIR LMS algorithm discussed in detail in section 1.7.2 of chapter 1. Listing 5.3 shows the main program ARMA.C, which first filters white noise (generated using the Gaussian noise generator described in section 4.2.1 of chapter 4) using a second-order IIR filter, and then uses the LMS algorithm to adaptively determine the filter function.

Listing 5.4 shows the function **`iir_biquad`**, which is used to filter the white noise, and Listing 5.5 shows the adaptive filter function, which implements the LMS algorithm in a way compatible with real-time input. Although this is a simple ideal example where exact convergence can be obtained, this type of adaptive system can also be used to model more complicated systems, such as communication channels or control systems. The white noise generator can be considered a training sequence which is known to the algorithm; the algorithm must determine the transfer function of the system. Figure 5.4 shows the error function for the first 7000 samples of the adaptive process. The error reduces relatively slowly due to the poles and zeros that must be determined. FIR LMS algorithms generally converge much faster when the system can be modeled as a MA system (see section 5.5.2 for an FIR LMS example). Figure 5.5 shows the path of the pole coefficients (b0,b1) as they adapt to the final result where b0 = 0.748 and b1 = –0.272.

(text continues on page 198)

```
/* 2 poles (2 b coefs) and 2 zeros (3 a coefs) adaptive iir biquad filter */

float iir_adapt_filter(float input,float d,float *a,float *b)
{
    int i;
    static float out_hist1,out_hist2;
    static float beta[2],beta_h1[2],beta_h2[2];
    static float alpha[3],alpha_h1[3],alpha_h2[3];
    static float in_hist[3];
    float output,e;

    output = out_hist1 * b[0];
    output += out_hist2 * b[1];              /* poles */

    in_hist[0] = input;
    for(i = 0 ; i < 3 ; i++)
        output += in_hist[i] * a[i];          /* zeros */

/* calclulate alpha and beta update coefficients */
    for(i = 0 ; i < 3 ; i++)
        alpha[i] = in_hist[i] + b[0]*alpha_h1[i] + b[1]*alpha_h2[i];

    beta[0] = out_hist1 + b[0]*beta_h1[0] + b[1]*beta_h2[0];
    beta[1] = out_hist2 + b[0]*beta_h1[1] + b[1]*beta_h2[1];

/* error calculation */
    e = d - output;
/* update coefficients */
    a[0] += e*0.2*alpha[0];
    a[1] += e*0.1*alpha[1];
    a[2] += e*0.06*alpha[2];

    b[0] += e*0.04*beta[0];
    b[1] += e*0.02*beta[1];

/* update history for alpha */
    for(i = 0 ; i < 3 ; i++) {
        alpha_h2[i] = alpha_h1[i];
        alpha_h1[i] = alpha[i];
    }

/* update history for beta */
    for(i = 0 ; i < 2 ; i++) {
```

LISTING 5.5 Function **iir_adapt_filter(input,d,a,b)**, which implements an LMS adaptive second-order IIR filter (contained in ARMA.C) (*Continued*)


```
        beta_h2[i] = beta_h1[i];
        beta_h1[i] = beta[i];
    }

/* update input/output history */
    out_hist2 = out_hist1;
    out_hist1 = output;

    in_hist[2] = in_hist[1];
    in_hist[1] = input;

    return(output);
}
```

LISTING 5.5 (*Continued*)

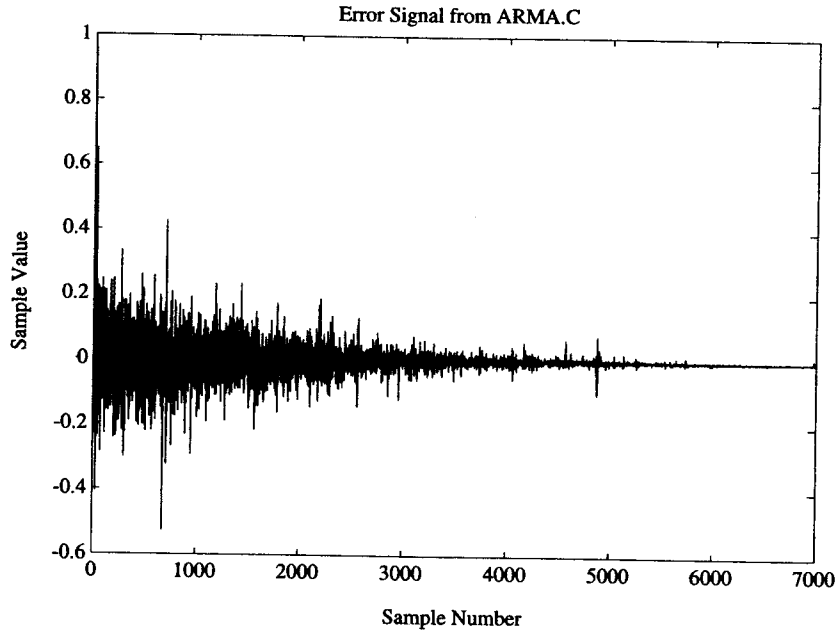

FIGURE 5.4 Error signal during the IIR adaptive process, illustrated by the program ARMA.C.

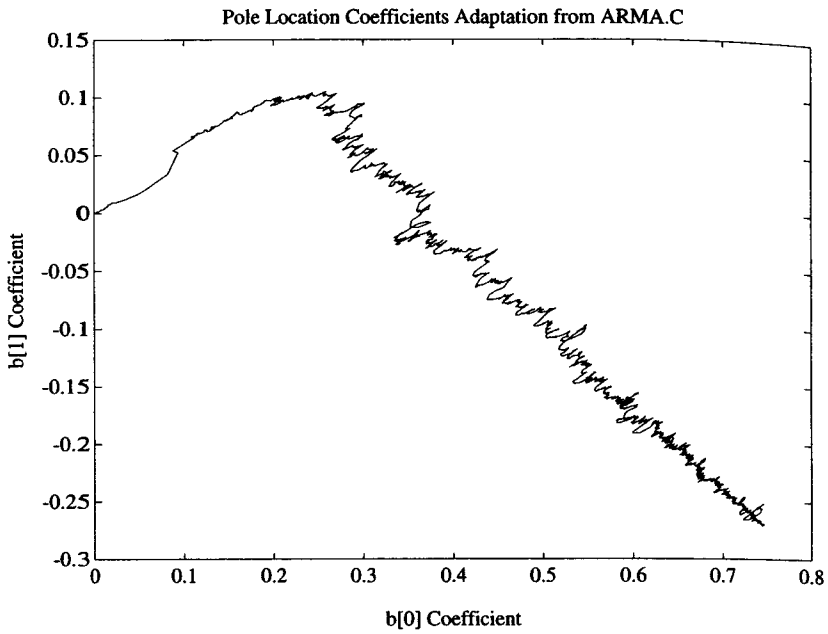

FIGURE 5.5 Pole coefficients (b0,b1) during the IIR adaptive process, illustrated by the program ARMA.C.

5.2.2 AR Frequency Estimation

The frequency of a signal can be estimated in a variety of ways using spectral analysis methods (one of which is the FFT illustrated in section 5.1.2). Another parametric approach is based on modeling the signal as resulting from an AR process with a single complex pole. The angle of the pole resulting from the model is directly related to the mean frequency estimate. This model approach can easily be biased by noise or other signals but provides a highly efficient real-time method to obtain mean frequency information.

The first step in the AR frequency estimation process is to convert the real signal input to a complex signal. This is not required when the signal is already complex, as is the case for a radar signal. Real-to-complex conversion can be done relatively simply by using a Hilbert transform FIR filter. The output of the Hilbert transform filter gives the imaginary part of the complex signal and the input signal is the real part of the complex signal. Listing 5.6 shows the program ARFREQ.C, which implements a 35-point Hilbert transform and the AR frequency estimation process. The AR frequency estimate determines the average frequency from the average phase differences between consecutive


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "rtdspc.h"

/* ARFREQ.C - take real data in one record and determine the
1st order AR frequency estimate versus time.  Uses a Hilbert transform
to convert the real signal to complex representation */

main()
{
/* 35 point Hilbert transform FIR filter cutoff at 0.02 and 0.48
   +/- 0.5 dB ripple in passband, zeros at 0 and 0.5 */

    static float  fir_hilbert35[35] = {
    0.038135,    0.000000,    0.024179,    0.000000,    0.032403,
    0.000000,    0.043301,    0.000000,    0.058420,    0.000000,
    0.081119,    0.000000,    0.120167,    0.000000,    0.207859,
    0.000000,    0.635163,    0.000000,   -0.635163,    0.000000,
   -0.207859,    0.000000,   -0.120167,    0.000000,   -0.081119,
    0.000000,   -0.058420,    0.000000,   -0.043301,    0.000000,
   -0.032403,    0.000000,   -0.024179,    0.000000,   -0.038135
                          };

    static float hist[34];
    int i,winlen;
    float sig_real,sig_imag,last_real,last_imag;
    float cpi,xr,xi,freq;

    cpi = 1.0/(2.0*PI);
    winlen = 32;

    last_real = 0.0;
    last_imag = 0.0;
    for(;;) {
/* determine the phase difference between sucessive samples */
      xr = 0.0;
      xi = 0.0;
      for(i = 0 ; i < winlen ; i++) {
        sig_imag = fir_filter(getinput(),fir_hilbert35,35,hist);
        sig_real = hist[16];
        xr += sig_real * last_real;
        xr += sig_imag * last_imag;
        xi += sig_real * last_imag;
```

LISTING 5.6 Program ARFREQ.C, which calculates AR frequency estimates in real-time. (*Continued*)

```
        xi -= sig_imag * last_real;
        last_real = sig_real;
        last_imag = sig_imag;
      }
/* make sure the result is valid, give 0 if not */
      if(fabs(xr) > 1e-10)
        freq = cpi*atan2(xi,xr);
      else
        freq = 0.0;
      sendout(freq);
    }
}
```

LISTING 5.6 *(Continued)*

complex samples. The arc tangent is used to determine the phase angle of the complex results. Because the calculation of the arc tangent is relatively slow, several simplifications can be made so that only one arc tangent is calculated for each frequency estimate. Let x_n be the complex sequence after the Hilbert transform. The phase difference is

$$\Phi_n = \arg[x_n] - \arg[x_{n-1}] = \arg[x_n x_{n-1}^*]. \tag{5.1}$$

The average frequency estimate is then

$$\hat{f} = \frac{\sum_{n=0}^{wlen-1} \Phi_n}{2\pi\, wlen} \cong \frac{\arg\left[\sum_{n=0}^{wlen-1} x_n x_{n-1}^*\right]}{2\pi}, \tag{5.2}$$

where the last approximation weights the phase differences based on the amplitude of the complex signal and reduces the number of arc tangents to one per estimate. The constant *wlen* is the window length (**winlen** in program ARFREQ) and controls the number of phase estimates averaged together. Figure 5.6 shows the results from the ARFREQ program when the CHKL.TXT speech data is used as input. Note that the higher frequency content of the "chi" sound is easy to identify.

3 SPEECH PROCESSING

Communication channels never seem to have enough bandwidth to carry the desired speech signals from one location to another for a reasonable cost. Speech compression attempts to improve this situation by sending the speech signal with as few bits per second as possible. The same channel can now be used to send a larger number of speech signals at a lower cost. Speech compression techniques can also be used to reduce the amount of memory needed to store digitized speech.

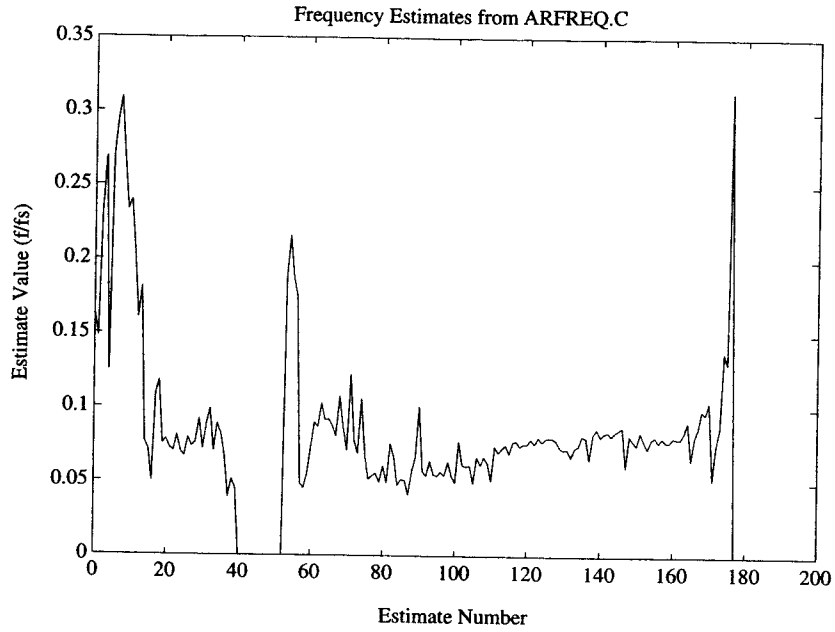

FIGURE 5.6 Frequency estimates from program ARFREQ.C, using the CHKL.DAT speech data as input.

5.3.1 Speech Compression

The simplest way to reduce the bandwidth required to transmit speech is to simply reduce the number bits per sample that are sent. If this is done in a linear fashion, then the quality of the speech (in terms of signal-to-noise ratio) will degrade rapidly when less than 8 bits per sample are used. Speech signals require 13 or 14 bits with linear quantization in order to produce a digital representation of the full range of speech signals encountered in telephone applications. The International Telegraph and Telephone Consultative Committee (CCITT, 1988) recommendation G.711 specifies the basic pulse code modulation (PCM) algorithm, which uses a logarithmic compression curve called μ-law. μ-law (see section 1.5.1 in chapter 1) is a piecewise linear approximation of a logarithmic transfer curve consisting of 8 linear segments. It compresses a 14-bit linear speech sample down to 8 bits. The sampling rate is 8000 Hz of the coded output. A compression ratio of 1.75:1 is achieved by this method without much computational complexity. Speech quality is not degraded significantly, but music and other audio signals would be degraded. Listing 5.7 shows the program MULAW.C, which encodes and decodes a speech signal using μ-law compression. The encode and decode algorithms that use tables to implement the com-

```
nclude <stdlib.h>
nclude <stdio.h>
nclude "rtdspc.h"
nclude "mu.h"

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

AW.C - PROGRAM TO DEMONSTRATE MU LAW SPEECH COMPRESSION

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

in()

  int i,j;
  for(;;) {
      i = (int) getinput();

 encode 14 bit linear input to mu-law */
      j = abs(i);
      if(j > 0x1fff) j = 0x1fff;
      j = invmutab[j/2];
      if(i < 0) j |= 0x80;

 decode the 8 bit mu-law and send out */
      sendout((float)mutab[j]);
  }
```

LISTING 5.7 Program MULAW.C, which encodes and decodes a speech signal using μ-law compression.

pression are also shown in this listing. Because the tables are rather long, they are in the include file MU.H.

5.3.2 ADPCM (G.722)

The CCITT recommendation G.722 is a standard for digital encoding of speech and audio signals in the frequency range from 50 Hz to 7000 Hz. The G.722 algorithm uses sub-band adaptive differential pulse code modulation (ADPCM) to compress 14-bit, 16 kHz samples for transmission or storage at 64 kbits/sec (a compression ratio of 3.5:1). Because the G.722 method is a wideband standard, high-quality telephone network applications as well as music applications are possible. If the sampling rate is increased, the same algorithm can be used for good quality music compression.

The G.722 program is organized as a set of functions to optimize memory usage and make it easy to follow. This program structure is especially efficient for G.722, since most of the functions are shared between the higher and lower sub-bands. Many of the

functions are also shared by both the encoder and decoder of both sub-bands. All of the functions are performed using fixed-point arithmetic, because this is specified in the CCITT recommendation. A floating-point version of the G.722 C code is included on the enclosed disk. The floating-point version runs faster on the DSP32C processor, which has limited support for the shift operator used extensively in the fixed-point implementation. Listing 5.8 shows the main program G722MAIN.C, which demonstrates the algorithm by encoding and decoding the stored speech signal "chicken little," and then operates on the real-time speech signal from **`getinput()`**. The output decoded signal is played using **`sendout()`** with an effective sample rate of 16 kHz (one sample is interpolated using simple linear interpolation giving an actual sample rate for this example of

```
#include <stdlib.h>
#include "rtdspc.h"

/* Main program for g722 encode and decode demo for 210X0 */

    extern int encode(int,int);
    extern void decode(int);
    extern void reset();

/* outputs of the decode function */
    extern int xout1,xout2;

    int chkl_coded[6000];
    extern int pm chkl[];

void main()
{
    int i,j,t1,t2;
    float xf1 = 0.0;
    float xf2 = 0.0;

/* reset, initialize required memory */
    reset();

/* code the speech, interpolate because it was recorded at 8000 Hz */
    for(i = 0 ; i < 6000 ; i++) {
        t1=64*chkl[i];
        t2=32*(chkl[i]+chkl[i+1]);
        chkl_coded[i]=encode(t1,t2);
    }

/* interpolate output to 32 KHz */
    for(i = 0 ; i < 6000 ; i++) {
```

LISTING 5.8 The main program (G722MAIN.C), which demonstrates the ADPCM algorithm in real-time. (*Continued*)

```
            decode(chkl_coded[i]);
            xf1 = (float)xout1;
            sendout(0.5*xf2+0.5*xf1);
            sendout(xf1);
            xf2 = (float)xout2;
            sendout(0.5*xf2+0.5*xf1);
            sendout(xf2);
        }

    /* simulate a 16 KHz sampling rate (actual is 32 KHz) */
    /* note: the g722 standard calls for 16 KHz for voice operation */
        while(1) {
            t1=0.5*(getinput()+getinput());
            t2=0.5*(getinput()+getinput());

            j=encode(t1,t2);
            decode(j);

            xf1 = (float)xout1;
            sendout(0.5*(xf1+xf2));
            sendout(xf1);
            xf2 = (float)xout2;
            sendout(0.5*(xf2+xf1));
            sendout(xf2);
        }
    }
```

LISTING 5.8 *(Continued)*

32 kHz). Listing 5.9 shows the **encode** function, and Listing 5.10 shows the **decode** function; both are contained in G.722.C. Listing 5.11 shows the functions **filtez**, **filtep**, **quantl**, **invqxl**, **logscl**, **scalel**, **upzero**, **uppol2**, **uppol1**, **invqah**, and **logsch**, which are used by the **encode** and **decode** functions. In Listings 5.9, 5.10, and 5.11, the global variable definitions and data tables have been omitted for clarity.

(text continues on page 215)

```
722 encode function two ints in, one int out */

encode(int xin1,int xin2)

int i;
int *h_ptr;
int *tqmf_ptr,*tqmf_ptr1;
long int xa,xb;
int xl,xh;
```

LISTING 5.9 Function **encode(xin1,xin2)** (contained in G.722.C). *(Continued)*


```
    int decis;
    int         sh;         /* this comes from adaptive predictor */
    int         eh;
    int         dh;
    int         il,ih;
    int szh,sph,ph,yh;
    int szl,spl,sl,el;

/* encode: put input samples in xin1 = first value, xin2 = second value */
/* returns il and ih stored together */

/* transmit quadrature mirror filters implemented here */
    h_ptr = h;
    tqmf_ptr = tqmf;
    xa = (long)(*tqmf_ptr++) * (*h_ptr++);
    xb = (long)(*tqmf_ptr++) * (*h_ptr++);
/* main multiply accumulate loop for samples and coefficients */
    for(i = 0 ; i < 10 ; i++) {
        xa += (long)(*tqmf_ptr++) * (*h_ptr++);
        xb += (long)(*tqmf_ptr++) * (*h_ptr++);
    }
/* final mult/accumulate */
    xa += (long)(*tqmf_ptr++) * (*h_ptr++);
    xb += (long)(*tqmf_ptr) * (*h_ptr++);

/* update delay line tqmf */
    tqmf_ptr1 = tqmf_ptr - 2;
    for(i = 0 ; i < 22 ; i++) *tqmf_ptr-- = *tqmf_ptr1--;
    *tqmf_ptr-- = xin1;
    *tqmf_ptr = xin2;

    xl = (xa + xb) >> 15;
    xh = (xa - xb) >> 15;

/* end of quadrature mirror filter code */

/* into regular encoder segment here */
/* starting with lower sub band encoder */

/* filtez - compute predictor output section - zero section */

    szl = filtez(delay_bpl,delay_dltx);

/* filtep - compute predictor output signal (pole section) */

    spl = filtep(rlt1,al1,rlt2,al2);
```

LISTING 5.9 *(Continued)*

```
:ompute the predictor output value in the lower sub_band encoder */

 sl = szl + spl;
 el = xl - sl;

 uantl: quantize the difference signal */

 il = quantl(el,detl);

 nvqxl: does both invqal and invqbl- computes quantized difference signal */
 or invqbl, truncate by 2 lsbs, so mode = 3 */

 nvqal case with mode = 3 */
 dlt = ((long)detl*qq4_code4_table[il >> 2]) >> 15;

 ogscl: updates logarithmic quant. scale factor in low sub band*/
 nbl = logscl(il,nbl);

 calel: compute the quantizer scale factor in the lower sub band*/
 alling parameters nbl and 8 (constant such that scalel can be scaleh) */
 detl = scalel(nbl,8);

 arrec - simple addition to compute recontructed signal for adaptive pred */
 plt = dlt + szl;

 pzero: update zero section predictor coefficients (sixth order)*/
 alling parameters: dlt, dlti(circ pointer for delaying */
 lt1, dlt2, ..., dlt6 from dlt */
 bpli (linear_buffer in which all six values are delayed */
 eturn params:      updated bpli, delayed dltx */

 upzero(dlt,delay_dltx,delay_bpl);

 ppol2- update second predictor coefficient apl2 and delay it as al2 */
 alling parameters: al1, al2, plt, plt1, plt2 */

 al2 = uppol2(al1,al2,plt,plt1,plt2);

 ppol1 :update first predictor coefficient apl1 and delay it as al1 */
 alling parameters: al1, apl2, plt, plt1 */

 al1 = uppol1(al1,al2,plt,plt1);

 econs : compute recontructed signal for adaptive predictor */
 rlt = sl + dlt;

 one with lower sub_band encoder; now implement delays for next time*/
```

LISTING 5.9 *(Continued)*


```
    rlt2 = rlt1;
    rlt1 = rlt;
    plt2 = plt1;
    plt1 = plt;

/* high band encode */

    szh = filtez(delay_bph,delay_dhx);

    sph = filtep(rh1,ah1,rh2,ah2);

/* predic: sh = sph + szh */
    sh = sph + szh;
/* subtra: eh = xh - sh */
    eh = xh - sh;

/* quanth - quantization of difference signal for higher sub-band */
/* quanth: in-place for speed params: eh, deth (has init. value) */
/* return: ih */
    if(eh >= 0) {
        ih = 3;      /* 2,3 are pos codes */
    }
    else {
        ih = 1;      /* 0,1 are neg codes */
    }
    decis = (564L*(long)deth) >> 12L;
    if(abs(eh) > decis) ih--;      /* mih = 2 case */

/* invqah: in-place compute the quantized difference signal
 in the higher sub-band*/

    dh = ((long)deth*qq2_code2_table[ih]) >> 15L ;

/* logsch: update logarithmic quantizer scale factor in hi sub-band*/

    nbh = logsch(ih,nbh);

/* note : scalel and scaleh use same code, different parameters */
    deth = scalel(nbh,10);

/* parrec - add pole predictor output to quantized diff. signal(in place)*/
    ph = dh + szh;

/* upzero: update zero section predictor coefficients (sixth order) */
/* calling parameters: dh, dhi(circ), bphi (circ) */
/* return params: updated bphi, delayed dhx */

    upzero(dh,delay_dhx,delay_bph);
```

LISTING 5.9 *(Continued)*

```
uppol2: update second predictor coef aph2 and delay as ah2 */
alling params: ah1, ah2, ph, ph1, ph2 */
eturn params:  aph2 */

ah2 = uppol2(ah1,ah2,ph,ph1,ph2);

uppol1:  update first predictor coef. aph2 and delay it as ah1 */

ah1 = uppol1(ah1,ah2,ph,ph1);

econs for higher sub-band */
yh = sh + dh;

lone with higher sub-band encoder, now Delay for next time */
rh2 = rh1;
rh1 = yh;
ph2 = ph1;
ph1 = ph;

ultiplexing ih and il to get signals together */
return(il | (ih << 6));
```

LISTING 5.9 *(Continued)*

```
lecode function, result in xout1 and xout2 */

l decode(int input)

int i;
int xa1,xa2;     /* qmf accumulators */
int *h_ptr;
int pm *ac_ptr,*ac_ptr1,*ad_ptr,*ad_ptr1;
int         ilr,ih;
int xs,xd;
int         rl,rh;
int     dl;

plit transmitted word from input into ilr and ih */
ilr = input & 0x3f;
ih = input >> 6;

OWER SUB_BAND DECODER */

iltez: compute predictor output for zero section */
```

LISTING 5.10 Function **decode(input)** (contained in G.722.C). *(Continued)*


```
    dec_szl = filtez(dec_del_bpl,dec_del_dltx);

/* filtep: compute predictor output signal for pole section */

    dec_spl = filtep(dec_rlt1,dec_al1,dec_rlt2,dec_al2);

    dec_sl = dec_spl + dec_szl;

/* invqxl: compute quantized difference signal for adaptive predic in low sb */
    dec_dlt = ((long)dec_detl*qq4_code4_table[ilr >> 2]) >> 15;

/* invqxl: compute quantized difference signal for decoder output in low sb */
    dl = ((long)dec_detl*qq6_code6_table[ilr]) >> 15;

    rl = dl + dec_sl;

/* logscl: quantizer scale factor adaptation in the lower sub-band */

    dec_nbl = logscl(ilr,dec_nbl);

/* scalel: computes quantizer scale factor in the lower sub band */

    dec_detl = scalel(dec_nbl,8);

/* parrec - add pole predictor output to quantized diff. signal(in place) */
/* for partially reconstructed signal */

    dec_plt = dec_dlt + dec_szl;

/* upzero: update zero section predictor coefficients */

    upzero(dec_dlt,dec_del_dltx,dec_del_bpl);

/* uppol2: update second predictor coefficient apl2 and delay it as al2 */

    dec_al2 = uppol2(dec_al1,dec_al2,dec_plt,dec_plt1,dec_plt2);

/* uppol1: update first predictor coef. (pole setion) */

    dec_al1 = uppol1(dec_al1,dec_al2,dec_plt,dec_plt1);

/* recons : compute recontructed signal for adaptive predictor */
    dec_rlt = dec_sl + dec_dlt;

/* done with lower sub band decoder, implement delays for next time */
```

LISTING 5.10 *(Continued)*

```
    dec_rlt2 = dec_rlt1;
    dec_rlt1 = dec_rlt;
    dec_plt2 = dec_plt1;
    dec_plt1 = dec_plt;

* HIGH SUB-BAND DECODER */

* filtez: compute predictor output for zero section */

    dec_szh = filtez(dec_del_bph,dec_del_dhx);

* filtep: compute predictor output signal for pole section */

    dec_sph = filtep(dec_rh1,dec_ah1,dec_rh2,dec_ah2);

* predic:compute the predictor output value in the higher sub_band decoder */

    dec_sh = dec_sph + dec_szh;

* invqah: in-place compute the quantized difference signal
in the higher sub_band */

    dec_dh = ((long)dec_deth*qq2_code2_table[ih]) >> 15L ;

* logsch: update logarithmic quantizer scale factor in hi sub band */

    dec_nbh = logsch(ih,dec_nbh);

* scalel: compute the quantizer scale factor in the higher sub band */

    dec_deth = scalel(dec_nbh,10);

* parrec: compute partially recontructed signal */
    dec_ph = dec_dh + dec_szh;

* upzero: update zero section predictor coefficients */

    upzero(dec_dh,dec_del_dhx,dec_del_bph);

*uppol2: update second predictor coefficient aph2 and delay it as ah2 */

    dec_ah2 = uppol2(dec_ah1,dec_ah2,dec_ph,dec_ph1,dec_ph2);

* uppol1: update first predictor coef. (pole setion) */

    dec_ah1 = uppol1(dec_ah1,dec_ah2,dec_ph,dec_ph1);
```

LISTING 5.10 *(Continued)*


```
/* recons : compute recontructed signal for adaptive predictor */
    rh = dec_sh + dec_dh;

/* done with high band decode, implementing delays for next time here */
    dec_rh2 = dec_rh1;
    dec_rh1 = rh;
    dec_ph2 = dec_ph1;
    dec_ph1 = dec_ph;

/* end of higher sub_band decoder */

/* end with receive quadrature mirror filters */
    xd = rl - rh;
    xs = rl + rh;

/* receive quadrature mirror filters implemented here */
    h_ptr = h;
    ac_ptr = accumc;
    ad_ptr = accumd;
    xa1 = (long)xd * (*h_ptr++);
    xa2 = (long)xs * (*h_ptr++);
/* main multiply accumulate loop for samples and coefficients */
    for(i = 0 ; i < 10 ; i++) {
        xa1 += (long)(*ac_ptr++) * (*h_ptr++);
        xa2 += (long)(*ad_ptr++) * (*h_ptr++);
    }
/* final mult/accumulate */
    xa1 += (long)(*ac_ptr) * (*h_ptr++);
    xa2 += (long)(*ad_ptr) * (*h_ptr++);

/* scale by 2^14 */
    xout1 = xa1 >> 14;
    xout2 = xa2 >> 14;

/* update delay lines */
    ac_ptr1 = ac_ptr - 1;
    ad_ptr1 = ad_ptr - 1;
    for(i = 0 ; i < 10 ; i++) {
        *ac_ptr-- = *ac_ptr1--;
        *ad_ptr-- = *ad_ptr1--;
    }
    *ac_ptr = xd;
    *ad_ptr = xs;

}
```

LISTING 5.10 *(Continued)*

```
.ltez - compute predictor output signal (zero section) */
iput: bpl1-6 and dlt1-6, output: szl */

:iltez(int *bpl,int *dlt)

.nt i;
.ong int zl;
:l = (long)(*bpl++) * (*dlt++);
:or(i = 1 ; i < 6 ; i++)
    zl += (long)(*bpl++) * (*dlt++);

'eturn((int)(zl >> 14));   /* x2 here */


.ltep - compute predictor output signal (pole section) */
iput rlt1-2 and al1-2, output spl */

'iltep(int rlt1,int al1,int rlt2,int al2)

ong int pl;
'l = (long)al1*rlt1;
'l += (long)al2*rlt2;
'eturn((int)(pl >> 14));   /* x2 here */


antl - quantize the difference signal in the lower sub-band */
uantl(int el,int detl)

nt ril,mil;
ong int wd,decis;

s of difference signal */
d = abs(el);
termine mil based on decision levels and detl gain */
or(mil = 0 ; mil < 30 ; mil++) {
   decis = (decis_levl[mil]*(long)detl) >> 15L;
   if(wd < decis) break;

 mil=30 then wd is less than all decision levels */
f(el >= 0) ril = quant26bt_pos[mil];
lse ril = quant26bt_neg[mil];
eturn(ril);
```

LISTING 5.11 Functions used by the encode and decode algorithms of G.722 (contained in G.722.C). (*Continued*)


```
/* logscl - update the logarithmic quantizer scale factor in lower sub-band */
/* note that nbl is passed and returned */

int logscl(int il,int nbl)
{
    long int wd;
    wd = ((long)nbl * 127L) >> 7L;   /* leak factor 127/128 */
    nbl = (int)wd + wl_code_table[il >> 2];
    if(nbl < 0) nbl = 0;
    if(nbl > 18432) nbl = 18432;
    return(nbl);
}

/* scalel: compute the quantizer scale factor in the lower or upper sub-band*/

int scalel(int nbl,int shift_constant)
{
    int wd1,wd2,wd3;
    wd1 = (nbl >> 6) & 31;
    wd2 = nbl >> 11;
    wd3 = ilb_table[wd1] >> (shift_constant + 1 - wd2);
    return(wd3 << 3);
}

/* upzero - inputs: dlt, dlti[0-5], bli[0-5], outputs: updated bli[0-5] */
/* also implements delay of bli and update of dlti from dlt */

void upzero(int dlt,int *dlti,int *bli)
{
    int i,wd2,wd3;
/*if dlt is zero, then no sum into bli */
    if(dlt == 0) {
        for(i = 0 ; i < 6 ; i++) {
            bli[i] = (int)((255L*bli[i]) >> 8L); /* leak factor of 255/256 */
        }
    }
    else {
        for(i = 0 ; i < 6 ; i++) {
            if((long)dlt*dlti[i] >= 0) wd2 = 128; else wd2 = -128;
            wd3 = (int)((255L*bli[i]) >> 8L);     /* leak factor of 255/256 */
            bli[i] = wd2 + wd3;
        }
    }
/* implement delay line for dlt */
```

LISTING 5.11 (*Continued*)

```
dlti[5] = dlti[4];
dlti[4] = dlti[3];
dlti[3] = dlti[2];
dlti[2] = dlti[1];
dlti[1] = dlti[0];
dlti[0] = dlt;


uppol2 - update second predictor coefficient (pole section) */
inputs: al1, al2, plt, plt1, plt2. outputs: apl2 */

uppol2(int al1,int al2,int plt,int plt1,int plt2)

long int wd2,wd4;
int apl2;
wd2 = 4L*(long)al1;
if((long)plt*plt1 >= 0L) wd2 = -wd2;     /* check same sign */
wd2 = wd2 >> 7;                          /* gain of 1/128 */
if((long)plt*plt2 >= 0L) {
    wd4 = wd2 + 128;                     /* same sign case */
}
else {
    wd4 = wd2 - 128;
}
apl2 = wd4 + (127L*(long)al2 >> 7L);  /* leak factor of 127/128 */

apl2 is limited to +-.75 */
if(apl2 > 12288) apl2 = 12288;
if(apl2 < -12288) apl2 = -12288;
return(apl2);


uppol1 - update first predictor coefficient (pole section) */
inputs: al1, apl2, plt, plt1. outputs: apl1 */

uppol1(int al1,int apl2,int plt,int plt1)

long int wd2;
int wd3,apl1;
wd2 = ((long)al1*255L) >> 8L;    /* leak factor of 255/256 */
if((long)plt*plt1 >= 0L) {
    apl1 = (int)wd2 + 192;       /* same sign case */
}
else {
    apl1 = (int)wd2 - 192;
}
```

LISTING 5.11 *(Continued)*


```
/* note: wd3= .9375-.75 is always positive */
    wd3 = 15360 - apl2;              /* limit value */
    if(apl1 > wd3) apl1 = wd3;
    if(apl1 < -wd3) apl1 = -wd3;
    return(apl1);
}

/* logsch - update the logarithmic quantizer scale factor in higher sub-band */
/* note that nbh is passed and returned */

int logsch(int ih,int nbh)
{
    int wd;
    wd = ((long)nbh * 127L) >> 7L;       /* leak factor 127/128 */
    nbh = wd + wh_code_table[ih];
    if(nbh < 0) nbh = 0;
    if(nbh > 22528) nbh = 22528;
    return(nbh);
}
```

LISTING 5.11 *(Continued)*

Figure 5.7 shows a block diagram of the G.722 encoder (transmitter), and Figure 5.8 shows a block diagram of the G.722 decoder (receiver). The entire algorithm has six main functional blocks, many of which use the same functions:

(1) A transmit quadrature mirror filter (QMF) that splits the frequency band into two sub-bands.

(2&3) A lower sub-band encoder and higher sub-band encoder that operate on the data produced by the transmit QMF.

(4&5) A lower sub-band decoder and higher sub-band decoder.

(6) A receive QMF that combines the outputs of the decoder into one value.

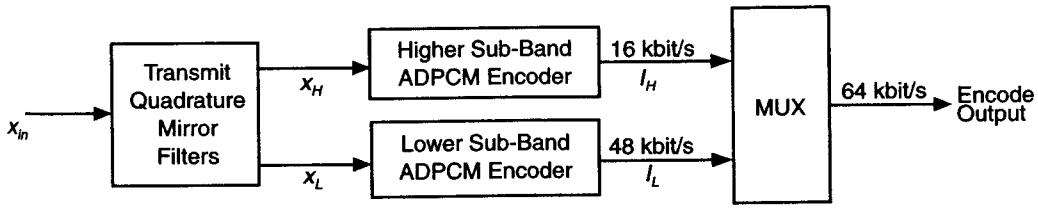

FIGURE 5.7 Block diagram of ADPCM encoder (transmitter) implemented by program G.722.C.

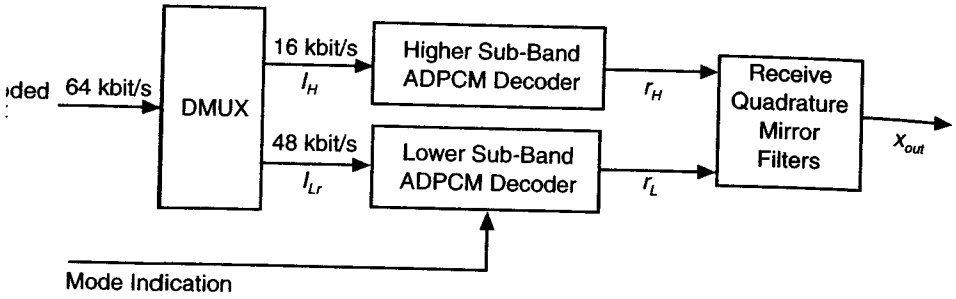

FIGURE 5.8 Block diagram of ADPCM decoder (receiver) implemented by program G.722.C.

The G.722.C functions have been checked against the G.722 specification and are fully compatible with the CCITT recommendation. The functions and program variables are named according to the functional blocks of the algorithm specification whenever possible.

Quadrature mirror filters are used in the G.722 algorithm as a method of splitting the frequency band into two sub-bands (higher and lower). The QMFs also decimate the encoder input from 16 kHz to 8 kHz (*transmit QMF*) and interpolate the decoder output from 8 kHz to 16 kHz (*receive QMF*). These filters are 24-tap FIR filters whose impulse response can be considered lowpass and highpass filters. Both the transmit and receive QMFs share the same coefficients and a delay line of the same number of taps.

Figure 5.9 shows a block diagram of the higher sub-band encoder. The lower and higher sub-band encoders operate on an estimated difference signal. The number of bits required to represent the difference is smaller than the number of bits required to represent the complete input signal. This difference signal is obtained by subtracting a predicted value from the input value:

```
el = xl - sl
eh = xh - sh
```

The predicted value, **sl** or **sh**, is produced by the adaptive predictor, which contains a second-order filter section to model poles, and a sixth-order filter section to model zeros in the input signal. After the predicted value is determined and subtracted from the input signal, the estimate signal **el** is applied to a nonlinear adaptive quantizer.

One important feature of the sub-band encoders is a feedback loop. The output of the adaptive quantizer is fed to an inverse adaptive quantizer to produce a difference signal. This difference signal is then used by the adaptive predictor to produce **sl** (the estimate of the input signal) and update the adaptive predictor.

The G.722 standard specifies an auxiliary, nonencoded data channel. While the G.722 encoder always operates at an output rate of 64 kbits per second (with 14-bit, 16kHz input samples), the decoder can accept encoded signals at 64, 56, or 48 kbps. The 56 and 48 kbps bit rates correspond to the use of the auxiliary data channel, which operates at either 8 or 16 kbps. A *mode indication signal* informs the decoder which mode is being used. This feature is not implemented in the G.722.C program.

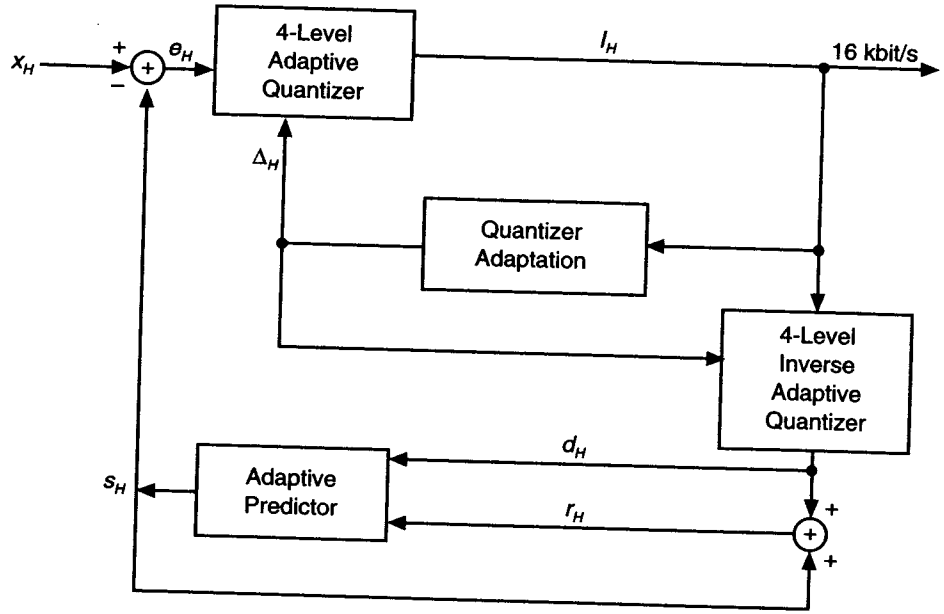

FIGURE 5.9 Block diagram of higher sub-band ADPCM encoder implemented by program G.722.C.

Figure 5.10 shows a block diagram of the higher sub-band decoder. In general, both the higher and lower sub-band encoders and decoders make the same function calls in almost the same order because they are similar in operation. For mode 1, a 60 level inverse adaptive quantizer is used in the lower sub-band, which gives the best speech quality. The higher sub-band uses a 4 level adaptive quantizer.

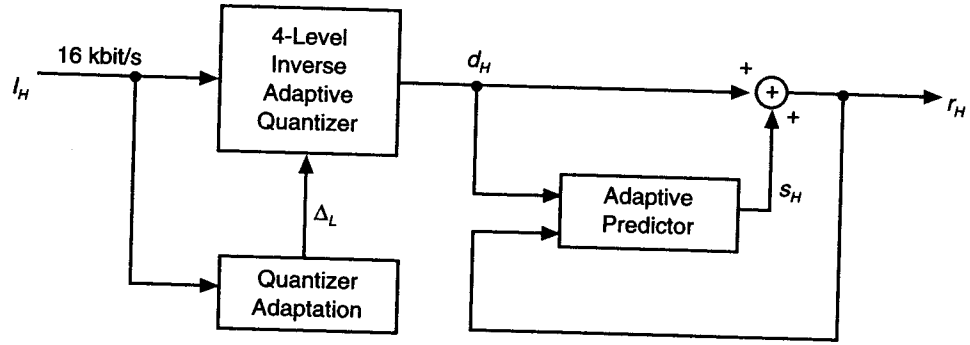

FIGURE 5.10 Block diagram of higher sub-band ADPCM decoder implemented by program G.722.C.

MUSIC PROCESSING

Music signals require more dynamic range and a much wider bandwidth than speech signals. Professional quality music processing equipment typically uses 18 to 24 bits to represent each sample and a 48 kHz or higher sampling rate. Consumer digital audio processing (in CD players, for example) is usually done with 16-bit samples and a 44.1 kHz sampling rate. In both cases, music processing is a far greater challenge to a digital signal processor than speech processing. More MIPs are required for each operation and quantization noise in filters becomes more important. In most cases DSP techniques are less expensive and can provide a higher level of performance than analog techniques.

5.4.1 Equalization and Noise Removal

Equalization refers to a filtering process where the frequency content of an audio signal is adjusted to make the source sound better, adjust for room acoustics, or remove noise that may be in a frequency band different from the desired signal. Most audio equalizers have a number of bands that operate on the audio signal in parallel with the output of each filter, added together to form the equalized signal. This structure is shown in Figure 5.11. The program EQUALIZ.C is shown in Listing 5.12. Each **gain** constant is used to adjust the relative signal amplitude of the output of each bandpass filter. The input signal is always added to the output such that if all the gain values are zero the signal is unchanged. Setting the **gain** values greater than zero will boost frequency response in each band. For example, a boost of 6 dB is obtained by setting one of the **gain** values to 1.0. The center frequencies and number of bandpass filters in analog audio equalizers vary widely from one manufacturer to another. A seven-band equalizer with center frequencies at 60,

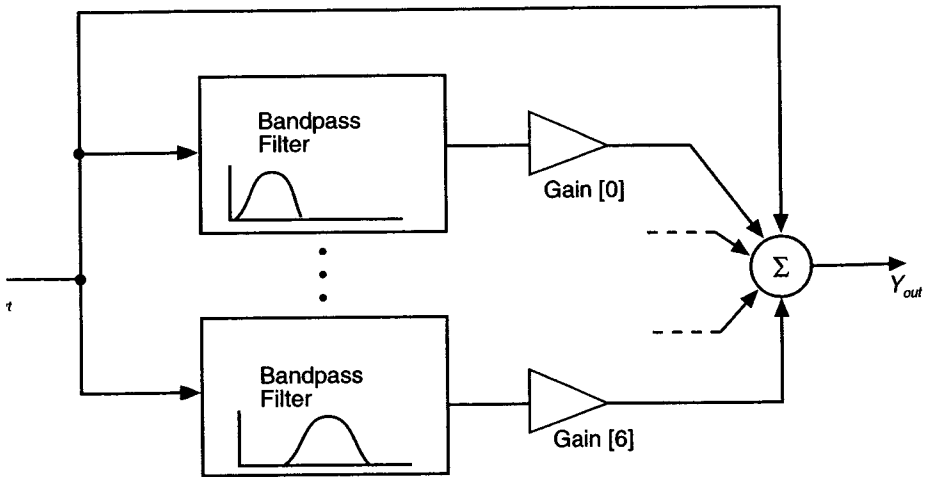

FIGURE 5.11 Block diagram of audio equalization implemented by program EQUALIZ.C.


```
#include <stdlib.h>
#include <math.h>
#include "rtdspc.h"

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

EQUALIZ.C - PROGRAM TO DEMONSTRATE AUDIO EQUALIZATION
            USING 7 IIR BANDPASS FILTERS.

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

/* gain values global so they can be changed in real-time */
/* start at flat pass through */
    float gain[7] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };

void main()
{
    int i;
    float signal_in,signal_out;

/* history arrays for the filters */
    static float hist[7][2];

/* bandpass filter coefficients for a 44.1 kHz sampling rate */
/* center freqs are 60, 150, 400, 1000, 2400, 6000, 15000 Hz */
/* at other rates center freqs are:                          */
/* at 32 kHz:       44, 109, 290,  726, 1742, 4354, 10884 Hz */
/* at 22.1 kHz:     30,  75, 200,  500, 1200, 3000,  7500 Hz */

    static float bpf[7][5] = {
  {  0.0025579741,  -1.9948111773,   0.9948840737,   0.0,  -1.0 },
  {  0.0063700872,  -1.9868060350,   0.9872598052,   0.0,  -1.0 },
  {  0.0168007612,  -1.9632060528,   0.9663984776,   0.0,  -1.0 },
  {  0.0408578217,  -1.8988473415,   0.9182843566,   0.0,  -1.0 },
  {  0.0914007276,  -1.7119922638,   0.8171985149,   0.0,  -1.0 },
  {  0.1845672876,  -1.0703823566,   0.6308654547,   0.0,  -1.0 },
  {  0.3760778010,   0.6695288420,   0.2478443682,   0.0,  -1.0 },
            };

    for(;;) {
/* sum 7 bpf outputs + input for each new sample */
        signal_out = signal_in = getinput();
        for(i = 0 ; i < 7 ; i++)
          signal_out += gain[i]*iir_filter(signal_in,bpf[i],1,hist[i]);
        sendout(signal_out);
    }
}
```

LISTING 5.12 Program EQUALIZ.C, which performs equalization on audio samples in real-time.

150, 400, 1000, 2400, 6000, and 15000 Hz is implemented by program EQUALIZ.C. The bandwidth of each filter is 60 percent of the center frequency in each case, and the sampling rate is 44100 Hz. This gives the coefficients in the example equalizer program EQUALIZ.C shown in Listing 5.12. The frequency response of the 7 filters is shown in Figure 5.12.

5.4.2 Pitch-Shifting

Changing the pitch of a recorded sound is often desired in order to allow it to mix with a new song, or for special effects where the original sound is shifted in frequency to a point where it is no longer identifiable. New sounds are often created by a series of pitch shifts and mixing processes.

Pitch-shifting can be accomplished by interpolating a signal to a new sampling rate, and then playing the new samples back at the original sampling rate (see Alles, 1980; or Smith and Gossett, 1984). If the pitch is shifted down (by an interpolation factor greater than one), the new sound will have a longer duration. If the pitch is shifted upward (by an interpolation factor less than one where some decimation is occurring), the sound becomes shorter. Listing 5.13 shows the program PSHIFT.C, which can be used to pitch-

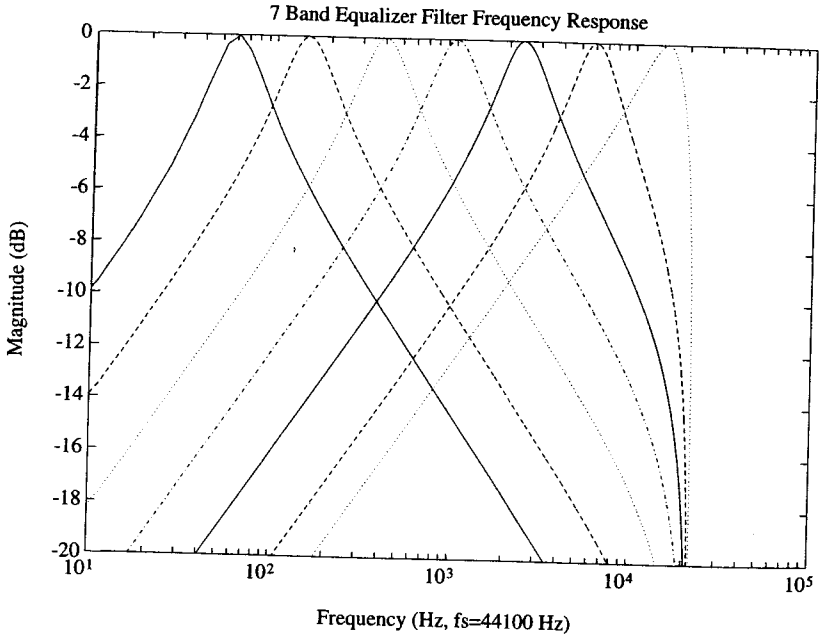

FIGURE 5.12 Frequency response of 7 filters used in program EQUALIZ.C.

shift a sound up or down by any number of semitones (12 semitones is an octave as indicated by equation 4.13). It uses a long Kaiser window filter for interpolation of the samples as illustrated in section 4.3.2 in chapter 4. The filter coefficients are calculated in the first part of the PSHIFT program before real-time input and output begins. The filtering is done with two FIR filter functions, which are shown in Listing 5.14. The history array is only updated when the interpolation point moves to the next input sample. This requires that the history update be removed from the **`fir_filter`** function discussed previously. The history is updated by the function **`fir_history_update`**. The coefficients are decimated into short polyphase filters. An interpolation ratio of up to 300 is performed and the decimation ratio is determined by the amount of pitch shift selected by the integer variable **`key`**.

```
#include <stdlib.h>
#include <math.h>
#include "rtdspc.h"

/* Kaiser Window Pitch Shift Algorithm */

/* set interpolation ratio */
    int ratio = 300;
/* passband specified, larger makes longer filters */
    float percent_pass = 80.0;
/* minimum attenuation in stopbands (dB), larger make long filters */
    float att = 50.0;
/* key value to shift by (semi-tones up or down) */
/* 12 is one octave */
    int key = -12;
    int lsize;

void main()
{
    int i,j;
    int nfilt,npair,n,k;
    float fa,fp,deltaf,beta,valizb,alpha;
    float w,ck,y,npair_inv,pi_ratio;
    float signal_in,phase,dec;
    int old_key = 0;              /* remember last key value */
    float **h;

    static float hist[100];       /* lsize can not get bigger than 100 */

    long int filter_length(float,float,float *);
    float izero(float);
```

LISTING 5.13 Program PSHIFT.C, which performs pitch shifting on audio samples in real-time. (*Continued*)

```
float fir_filter_no_update(float input,float *coef,int n,float *history);
void fir_update_history(float input,int n,float *history);

fp = percent_pass/(200.0*ratio);
fa = (200.0 - percent_pass)/(200.0*ratio);
deltaf = fa-fp;

nfilt = filter_length( att, deltaf, &beta );

lsize = nfilt/ratio;

nfilt = (long)lsize*ratio + 1;
npair = (nfilt - 1)/2;

h = (float **) calloc(ratio,sizeof(float *));
if(!h) exit(1);
for(i = 0 ; i < ratio ; i++) {
    h[i] = (float *) calloc(lsize,sizeof(float));
    if(!h[i]) exit(1);
}

/* Compute Kaiser window sample values */
i = 0;
j = 0;
valizb = 1.0 / izero(beta);
npair_inv = 1.0/npair;
pi_ratio = PI/ratio;
h[i++][j] = 0.0;        /* n = 0 case */
for (n = 1 ; n < npair ; n++) {
    k = npair - n;
    alpha = k * npair_inv;
    y = beta * sqrt(1.0 - (alpha * alpha));
    w = valizb * izero(y);
    ck = ratio*sin(k*pi_ratio)/(k*PI);
    h[i++][j] = w * ck;
    if(i == ratio) {
        i = 0;
        j++;
    }
}
force the pass through point */
h[i][lsize/2] = 1.0;

second half of response */
for(n = 1; n < npair; n++) {
    i = npair - n;          /* "from" location */
```

LISTING 5.13 *(Continued)*


```
    k = npair + n;          /* "to" location */
    h[k%ratio][k/ratio] = h[i%ratio][i/ratio];
}

/* interpolate the data by calls to fir_filter_no_update,
decimate the interpolated samples by only generating the samples
required */
    phase = 0.0;
    dec = (float)ratio;
    for( ; ;) {

/* decimation ratio for key semitones shift */
/* allow real-time updates */
        if(key != old_key) {
            dec = ratio*pow(2.0,0.0833333333*key);
            old_key = key;
        }

        signal_in = getinput();
        while(phase < (float)ratio) {
            k = (int)phase;    /* pointer to poly phase values */
            sendout(fir_filter_no_update(signal_in,h[k],lsize,hist));
            phase += dec;
        }
        phase -= ratio;
        fir_update_history(signal_in,lsize,hist);
    }
}

/* Use att to get beta (for Kaiser window function) and nfilt (always odd
    valued and = 2*npair +1) using Kaiser's empirical formulas.          */
long int filter_length(float att,float deltaf,float *beta)
{
    long int npair;
    *beta = 0.0;      /* value of beta if att < 21 */
    if(att >= 50.0) *beta = .1102 * (att - 8.71);
    if (att < 50.0 & att >= 21.0)
        *beta = .5842 * pow( (att-21.0), 0.4) + .07886 * (att - 21.0);
    npair = (long int)( (att - 8.0) / (28.72 * deltaf) );
    return(2*npair + 1);
}

/* Compute Bessel function Izero(y) using a series approximation */
float izero(float y){
    float s=1.0, ds=1.0, d=0.0;
    do {
```

LISTING 5.13 *(Continued)*

```
    d = d + 2;
    ds = ds * (y*y)/(d*d);
    s = s + ds;
} while( ds > 1E-7 * s);
return(s);
```

LISTING 5.13 *(Continued)*

```
/* run the fir filter and do not update the history array */

float fir_filter_no_update(float input,float *coef,int n,float *history)
{
    int i;
    float *hist_ptr,*coef_ptr;
    float output;

    hist_ptr = history;
    coef_ptr = coef + n - 1;          /* point to last coef */

/* form output accumulation */
    output = *hist_ptr++ * (*coef_ptr--);
    for(i = 2 ; i < n ; i++) {
        output += (*hist_ptr++) * (*coef_ptr--);
    }
    output += input * (*coef_ptr);      /* input tap */

    return(output);
}

/* update the fir_filter history array */

void fir_update_history(float input,int n,float *history)
{
    int i;
    float *hist_ptr,*hist1_ptr;

    hist_ptr = history;
    hist1_ptr = hist_ptr;                /* use for history update */
    hist_ptr++;

    for(i = 2 ; i < n ; i++) {
        *hist1_ptr++ = *hist_ptr++;      /* update history array */
    }
    *hist1_ptr = input;                  /* last history */
}
```

LISTING 5.14 Functions **fir_filter_no_update** and **fir_filter_update_history** used by program PSHIFT.C.

5.4.3 Music Synthesis

Music synthesis is a natural DSP application because no input signal or A/D converter is required. Music synthesis typically requires that many different sounds be generated at the same time and mixed together to form a chord or multiple instrument sounds (see Moorer, 1977). Each different sound produced from a synthesis is referred to as a voice. The duration and starting point for each voice must be independently controlled. Listing 5.15 shows the program MUSIC.C, which plays a sound with up to 6 voices at the same time. It uses the function **note** (see Listing 5.16) to generate samples from a second order IIR oscillator using the same method as discussed in section 4.5.1 in chapter 4. The envelope of each note is specified using break points. The array **trel** gives the relative times when the amplitude should change to the values specified in array **amps**. The envelope will grow and decay to reach the amplitude values at each time specified based on the calculated first-order constants stored in the **rates** array. The frequency of the second order oscillator in **note** is specified in terms of the semitone note number **key**. A **key** value of 69 will give 440 Hz, which is the musical note A above middle C.

```
#include <stdlib.h>
#include <math.h>
#include "rtdspc.h"
#include "song.h"          /* song[108][7] array */

/* 6 Voice Music Generator */

typedef struct {
    int key,t,cindex;
    float cw,a,b;
    float y1,y0;
} NOTE_STATE;

#define MAX_VOICES 6

    float note(NOTE_STATE *,int *,float *);

void main()
{
    long int n,t,told;
    int vnum,v,key;
    float ampold;
    register long int i,endi;
    register float sig_out;
```

LISTING 5.15 Program MUSIC.C, which illustrates music synthesis by playing a 6-voice song. *(Continued)*

```
    static NOTE_STATE notes[MAX_VOICES*SONG_LENGTH];

    static float trel[5] = {    0.1 ,    0.2,    0.7,  1.0, 0.0 };
    static float amps[5] = { 3000.0 , 5000.0, 4000.0, 10.0, 0.0 };
    static float rates[10];
    static int tbreaks[10];

    for(n = 0 ; n < SONG_LENGTH ; n++) {

/* number of samples per note */
        endi = 6*song[n][0];

/* calculate the rates required to get the desired amps */
        i = 0;
        told = 0;
        ampold = 1.0;       /* always starts at unity */
        while(amps[i] > 1.0) {
            t = trel[i]*endi;
            rates[i] = exp(log(amps[i]/ampold)/(t-told));
            ampold = amps[i];
            tbreaks[i] = told = t;
            i++;
        }

/* set the key numbers for all voices to be played (vnum is how many) */
        for(v = 0 ; v < MAX_VOICES ; v++) {
            key = song[n][v+1];
            if(!key) break;
            notes[v].key = key;
            notes[v].t = 0;
        }
        vnum = v;

        for(i = 0 ; i < endi ; i++) {
            sig_out = 0.0;
            for(v = 0 ; v < vnum ; v++) {
                sig_out += note(&notes[v],tbreaks,rates);
            }
            sendout(sig_out);
        }
    }
    flush();
}
```

LISTING 5.15 *(Continued)*


```
#include <stdlib.h>
#include <math.h>
#include "rtdspc.h"

/* Function to generate samples from a second order oscillator */

/* key constant is 1/12 */
#define KEY_CONSTANT 0.083333333333333

/* this sets the A above middle C reference frequency of 440 Hz */
#define TWO_PI_DIV_FS_440  (880.0 * PI / SAMPLE_RATE)

/* cw is the cosine constant required for fast changes of envelope */
/* a and b are the coefficients for the difference equation */
/* y1 and y0 are the history values */
/* t is time index for this note */
/* cindex is the index into rate and tbreak arrays (reset when t=0) */

typedef struct {
    int key,t,cindex;
    float cw,a,b;
    float y1,y0;
} NOTE_STATE;

/*
key:
    semi-tone pitch to generate,
    number 69 will give A above middle C at 440 Hz.
rate_array:
    rate constants determines decay or rise of envelope (close to 1)
tbreak_array:
    determines time index when to change rate
*/

/* NOTE_STATE structure, time break point array, rate parameter array */

float note(NOTE_STATE *s,int *tbreak_array,float *rate_array)
{
    register int ti,ci;
    float wosc,rate,out;

    ti = s->t;
/* t=0 re-start case, set state variables */
    if(!ti) {
        wosc = TWO_PI_DIV_FS_440 * pow(2.0,(s->key-69) * KEY_CONSTANT);
```

LISTING 5.16 Function **note(state,tbreak_array,rate_array)** generates the samples for each note in the MUSIC.C program. *(Continued)*

```
        s->cw = 2.0 * cos(wosc);
        rate = rate_array[0];
        s->a = s->cw * rate;      /* rate change */
        s->b = -rate * rate;
        s->y0 = 0.0;
        out = rate*sin(wosc);
        s->cindex = 0;
    }
    else {
        ci = s->cindex;
    /* rate change case */
        if(ti == tbreak_array[ci]) {
            rate = rate_array[++ci];
            s->a = s->cw * rate;
            s->b = -rate * rate;
            s->cindex = ci;
        }

    /* make new sample */
        out = s->a * s->y1 + s->b * s->y0;
        s->y0 = s->y1;
    }
    s->y1 = out;
    s->t = ++ti;
    return(out);
}
```

LISTING 5.16 *(Continued)*

APTIVE FILTER APPLICATIONS

A signal can be effectively improved or enhanced using adaptive methods, if the signal frequency content is narrow compared to the bandwidth and the frequency content changes with time. If the frequency content does not change with time, a simple matched filter will usually work better with less complexity. The basic LMS algorithm is illustrated in the next section. Section 5.5.2 illustrates a method that can be used to estimate the changing frequency of a signal using an adaptive LMS algorithm.

5.5.1 LMS Signal Enhancement

Figure 5.13 shows the block diagram of an LMS adaptive signal enhancement that will be used to illustrate the basic LMS algorithm. This algorithm was described in section 1.7.2 in chapter 1. The input signal is a sine wave with added white noise. The adaptive LMS

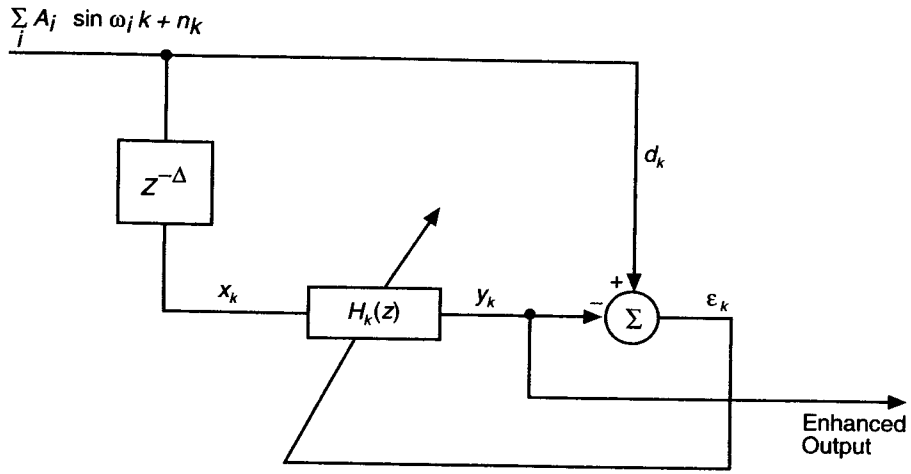

FIGURE 5.13 Block diagram of LMS adaptive signal enhancement.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "rtdspc.h"

#define N 351
#define L 20            /* filter order, (length L+1) */

/* set convergence parameter */
    float mu = 0.01;

void main()
{
    float lms(float,float,float *,int,float,float);
    static float d[N],b[21];
    float signal_amp,noise_amp,arg,x,y;
    int k;

/* create signal plus noise */
    signal_amp = sqrt(2.0);
    noise_amp = 0.2*sqrt(12.0);
    arg = 2.0*PI/20.0;
```

LISTING 5.17 Program LMS.C which illustrates signal-to-noise enhancement using the LMS algorithm. *(Continued)*

```
        for(k = 0 ; k < N ; k++) {
            d[k] = signal_amp*sin(arg*k) + noise_amp*gaussian();
        }

    /* scale based on L */
        mu = 2.0*mu/(L+1);

        x = 0.0;
        for(k = 0 ; k < N ; k++) {
            sendout(lms(x,d[k],b,L,mu,0.01));
    /* delay x one sample */
            x = d[k];
        }
    }
```

LISTING 5.17 (*Continued*)

```
/*
      function lms(x,d,b,l,mu,alpha)

Implements NLMS Algorithm b(k+1)=b(k)+2*mu*e*x(k)/((l+1)*sig)

x      = input data
d      = desired signal
b[0:l] = Adaptive coefficients of lth order fir filter
l      = order of filter (> 1)
mu     = Convergence parameter (0.0 to 1.0)
alpha  = Forgetting factor   sig(k)=alpha*(x(k)**2)+(1-alpha)*sig(k-1)
         (>= 0.0 and < 1.0)

returns the filter output
*/

float lms(float x,float d,float *b,int l,
                  float mu,float alpha)
{
    int ll;
    float e,mu_e,lms_const,y;
    static float px[51];       /* max L = 50 */
    static float sigma = 2.0; /* start at 2 and update internally */

    px[0]=x;
```

LISTING 5.18 Function **lms(x,d,b,l,mu,alpha)** implements the LMS algorithm. (*Continued*)


```
/* calculate filter output */
    y=b[0]*px[0];
    for(ll = 1 ; ll <= l ; ll++)
        y=y+b[ll]*px[ll];

/* error signal */
    e=d-y;

/* update sigma */
    sigma=alpha*(px[0]*px[0])+(1-alpha)*sigma;
    mu_e=mu*e/sigma;

/* update coefficients */
    for(ll = 0 ; ll <= l ; ll++)
        b[ll]=b[ll]+mu_e*px[ll];
/* update history */
    for(ll = l ; ll >= 1 ; ll--)
        px[ll]=px[ll-1];

    return(y);
}
```

LISTING 5.18 (*Continued*)

algorithm (see Listings 5.17 and 5.18) is a 21 tap (20th order) FIR filter where the filter coefficients are updated with each sample. The desired response in this case is the noisy signal and the input to the filter is a delayed version of the input signal. The delay (Δ) is selected so that the noise components of d_k and x_k are uncorrelated (a one-sample delay works well for sine waves and white noise).

The convergence parameter **mu** is the only input to the program. Although many researchers have attempted to determine the best value for **mu**, no universal solution has been found. If **mu** is too small, the system may not converge rapidly to a signal, as is illustrated in Figure 5.14. The adaptive system is moving from no signal (all coefficients are zero) to an enhanced signal. This takes approximately 300 samples in Figure 5.14b with **mu** = 0.01 and approximately 30 samples in Figure 5.14c with **mu** = 0.1.

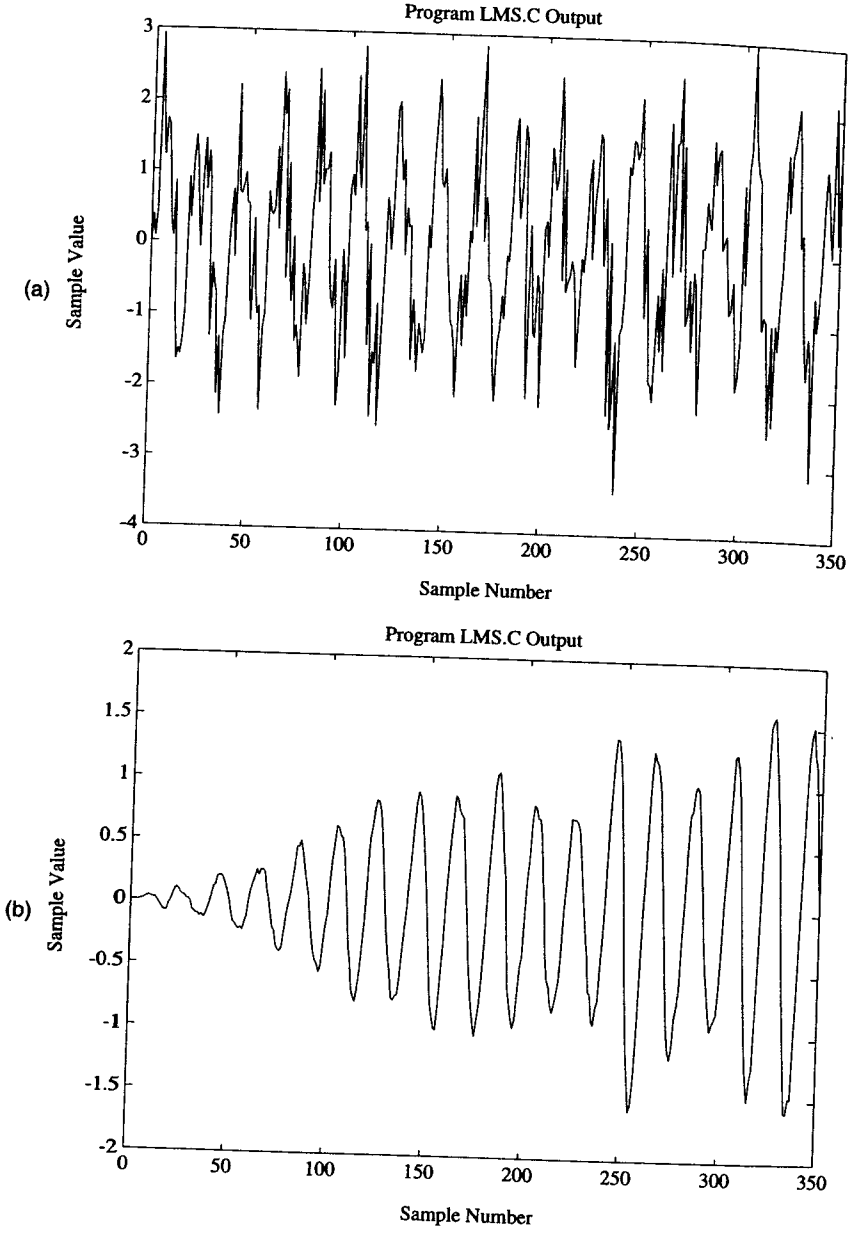

FIGURE 5.14 (a) Original noisy signal used in program LMS.C. (b) Enhanced signal obtained from program LMS.C with μ = 0.01.

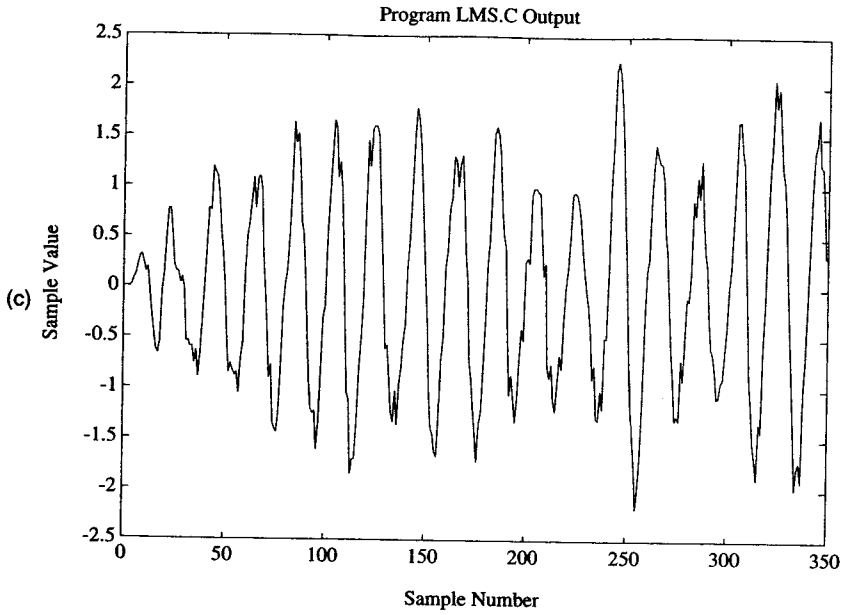

FIGURE 5.14 (c) Enhanced signal obtained from program LMS.C with μ = 0.1. *(Continued)*

5.5.2 Frequency Tracking with Noise

Listing 5.19 shows the INSTF.C program, which uses the **`lms`** function to determine instantaneous frequency estimates. Instead of using the output of the adaptive filter as illustrated in the last section, the INSTF program uses the filter coefficients to estimate the frequency content of the signal. A 1024-point FFT is used to determine the frequency response of the adaptive filter every 100 input samples. The same peak location finding algorithm as used in section 5.1.2 is used to determine the interpolated peak frequency response of the adaptive filter. Note that because the filter coefficients are real, only the first half of the FFT output is used for the peak search.

Figure 5.15 shows the output of the INSTF program when the 100,000 samples from the OSC.C program (see section 4.5.1 of chapter 4) are provided as an input. Figure 5.15(a) shows the result without added noise, and Figure 5.15(b) shows the result when white Gaussian noise (standard deviation = 100) is added to the signal from the OSC program. Listing 5.19 shows how the INSTF program was used to add the noise to the input signal using the **`gaussian()`** function. Note the positive bias in both results due to the finite length (128 in this example) of the adaptive FIR filter. Also, in Figure 5.15(b) the first few estimates are off scale because of the low signal level in the beginning portion of the waveform generated by the OSC program (the noise dominates in the first 10 estimates).

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "rtdspc.h"

/* LMS Instantaneous Frequency Estimation Program */

#define L 127       /* filter order, L+1 coefficients */
#define LMAX 200    /* max filter order, L+1 coefficients */
#define NEST 100    /* estimate decimation ratio in output */

/* FFT length must be a power of 2 */
#define FFT_LENGTH 1024
#define M 10           /* must be log2(FFT_LENGTH) */

/* set convergence parameter */
    float mu = 0.01;

void main()
{
    float lms(float,float,float *,int,float,float);
    static float b[LMAX];
    static COMPLEX samp[FFT_LENGTH];
    static float   mag[FFT_LENGTH];
    float x,d,tempflt,p1,p2;
    int i,j,k;

/* scale based on L */
    mu = 2.0*mu/(L+1);

    x = 0.0;
    for(;;) {
        for(i = 0 ; i < NEST ; i++) {
/* add noise to input for this example */
            x = getinput() + 100.0*gaussian();
            lms(x,d,b,L,mu,0.01);
/* delay d one sample */
            d = x;
        }
```

LISTING 5.19 Program INSTF.C. which uses function **lms(x,d,b,l,mu,alpha)** to implement the LMS frequency tracking algorithm. (*Continued*)


```
/* copy L+1 coefficients */
        for(i = 0 ; i <= L ; i++) {
            samp[i].real = b[i];
            samp[i].imag = 0.0;
        }

/* zero pad */
        for( ; i < FFT_LENGTH ; i++) {
            samp[i].real = 0.0;
            samp[i].imag = 0.0;
        }

        fft(samp,M);

        for(j = 0 ; j < FFT_LENGTH/2 ; j++) {
            tempflt  = samp[j].real * samp[j].real;
            tempflt += samp[j].imag * samp[j].imag;
            mag[j] = tempflt;
        }
/* find the biggest magnitude spectral bin and output */
        tempflt = mag[0];
        i=0;
        for(j = 1 ; j < FFT_LENGTH/2 ; j++) {
            if(mag[j] > tempflt) {
                tempflt = mag[j];
                i=j;
            }
        }
/* interpolate the peak loacation */
        if(i == 0) {
            p1 = p2 = 0.0;
        }
        else {
            p1 = mag[i] - mag[i-1];
            p2 = mag[i] - mag[i+1];
        }
        sendout(((float)i + 0.5*((p1-p2)/(p1+p2+1e-30)))/FFT_LENGTH);
    }
}
```

LISTING 5.19 (*Continued*)

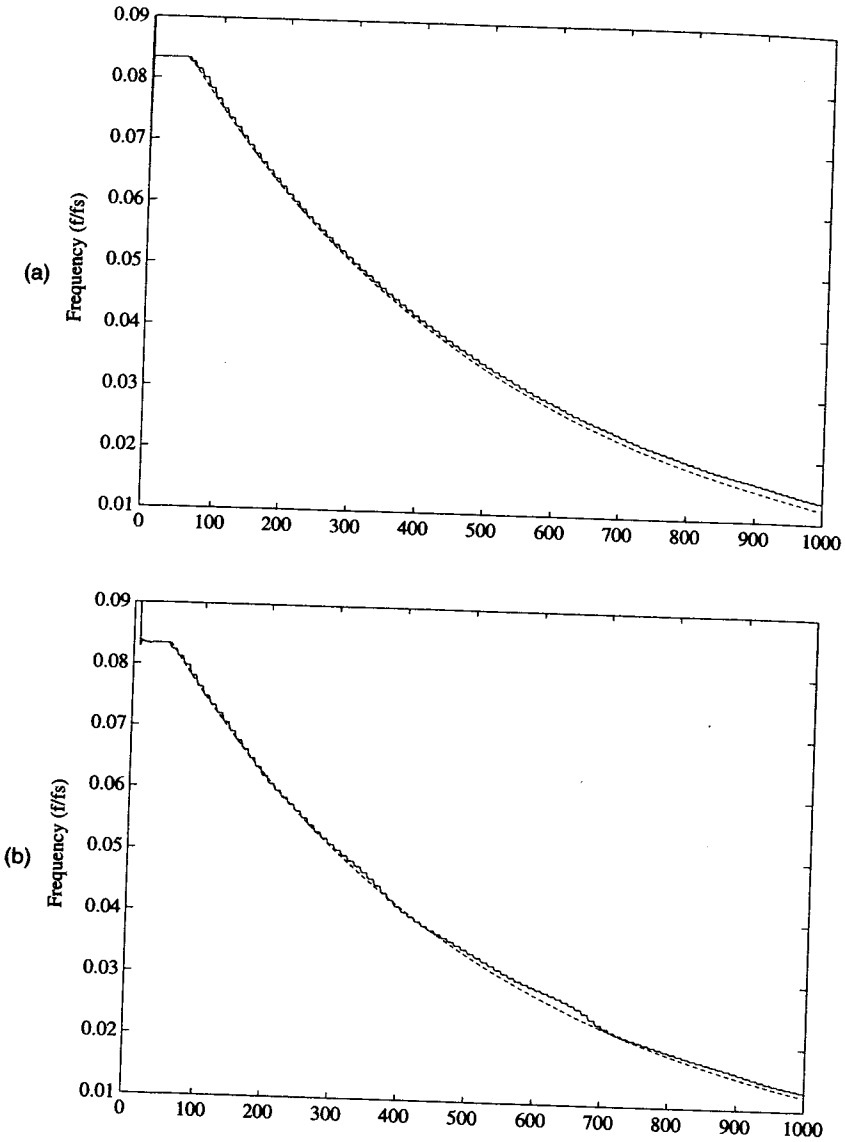

FIGURE 5.15 **(a)** Frequency estimates obtained from program INSTF.C (solid line) with input from OSC.C and correct frequencies (dashed line) generated by OSC.C. The INSTF.C program shown in Listing 5.19 was modified to not add noise to the input signal. **(b)** Frequency estimates obtained from program INSTF.C (solid line) with input from OSC.C and correct frequencies (dashed line) generated by OSC.C. Noise with a standard deviation of 100 was added to the signal from the OSC program.

5.6 REFERENCES

MOORER, J. (August 1977). Signal Processing Aspects of Computer Music: A Survey. *Proceedings of IEEE, 65,* (8).

ALLES, H. (April 1980). Music Synthesis Using Real Time Digital Techniques. *Proceedings of the IEEE, 68,* (4).

SMITH J. and GOSSETT, P. (1984). A Flexible Sample Rate Conversion Method. *Proceedings of ICASSP.*

CROCHIERE, R. and RABINER, L. (March 1981). Interpolation and Decimation of Digital Signals—A Tutorial Review. *Proceedings of the IEEE, 69,* 300–331.

SKOLNIK, M. (1980). *Introduction to Radar Systems,* (2nd ed.). New York: McGraw-Hill.

General Aspects of Digital Transmission Systems (Nov. 1988). Terminal Equipments Recommendations G.700–G.795. International Telegraph and Telephone Consultative Committee (CCITT) 9th Plenary Assembly, Melbourne.

APPENDIX

DSP FUNCTION LIBRARY AND PROGRAMS

The enclosed disk is an IBM-PC compatible high-density disk (1.44 MBytes capacity) and contains four directories called PC, ADSP21K, DSP32C, and C30 for the specific programs that have been compiled and tested for the four platforms discussed in this book. Each directory contains a file called READ.ME, which provides additional information about the software. A short description of the platforms used in testing associated with each directory is as follows:

Directory Name	Platform used to Compile and Test Programs	Available MIPs	Sampling Rate (kHz)
PC	General Purpose IBM-PC or workstation (ANSI C)	Not Real-time	Any
ADSP21K	Analog Devices EZ-LAB ADSP-21020/ADSP-21060 (version 3.1 compiler software)	25	32
DSP32C	CAC AC4-A0 Board with DBDADA-16 (version 1.6.1 compiler software)	12.5	16
C30	Domain Technologies DSPCard-C31 (version 4.50 compiler software)	16.5	16

The following table is a program list of the C programs and functions described in detail in chapters 3, 4, and 5. The first column gives the section number in the text where the program is described and then a short description of the program. The remaining columns give the filenames of the four different versions of the source code for the four different platforms. Note that the files from each platform are in different directories as shown in the previous table.

	PC filename (*.c)	210X0 filename (*.c)	DSP32C filename (*.c)	320C30 filename (*.c)
3.3.3 1024-Point FFT Test Function	fft1k	fftn	fft1k	fft1k
3.4.2 Interrupt-Driven Output example	NA	intout	NA	NA
4.1.1 FIR Filter Function (fir_filter)	filter	filter	filter	filter
4.1.2 FIR Filter Coefficient by Kaiser Window	ksrfir	NA	NA	NA
4.1.2 FIR Filter Coefficients by Parks-McClellan	remez	NA	NA	NA
4.1.3 IIR Filter Function (iir_filter)	filter	filter	filter	filter
4.1.4 Real-Time getinput Function (ASCII text for PC)	getsend	getinput	getinput	send_c30
4.1.4 Real-Time getinput Function (WAV file format)	getwav	NA	NA	NA
4.1.4 Real-Time sendout Function (ASCII text for PC)	getsend	sendout	sendout	send_c30
4.1.4 Real-Time sendout Function (WAV file format)	sendwav	NA	NA	NA
4.2.1 Gaussian Noise Generation Function	filter	filter	filter	filter
4.2.2 Signal-to-Noise Ratio Improvement	mkgwn	mkgwn	mkgwn	mkgwn
4.3.3 Sample Rate Conversion example	interp3	NA	NA	NA
4.4.1 Fast Convolution Using FFT Methods	rfast	rfast21	rfast32	rfast30
4.4.2 Interpolation Using the FFT	intfft2	NA	NA	NA
4.5.1 IIR Filters as Oscillators	osc	osc	osc	osc
4.5.2 Table-Generated Waveforms	wavetab	wavetab	wavetab	wavetab
5.1.1 Speech Spectrum Analysis	rtpse	NA	NA	NA
5.1.2 Doppler Radar Processing	radproc	NA	NA	NA
5.2.1 ARMA Modeling of Signals	arma	NA	NA	NA
5.2.2 AR Frequency Estimation	arfreq	NA	NA	NA
5.3.1 Speech Compression	mulaw	mulaw	mulaw	mulaw
5.3.2 ADPCM (G.722 fixed-point)	g722	g722_21k	NA	g722c3
5.3.2 ADPCM (G.722 floating-point)	NA	g722_21f	g722_32c	g722c3f
5.4.1 Equalization and Noise Removal	equaliz	equaliz	equaliz	equaliz
5.4.2 Pitch-Shifting	pshift	pshift	pshift	pshift
5.4.3 Music Synthesis	music	mu21k	mu32c	muc3
5.5.1 LMS Signal Enhancement	lms	NA	NA	NA
5.5.2 Frequency Tracking with Noise	instf	NA	NA	NA

Note: "NA" refers to programs that are not applicable to a particular hardware platform.

Make files (with an extension .MAK) are also included on the disk for each platform. If the user does not have a make utility availible, PC batch files (with an extension .BAT) are also included with the same name as the make file. The following table is a make file list for many of the C programs described in detail in Chapters 3, 4 and 5:

	PC filename (*.mak)	210X0 filename (*.mak)	DSP32C filename (*.mak)	320C30 filename (*.mak)
3.4.2 Interrupt-Driven Output Example	NA	iout21k	NA	NA
4.2.2 Signal-to-Noise Ratio Improvement	mkgwn	mkgwn	mkgwn	mkgwn
4.3.3 Sample Rate Conversion Example	interp3	NA	NA	NA
4.4.1 Fast Convolution Using FFT Methods	rfast	rf21k	rf32	rfc30
4.4.2 Interpolation Using the FFT	intfft2	NA	NA	NA
4.5.1 IIR Filters as Oscillators	osc	osc21k	osc	osc
4.5.2 Table Generated Waveforms	wavetab	wavetab	wavetab	wavetab
5.1.1 Speech Spectrum Analysis	rtpse	NA	NA	NA
5.1.2 Doppler Radar Processing	radproc	NA	NA	NA
5.2.1 ARMA Modeling of Signals	arma	NA	NA	NA
5.2.2 AR Frequency Estimation	arfreq	arfreq	arfreq	arfreq
5.3.1 Speech Compression	mulaw	mulaw	mulaw	mulaw
5.3.2 ADPCM (G.722 fixed-point)	g722	g722_21k	NA	g722c3
5.3.2 ADPCM (G.722 floating-point)	NA	g722_21f	g722_32c	g722c3f
5.4.1 Equalization and Noise Removal	eqpc	eq	eq	eq
5.4.2 Pitch Shifting	ps	ps	ps	ps
5.4.3 Music Synthesis	music	mu21k	mu32c	muc3
5.5.1 LMS Signal Enhancement	lms	NA	NA	NA
5.5.2 Frequency Tracking with Noise	instf	instf	instf	instf

Note: "NA" refers to programs that are not applicable to a particular platform.

INDEX

A

A/D converter, 3, 54, 125, 132, 225
accumulation, 136, 223
adaptive, 46, 48, 50, 51, 52, 111, 112, 186, 193, 194, 195, 196, 197, 198, 202, 205, 206, 209, 211, 216, 217, 228, 229, 231, 233
adaptive filters, 1, 46, 111
address of operator, 77, 87
ADPCM, 202, 204, 215, 217
ADSP-21020, 99, 104, 105, 116, 118, 119, 129, 130
ADSP-21060, 99, 104, 107
ADSP-210XX, 104, 107, 112, 114, 121, 127
aliases, 74, 75
aliasing, 7, 162, 163
analog filters, 21
analog-to-digital converter, 41, 42, 132
AR Processes, 43
architecture, 92, 99, 102, 107, 108, 113, 116, 130, 131
ARFREQ.C, 198, 199, 201
arithmetic operators, 59, 60
ARMA filters, 17, 18
ARMA.C, 193, 194, 195, 196, 197, 198
array index, 66, 78
arrays, 54, 56, 58, 59, 78, 81, 82, 84, 88, 114, 128, 146, 167, 179, 219, 226
assembly language, 74, 92, 99, 102, 108, 111, 113, 114, 115, 116, 117, 118, 120, 121, 125, 127
Assembly-C Language Interfaces, 118, 120
Assignment Operators, 59
attenuation, 22, 24, 136, 137, 138, 141, 142, 147, 162, 163, 165, 166, 221
autocorrelation, 39, 42, 43, 44, 49, 111
automatic variables, 71
autoregressive (AR), 17, 44
average power, 188

B

bandpass filter, 138, 140, 218, 219
bandwidth, 33, 102, 113, 147, 160, 190, 200, 201, 218, 228
bilinear transform, 21, 147, 150
bit reversal, 122, 123
bitwise operators, 59, 60
Box-Muller method, 158
butterfly, 29

C

C preprocessor, 74, 87, 113
C Programming Pitfalls, 87
C++, 82, 97
calloc, 78, 79, 80, 83, 84, 89, 150, 171, 173, 177, 222
case statement, 65
cast operator, 79

causality, 10
circular convolution, 170
clipping, 33, 34
coefficient quantization, 21, 145
combined operators, 61
comments, 54, 92, 93, 94
complex conjugate, 19, 20, 91
complex conversion, 198
Complex Data, 90
complex numbers, 85, 87, 90, 91
complex signal, 190, 198, 200
compound statements, 64
compression, 33, 46, 200, 201, 202
conditional compilation, 74
conditional execution, 63, 95
constants, 20, 26, 42, 62, 71, 74, 90, 124, 225, 227
continue, 18, 66, 67, 68, 90, 96, 127
continuous time signals, 4
control structures, 63, 64, 66, 67, 95, 96
converter, 3, 41, 42, 54, 125, 132, 225
convolution, 9, 10, 18, 25, 134, 135, 165, 168, 170, 171, 172
cross correlation, 49

D

data structures, 53, 54, 55, 77
data types, 53, 56, 58, 80, 82, 90
debuggers, 117
decimation, 160, 162, 163, 164, 173, 182, 220, 221, 222, 234
declaring variables, 57
delay, 11, 13, 107, 132, 133, 134, 146, 165, 168
DFT, 18, 25, 26, 27, 28, 29, 30, 32, 44
difference equation, 10, 17, 22, 23, 133, 226
differentiator, 138
digital filters, 1, 2, 17, 19, 21, 52, 163, 184
Dirac delta function, 3
direct form, 21, 145, 146, 148
discrete Fourier transform, 1, 3, 18, 25, 26, 44, 52
discrete Time Signals, 5
disk files, 151
do-while loop, 66, 96
documentation, 94
Doppler, 190, 191
double precision, 57
downsampling, 160
DSP programs, 53, 59, 92, 93, 114
DSP3210, 99, 100, 102, 104, 112, 120
DSP32C, 99, 100, 101, 102, 111, 112, 114, 115, 117, 118, 120, 129, 130, 203
dynamic memory allocation, 77, 78

E

efficiency, 92, 93, 111, 113, 120, 121, 127, 128, 129, 135, 150
elliptic filter, 147, 149
enhancement, 160, 228, 229
EQUALIZ.C, 218, 219, 220
equiripple, 134
execution time, 65, 80, 89, 92, 93, 123, 124, 125
expected value, 37, 39, 42, 43
exponential, 32, 128, 178
expression, 8, 19, 37, 49, 59, 60, 61, 62, 63, 64, 65, 66, 67, 70, 86, 87, 91, 95, 128, 138, 165
extensibility, 92, 93
extern, 71, 72, 73, 155, 203

F

fast convolution, 134, 168, 170, 171, 172
fast filtering, 168
fast Fourier transform, 26, 28, 52, 160, 184
filter design, 18, 19, 22, 134, 136, 138, 140, 141, 145, 147, 184
filter functions, 221
filter order, 229, 234
filter specifications, 23, 24, 137
filter structures, 44, 45, 46
FILTER.C, 134, 158
FILTER.H, 141, 150, 162, 171
finite impulse response (FIR), 17, 133, 140
FIR filter, 18, 20, 22, 23, 50, 111, 113, 121, 128, 129, 134, 136, 138, 142, 144, 145, 147, 151, 160, 162, 165, 168, 171, 176, 198, 199, 221, 231, 233
fir_filter, 134, 135, 136, 151, 162, 167, 168, 199, 221, 222, 223, 224
floating point, 203
flush, 154, 155, 156, 157, 180, 183, 226
fopen, 152, 153, 155
for loop, 54, 67, 72, 76, 87, 91, 94, 95, 96, 135
Fourier transform, 1, 3, 4, 14, 15, 17, 18, 19, 25, 26, 28, 31, 44, 52, 160, 170, 184
free, 67, 78, 79, 80, 84, 93, 112, 125, 173
frequency domain, 7, 15, 16, 17, 18, 23, 24, 25, 30, 32, 44, 132, 168, 170, 176
frequency estimation, 198, 234
frequency response, 15, 17, 18, 19, 20, 21, 22, 23, 134, 138, 142, 149, 166, 176, 218, 220, 233
frequency tracking, 186, 233, 235
function call, 60, 86, 127, 128
function prototype, 73

G

G.711, 201
G.722, 202, 215, 216
G722.C, 204, 208, 211, 214, 215, 216, 217
Gaussian, 37, 39, 45, 47, 158, 159, 160, 162, 193, 233
GETSEND.C, 152
GETWAV.C, 154
global variables, 93, 114, 118, 125
goto, 67, 68, 95, 96

H

hamming window, 188
Harris, 28, 52
highpass filter, 191
Hilbert transform, 198, 199, 200

I

IBM PC, 58, 79, 88, 97
ideal lowpass filter, 137, 165
identifier, 56
if-else, 63, 64, 65, 66, 67, 68, 95
IIR filter design, 22, 145
IIR filters, 18, 21, 50, 111, 132, 134, 145, 178
iir_filter, 145, 146, 147, 148, 150, 151, 219, 239
impulse response, 9, 10, 17, 18, 21, 133, 140, 145, 170, 172, 178, 216
impulse sequence, 8, 9, 32
indexing, 67, 87
infinite loop, 67, 151
initialization, 59, 67, 82, 83, 91, 113, 124
input/output functions, 151
INSTF.C, 233, 235, 236
INTERP3.C, 167, 168
interpolation, 160, 163, 164, 165, 166, 167, 168, 170, 176, 177, 179, 193, 203, 220, 221
interrupts, 102, 107, 121, 125, 126
INTFFT2.C, 176, 177
INTOUT.C, 126
inverse DFT, 18
inverse FFT, 170, 171, 172, 176, 177
iteration, 67, 90, 91, 95, 128, 140, 174

K

Kaiser window, 18, 134, 137, 138, 141, 142, 143, 144, 165, 221, 222, 223
keyboard, 98, 138
keywords, 56, 66, 75, 76, 90, 91, 114
KSRFIR.C, 138

L

label, 9, 65, 68, 69
library functions, 87, 113
linear interpolation, 179, 203
linear operators, 1, 2, 11, 17, 18, 25, 32
linear phase, 20, 21, 52, 133, 135, 140, 142, 147, 162, 164, 184
linear time invariant operators, 1, 8
LMS, 50, 51, 193, 196, 228, 229, 231, 232, 233, 234, 235
local variables, 55, 70
log, 149, 158, 159, 180, 189, 226
$\log_2$, 29, 170, 171, 176, 188, 191, 234
logical operators, 59, 61, 91
loops, 46, 61, 66, 67, 68, 95, 96, 107, 108, 125, 128, 129
lowpass filter, 23, 24, 137, 138, 141, 147, 148, 149, 162, 163, 164, 165

M

macros, 74, 75, 76, 82, 85, 90, 120, 121, 128
magnitude, 23, 24, 48, 141, 149, 192, 235
maintainability, 92, 93
malloc, 78, 79, 80
matrices, 80, 81
matrix operations, 90, 111
mean, 37, 39, 40, 41, 42, 43, 44, 46, 48, 50, 51, 158, 159, 191, 193, 198
mean squared error, 40, 41, 42, 48, 50
mean value, 40, 43, 193
memory allocation functions, 79
memory map, 113
memory mapped, 100, 124
MKGWN.C, 162
modeling, 21, 43, 46, 48, 186, 193, 194, 198
modulus operator, 60
moment, 39, 43
moving average (MA), 44
MULAW.C, 201, 202
multiprocessor, 108, 130
music, 132, 178, 182, 186, 201, 202, 218, 225, 226, 228

N

noise, 21, 28, 35, 42, 43, 44, 45, 46, 47, 50, 51, 98, 132, 145, 158, 160, 162, 163, 186, 187, 193, 198, 201, 218, 228, 229, 231, 233, 234, 236
nonlinear, 1, 2, 32, 33, 164, 216
normal equation, 49
null pointer, 80

numerical C, 87, 90, 91, 113, 121, 124
Nyquist rate, 176, 187

O

operator precedence, 62
optimal filter, 46
OSC.C, 181, 233, 236
oscillators, 178
oversized function, 93

P

parameters, 1, 24, 43, 46, 51, 74, 75, 76, 121, 126, 138, 186, 188, 193
parametric, 186, 193, 198
pass-by-address, 87
periodic, 5, 8, 25, 28, 29, 32, 178, 183
periodogram, 186
phase response, 22, 23
physical input/output, 124
pitch-shifting, 220, 223
pointer operators, 77
pointers, 53, 56, 60, 69, 71, 72, 73, 77, 78, 80, 81, 82, 84, 86, 88, 90, 128, 135, 150
pole-zero plot, 149
poles, 51, 146, 147, 149, 150, 178, 193, 194, 195, 216
polled input/output, 124
polling, 125
polynomial interpolation, 163
post increment, 78
power spectral estimation, 186, 187, 188, 189, 191
power spectrum, 27, 28, 44, 125, 158, 163, 186, 189
precedence, 62
preprocessor directives, 74, 75
privacy, 71
probability, 2, 35, 36, 37, 39, 40, 41, 42, 43, 52, 185
program control, 53, 54, 63, 65, 68
program jumps, 67, 69
programming style, 92, 95, 97
promotion, 63
properties of the DFT, 26
PSHIFT.C, 220, 223, 224

Q

quantization, 3, 21, 32, 33, 40, 41, 42, 99, 145, 201, 207, 218

R

radar, 46, 186, 190, 191, 192, 193, 198
RADPROC.C, 191
rand, 158, 159
random number generator, 158
random processes, 2, 35, 42, 43
random variables, 36, 37, 39, 42, 43, 52, 158, 159, 185
realloc, 78, 79, 80
rectangular window, 138
referencing Structures, 82
register, 71, 72, 107, 108, 111, 115, 118, 120, 121, 128, 129, 182
reliability, 92, 93
Remez exchange algorithm, 18, 134, 140
REMEZ.C, 19, 134, 138
RFAST.C, 171
RIFF, 151, 153, 155, 156
RTPSE.C, 188

S

s-plane, 147, 149
sample rate conversion, 112, 160, 167
sampled signal, 3, 4, 7, 132, 160
sampling function, 3, 4, 5
sampling rate, 15, 24, 99, 127, 151, 156, 160, 162, 163, 201, 202, 204, 218, 219, 220
scaling, 99
scanf, 66, 87, 88, 89
scope, 71, 72, 73
seed, 107, 194
SENDWAV.C, 157
sequences, 1, 2, 10, 11, 16, 17, 25, 27, 28, 40, 158, 168, 170, 172
serial, 100, 102, 107, 108, 124
shift operator, 99, 203
signal enhancement, 228, 229
simulation, 158
simulator, 102, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120
sinc function, 137, 138, 165
single-line conditional expressions, 65
singular, 49
sinusoid, 42
sizeof, 79, 80, 83, 84, 89, 153, 154, 155, 156, 171, 173, 177, 222
software quality, 93
software tools, 111, 117
source level debuggers, 117
spectral analysis, 30, 168, 198
spectral density, 42, 44
spectral estimation, 186, 187, 188, 189, 191, 193
speech compression, 46, 200, 201, 202
speech signal, 200, 201, 202, 203

srand, 194, 195
stack, 70, 71, 78, 80, 107, 118, 121
standard deviation, 55, 160, 162, 193, 233, 236
stationary, 42, 43, 190, 191
statistics, 43, 44, 49, 54
status, 68, 69, 107, 125
stopband, 22, 24, 134, 136, 137, 138, 141, 142, 147, 163, 165, 166
storage class, 71, 72, 82
stream, 100, 151
structured programming, 63, 95, 96
structures, 21, 44, 45, 46, 53, 54, 55, 63, 64, 66, 67, 68, 77, 82, 84, 85, 86, 90, 95, 96, 134
superposition, 4
switch, 64, 65, 67, 68, 95, 142, 143
synthesis, 98, 132, 178, 184, 225, 226
system design, 114, 124

T

table generated waveforms, 179
taps, 100, 133, 216
thermal noise, 158
tightly bound, 93
time domain, 7, 13, 15, 17, 25, 30, 40, 44, 132, 168, 170, 176
time invariant operators, 1, 8, 10
TMS320C30, 99, 100, 108, 109, 113, 116, 117, 120, 121, 129, 130
TMS320C40, 99, 100, 108, 110
transfer function, 13, 14, 19, 20, 21, 45, 48, 50, 145, 147, 193
transform domain, 13
transition band, 24, 137, 163, 165
truncation, 3, 32, 63, 107
two's complement, 60, 64
type conversion, 62
typedef, 83, 84, 85, 122, 225, 226
types of numbers, 56

U

unary minus, 60, 62
underscore, 56
unit circle, 16, 17, 51, 178
unsigned, 57, 58, 59, 61, 63, 90, 153
upsampling, 160
user interface, 116

V

variance, 37, 39, 40, 42, 44, 54, 55, 71, 73, 75, 158, 159

W

WAV file, 151, 153, 154, 155, 156, 157
waveform synthesis, 178
waveforms, 7, 178, 179, 186
WAVETAB.C, 179
white noise, 42, 45, 46, 160, 162, 193, 228, 231
Wiener filter, 46, 48, 49, 160
windowing, 27, 134, 164
windows, 28, 52

Z

z-plane, 16, 149
z-transform, 11, 12, 13, 14, 15, 16, 17, 21, 26
zero padding, 170

C ALGORITHMS FOR REAL-TIME DSP

PAUL M. EMBREE

Prentice Hall PTR
Upper Saddle River, NJ 07458