

# 8

## *Digital Signal Processing*

**Signal processing** deals with the enhancement, extraction, and representation of information for communication or analysis. Many different fields of engineering rely on signal processing technology. Acoustics, telephony, radio, television, seismology, and radar are some examples.

Initially, signal processing systems were implemented exclusively with analog hardware. However, recent advances in high-speed digital technology have made discrete signal processing systems more popular. Digital systems have an advantage over analog systems in that they can process signals with an extraordinary degree of precision. Unlike the resistive and capacitive networks of analog systems, digital systems can be built numerically with the simple operations of addition and multiplication.

**Digital signal processing** is a field of numerical mathematics that is concerned with the processing of discrete signals. This area of mathematics deals with the principles that underlie all digital systems.

### 8.1 BACKGROUND

The following sections present some concepts that are important to the understanding of digital signal processing (**DSP**). Although these sections present many DSP fundamentals, they are in no way intended to be a complete description of the field. It is recommended that the interested reader consult the excellent references cited at the end of this book.

### 8.1.1 The Sampling Theorem

Shannon's **sampling theorem** is of fundamental importance to digital signal processing. This theorem is as follows:

Suppose that  $x(t)$  is a continuous time signal that is bandlimited with a maximum frequency of  $f_M$ . Then  $x(t)$  can be uniquely represented by a sequence of time samples:

$$x(mT) = \dots, x(-2T), x(-T), x(0), x(T), x(2T), \dots$$

where  $T$  is the **sampling interval**, and  $1/T = 2f_M$  is called the **Nyquist rate**. The maximum frequency of the signal  $f_M$  is called the **Nyquist frequency**.

In essence, the sampling theorem establishes a minimum rate at which an analog signal must be sampled. This sampling frequency is  $F_s = 1/T$ . Sampling at any rate faster than the Nyquist rate (i. e., making  $T$  shorter than  $2f_M$ ) also produces a valid representation of  $x(t)$ .

In digital sampling, frequencies are often specified normalized to  $F_s$ . For example,  $F_s/2$ , the Nyquist frequency, is the highest frequency that can be represented and has a normalized value of  $1/2$ .

The above theorem is very important, since in practice all signals can be considered to be bandlimited over the frequency range of interest. For example, musical sounds with frequency components higher than 20 KHz cannot be heard by most humans. For listening purposes, the sound only needs to be sampled at the Nyquist rate  $2f_M = 40$  KHz.

### 8.1.2 The Discrete Fourier Transform

The discrete Fourier transform (**DFT**) plays an important role in digital signal analysis. The DFT serves as a gateway between the time representation and the frequency representation of a signal. The **forward DFT** transforms a time signal into the frequency domain and is defined as:

$$X(k) = [x(0) + x(1)e^{-j\omega k} + \dots x(m)e^{-jm\omega k} \dots + x(n-1)e^{-j(n-1)\omega k}]$$

for  $k = 0, 1, 2, \dots, n-1$

where  $\omega = 2 * \pi/n$  and  $\pi = 3.141592654 \dots$

The **inverse DFT** transforms the frequency representation of a signal back into the time domain and is defined as:

$$x(m) = [X(0) + X(1)e^{j\omega m} + \dots X(k)e^{jk\omega m} \dots + X(n-1)e^{j(n-1)\omega m}]/n$$

for  $m = 0, 1, 2, \dots, n-1$

where again  $\omega = 2 * \pi/n$  and  $\pi = 3.141592654 \dots$

The time sequence  $\{x(0), x(1), \dots, x(n-1)\}$  and the frequency sequence  $\{X(0), X(1), \dots, X(n-1)\}$  are Fourier transform pairs. It should be noted that both these sequences are equally spaced. The interval between the samples of the time sequence is  $T$ . The interval between the samples of the frequency sequence is  $1/(nT)$ .

### Example 1

Find the frequency resolution of a 128-point FFT for a sample rate of 100 Hz. The interval between the samples of the time sequence is:

$$T = 1/100 = 0.01 \text{ second}$$

This yields a frequency resolution of

$$1/(128 * T) = 0.78125 \text{ Hz}$$

Throughout this development, it is assumed that the Nyquist condition of the sampling theorem has been satisfied. The factor of  $T$ , found in some DFT definitions, is redundant and is eliminated here, since it in no way affects the above transforms.

### Example 2

The frequency composition of time domain signals can be analyzed with the DFT. The DFT of the complex sinusoid  $A * [\cos(\omega * k * i) + j * \sin(\omega * k * i)]$  consists of one frequency component at the  $k$ th harmonic. The DFT of the complex sinusoid

$$A * [\cos(\omega * k * i) - j * \sin(\omega * k * i)]$$

is also one frequency component at the  $(n - k)$ th harmonic  $k = 1, 2, \dots, n/2$  and  $i = 0, 1, 2, \dots, n - 1$ . Both components are of magnitude  $A * n$ .

**Two-Dimensional DFT.** The two-dimensional DFT is a simple extension of the one-dimensional formulas. The **forward 2-D discrete Fourier transform**  $X(k, l)$  for a square array of  $N \times N$  elements is

$$X(k, l) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m, n) e^{-j2\pi(km+ln)/N}; \quad k, l = 0, 1, \dots, N - 1$$

The **inverse 2-D discrete Fourier transform**  $x(m, n)$  is given by

$$x(m, n) = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} X(k, l) e^{j2\pi(km+ln)/N}$$

The two-dimensional DFT can be efficiently computed by decomposing the 2-D transform into separable 1-D row and column FFTs.

**See Functions:** `fft2d()`, `fft2d_r()`.

### 8.1.3 The Discrete Cosine Transform

The discrete cosine transform (DCT) has become a useful tool in image compression. The  $8 \times 8$  2-D DCT has been adopted by the JPEG committee as part of its standard for compressing still images. The DCT is closely related to the discrete Fourier transform.

**One-Dimensional Discrete Cosine Transform.** The Fourier transform of any real symmetric function can be represented by a cosine series. Fewer edge effects result if the cosine series implements a half-sample shift of the following form:

$$X(k) = \sqrt{\frac{2}{N}} a(k) \sum_{n=0}^{N-1} x(n) \cos \frac{(2n+1)k\pi}{2N}; \quad k = 0, 1, \dots, N-1$$

$$a(0) = 1/\sqrt{2} \quad \text{and} \quad a(k) = 1; \quad k \neq 0$$

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X(k) a(k) \cos \frac{(2n+1)k\pi}{2N}; \quad n = 0, 1, \dots, N-1$$

Note that, except for a constant scaling factor, the forward transform  $X(k)$  is identical to the inverse transform  $x(n)$ .

**Example**

If the real sinusoid  $A * \cos(\omega * k * (2 * i + 1))$  is used to generate the  $x$  vector, the transformed vector will consist of one frequency component at the  $k$ th harmonic, where  $k = 0, 1, 2, \dots, n-1$ ,  $i = 0, 1, 2, \dots, n-1$ ,  $\omega = \pi/(2 * n)$ , and  $\pi = 3.141592654 \dots$ . If  $k$  is greater than zero, the magnitude of the component is  $A(n/2)^{1/2}$ .

**See Function:** `dct()`.

**Two-Dimensional Discrete Cosine Transform.** Due to separability, the 2-D DCT can be calculated with consecutive 1-D row and column cosine transforms. However, a fast block matrix approach is more efficient [see P. M. Embree, et al. (1991)].

The 2-D DCT is of the form

$$X(k, l) = \frac{2}{N} a(k) a(l) \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m, n) \cos \frac{\pi k(2m+1)}{2N} \cos \frac{\pi l(2n+1)}{2N}$$

$$a(0) = 1/\sqrt{2}; \quad a(j) = 1; \quad j \neq 0$$

$$x(m, n) = \frac{2}{N} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} a(k) a(l) X(k, l) \cos \frac{\pi k(2m+1)}{2N} \cos \frac{\pi l(2n+1)}{2N}$$

Note that, as in the one-dimensional case, the forward transform  $X(k, l)$  is very similar to the inverse transform  $x(m, n)$ .

**Example**

If the real 2-D sinusoid  $A * \cos(\omega * k * (2 * i + 1)) * \cos(\omega * l * (2 * j + 1))$  is used to generate the matrix  $m$ , the transformed matrix will consist of one frequency component at the  $k$ th row and the  $l$ th column, where  $k = 0, 1, 2, \dots, n-1$ ,  $l = 0, 1, 2, \dots, n-1$ ,  $i = 0, 1, 2, \dots, n-1$ ,  $j = 0, 1, 2, \dots, n-1$ ,  $\omega = \pi/(2 * n)$ , and  $\pi = 3.141592654 \dots$ . If  $k$  and  $l$  are greater than zero, the magnitude of the component is  $A * n/2$ .

**See Function:** `dct2d()`.

### 8.1.4 Convolution

The **convolution theorem** is very important to digital signal processing. This theorem is as follows:

Suppose that  $x(n)$  and  $h(n)$  are time signals with discrete Fourier transforms  $X(k)$  and  $H(k)$ . The **convolution** of  $x(n)$  and  $h(n)$  is defined as the sequence  $y(n)$ , where:

$$y(n) = \sum_{k=-\infty}^{\infty} x(k) h(n-k)$$

Then the discrete Fourier transform  $y(n)$  is given by

$$Y(k) = X(k) H(k)$$

It is assumed that  $X(k)$  and  $H(k)$  are Fourier transforms of time domain sequences that have been properly augmented with zeros so as to implement linear, rather than circular, convolution.

Thus convolution in the time domain corresponds to multiplication in the frequency domain. This theorem can be used to prove Shannon's sampling theorem. Convolution is also very important to digital filter theory.

Note also that multiplication in the time domain corresponds to convolution in the frequency domain. This property can be exploited both in spectral estimation and in designing FIR filters with window methods.

#### Example

Find the convolution of two identical square pulses  $x(n)$  and  $h(n)$  of unit height, each with a duration of three samples. The answer is the triangular pulse  $y(n)$  with a duration of five samples, as shown in the following table.

|        |    |    |    |   |   |   |   |
|--------|----|----|----|---|---|---|---|
| $n$    | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| $x(n)$ | 0  | 0  | 1  | 1 | 1 | 0 | 0 |
| $h(n)$ | 0  | 0  | 1  | 1 | 1 | 0 | 0 |
| $y(n)$ | 0  | 1  | 2  | 3 | 2 | 1 | 0 |

Convolution can be efficiently implemented with the FFT.

**See Functions:** convolve(), convofft().

**Two-Dimensional Convolution.** The one-dimensional definition of convolution is easily extended to two dimensions. The **2-D convolution** of  $x(k_1, k_2)$  with  $h(k_1, k_2)$  is  $y(n_1, n_2)$ , where

$$y(n_1, n_2) = \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} x(k_1, k_2) h(n_1 - k_1, n_2 - k_2)$$

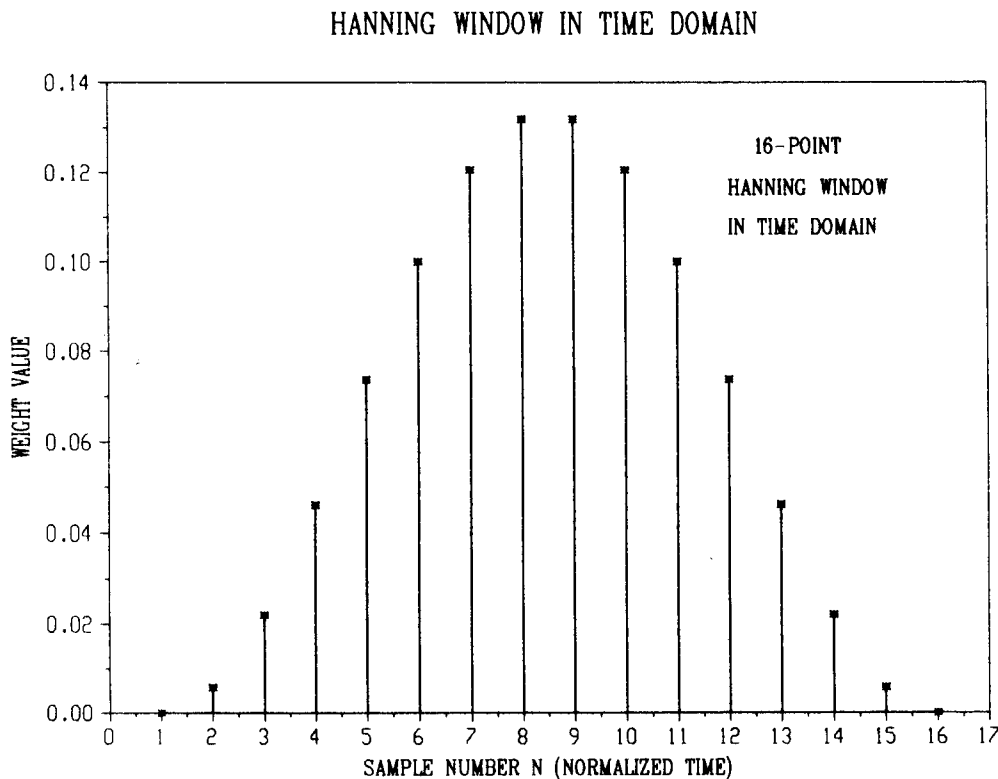
Two-dimensional convolution can be efficiently implemented with the 2-D FFT.

**See Function:** `conv2dft()`.

### 8.1.5 Windowing

Window functions are frequently used in digital signal processing. The most common applications are in spectral analysis, antenna design, and digital filtering. Data windows are usually applied with simple multiplication in the time domain.

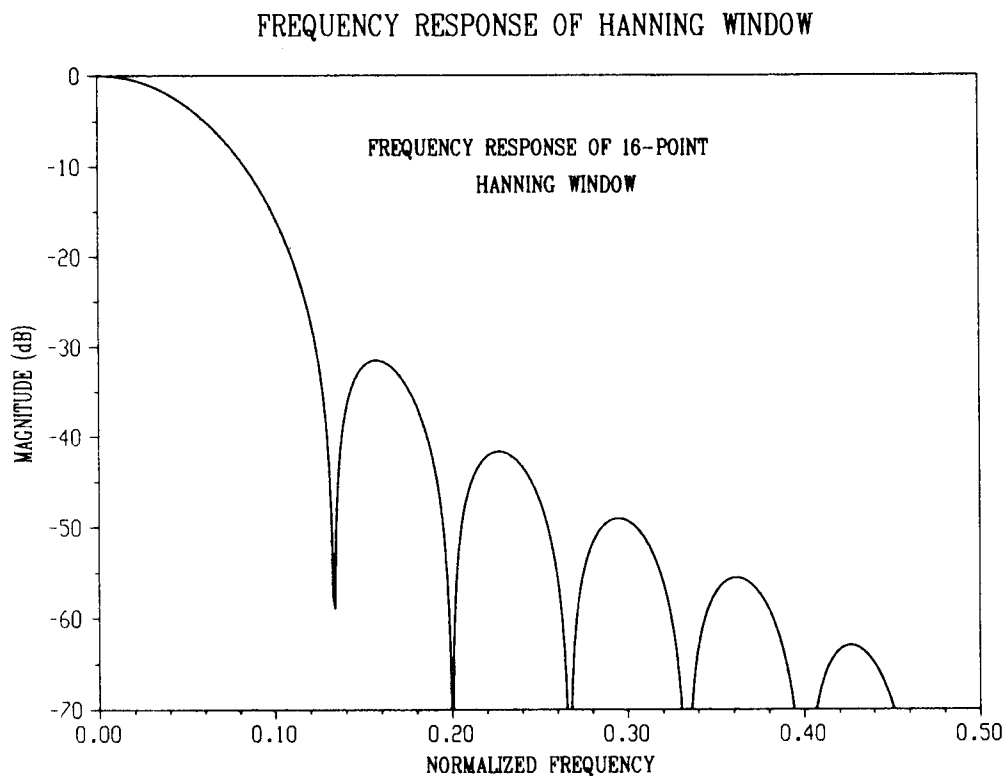
The time-domain weights of a 16-point Hanning window are shown in Figure 8.1. Note the tapered nature of this curve. Most data windows have this basic bell shape. Windows can also be implemented in the frequency domain with convolution (see the Convolution Theorem). The frequency response of a 16-point Hanning window is shown in Figure 8.2.



**Figure 8.1** 16-Point Hanning Window in Time Domain

Note that the first, second, and third sidelobes of the frequency response are approximately  $-30$ ,  $-40$ , and  $-50$  dB, respectively. This is the characteristic shape of all Hanning windows. The definition of a **dB** is given in the Digital Filtering section 8.2 that follows. Several other commonly used windows are

1. Hamming window
2. Kaiser-Bessel window
3. Blackman window
4. Approximate Blackman window
5. Blackman-Harris window



**Figure 8.2** Frequency Response of a Hanning Window

It is beyond the scope of this section to describe the subtle differences between these various windows. The effects of these various windows are qualitatively the same. For more information about the advantages of data windowing, the interested reader is advised to consult the excellent reference by F. J. Harris (1978).

Most window functions have a fixed windowing effect. However, the Kaiser–Bessel window provides a selectable amount of windowing by varying the value of  $b$  in the equation below. This makes the Kaiser–Bessel window useful for a variety of digital filter design applications.

A Kaiser–Bessel window of length  $M + 1$  is of the form

$$w[n] = I_0[b(1 - [(n - a)/a]^2)^{1/2}]/I_0(b), \quad 0 \leq n \leq M$$

$$= 0.0, \quad \text{otherwise}$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind and  $a = M/2$  and  $b$  is the shaping parameter.

**See Function:** `tdwindow()`.

## 8.2 DIGITAL FILTERING

The purpose of most filtering applications is either to enhance desirable signal frequencies, or to reject undesirable frequencies (e. g., noise). Digital filters are useful

in many engineering and data processing applications. Not only are their frequency responses often sharper than analog filters, but their phase responses can be made to be almost exactly linear. Many recent developments in high fidelity audio technology are due to digital signal processing techniques. For example, the improved dynamic range of compact disc players is mostly due to digital encoding and filtering methods.

This section deals with the following two kinds of digital filters:

1. Finite impulse response (**FIR**) filters
2. Infinite impulse response (**IIR**) filters

### 8.2.1 Finite Impulse Response (FIR) Filters

The basic form of all FIR filters is shown in Figure 8.3. Each stage of  $z^{-1}$  corresponds to one sample of delay. The triangular symbols signify multiplication by the appropriate filter weight  $h(i)$ . The sequence of filter weights  $h(n)$  is often referred to as the **impulse response** of the filter. The frequency content of a signal is altered by convolving the signal with the impulse response of the filter.

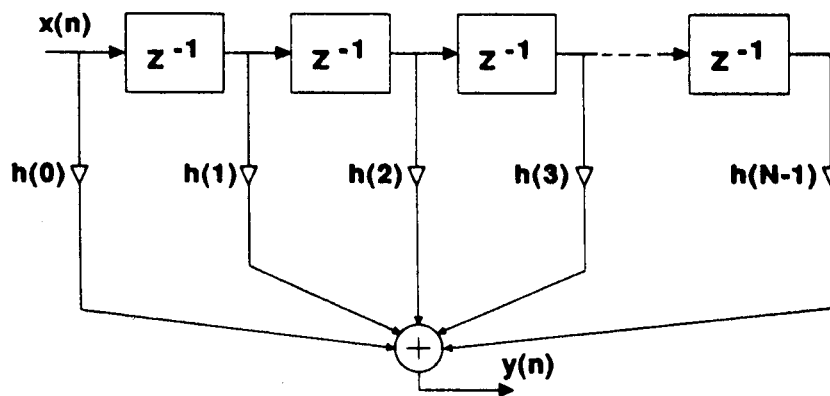


Figure 8.3 Finite Impulse Response Filter

The most commonly used filters are **lowpass**, **highpass**, and **bandpass** designs. Idealizations of these filter types are shown in Figure 8.4. Lowpass filters reject high frequencies. Highpass filters reject low frequencies. Bandpass filters reject high and low frequencies.

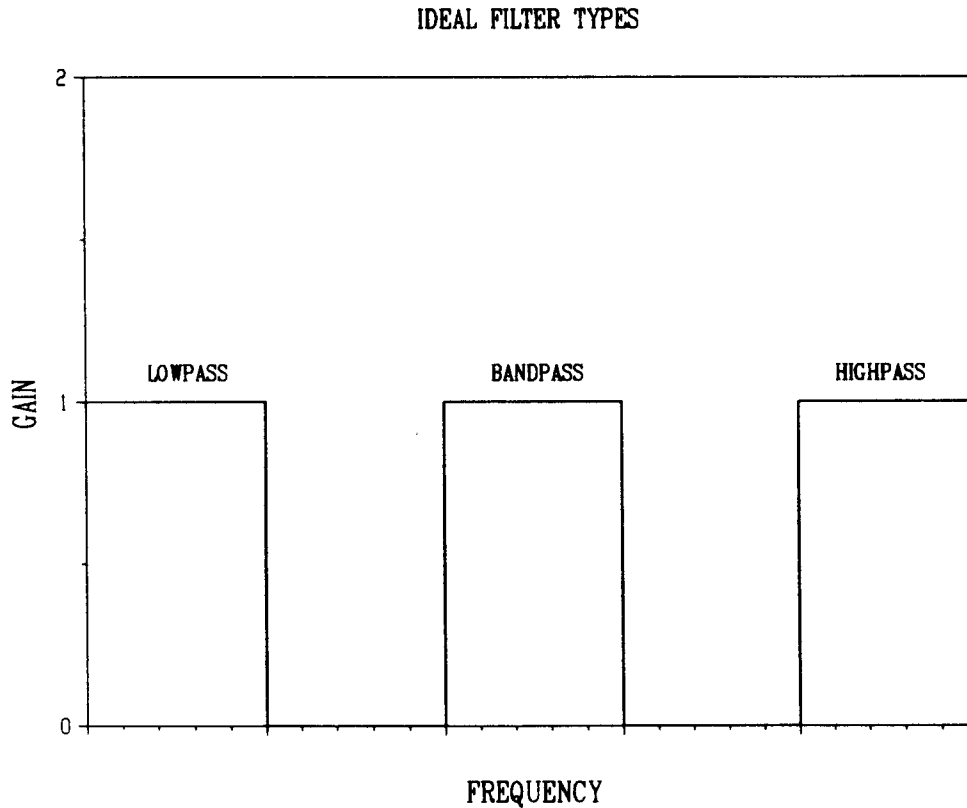
The extremely sharp rolloffs of ideal filters can only be realized with an infinite number of time delays. The impulse response of an ideal lowpass filter with cutoff frequency  $F_c$  is given by

$$h_L(n) = \frac{\sin(2\pi F_c n)}{\pi n} \quad -\infty < n < \infty$$

The impulse response of an ideal highpass filter with cutoff frequency  $F_c$  is given by

$$h_H(n) = \delta(n) - \frac{\sin(2\pi F_c n)}{\pi n} \quad -\infty < n < \infty$$

The impulse response of an ideal bandpass filter with upper cutoff frequency  $F_c$  and lower cutoff frequency  $F_1$  is given by



**Figure 8.4** Ideal Filter Types

$$h_B(n) = \frac{\sin(2\pi F_c n) - \sin(2\pi F_1 n)}{\pi n} \quad -\infty < n < \infty$$

Unfortunately, the finite nature of most digital data dictates that the digital filter must also be finite. In general, the duration of the filter's impulse response should be less than one-tenth of the duration of the data that is to be filtered. Truncating the number of filter weights produces undesirable oscillations at the corner frequencies of the filter's response. These oscillations are often called the **Gibbs phenomenon**. Window design approaches reduce the Gibbs effect by placing a tapered window across the sequence of truncated filter weights.

It is beyond the scope of these sections to provide all the mathematical development necessary to understand digital filter theory. For more information about digital filtering, the interested reader is advised to consult the excellent references on the subject. Instead, the Kaiser window method of filter design is described. The following sections deal exclusively with Kaiser window design approaches.

**Kaiser Window Filter Design.** The basic approach of window design methods is to truncate the infinite length impulse response of an ideal frequency filter by multiplying by a time-domain window (in this case, a Kaiser–Bessel window). Time-domain windows are bell-shaped, which tapers the filter impulse response and reduces the Gibbs effect. The Kaiser–Bessel window is useful because of the flexibility it lends to filter designs. Recall that a Kaiser–Bessel window of length  $N = M + 1$  is of the form

$$\omega[n] = I_0[b(1 - [(n - a)/a]^2)^{1/2}]/I_0(b), \quad 0 \leq n \leq M$$

$$= 0.0, \quad \text{otherwise}$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind and  $a = M/2$  and  $b$  is the shaping parameter.

In the frequency domain, the passband of a filter is separated from its stopband by the transition band. The gain in the passband is reasonably close to one. Frequencies in the stopband are attenuated by some specified amount  $d_s$  (i. e., the gain in the stopband is less than  $d_s < 1$ ). Frequently the gain in the stopband is specified in *decibels* (dB):

$$D_s \text{ dB} = 20 \log_{10}(d_s)$$

The stopband attenuation of Kaiser window filters is controlled by varying the shaping parameter,  $b$ . Increasing  $b$  increases the stopband attenuation. Kaiser determined that the stopband attenuation,  $D_s = -A$  was empirically related to  $b$ :

$$b = 0.1102(A - 8.7), \quad \text{if } A > 50$$

$$b = 0.5842(A - 21)^{0.4} + 0.07886(A - 21) \quad \text{if } 21 \leq A \leq 50$$

$$b = 0.0, \quad \text{if } A < 21$$

Values of  $b$  for some typical stopband attenuations,  $A$ , are given in the following table.

|     |       |       |       |       |       |       |       |        |
|-----|-------|-------|-------|-------|-------|-------|-------|--------|
| $b$ | 2.120 | 3.384 | 4.538 | 5.658 | 6.764 | 7.865 | 8.960 | 10.056 |
| $A$ | 30    | 40    | 50    | 60    | 70    | 80    | 90    | 100    |

Let  $F_t$  be the width of the transition band of the filter, normalized by the sampling rate (i. e.,  $0 \leq F_t \leq 0.5$ ) For lowpass and highpass filters, the length of the filter,  $N$ , needed to achieve the specified values of  $A$  and  $F_t$  is given by

$$N = (A - 8)/(14.357F_t)$$

For similar design specifications, symmetric bandpass filters are about twice as long as lowpass and highpass filters.

**Lowpass Filtering.** The simplest digital lowpass filter is an  $n$ -point averager. Every output sample of this filter is a local average of  $n$  neighboring input samples. For a comparison, the impulse response of a 4-point averager is shown in Figure 8.5 along with the impulse response of a 10-point Kaiser–Bessel lowpass filter.

Digital lowpass filters are characterized by a passband,  $(0, F_1)$ , over which the frequency response is reasonably close to unity gain. They also often have a stopband,  $(F_2, F_s/2)$  over which frequencies are attenuated by some specified amount. The maximum attenuation is at the highest frequency,  $F_s/2$ .

The frequency parameter,  $fc$ , sets the nominal passband ( $-6$  dB) of the Kaiser lowpass filter, since it is the average of  $F_1$  and  $F_2$ :

$$fc = (F_1 + F_2)/2$$

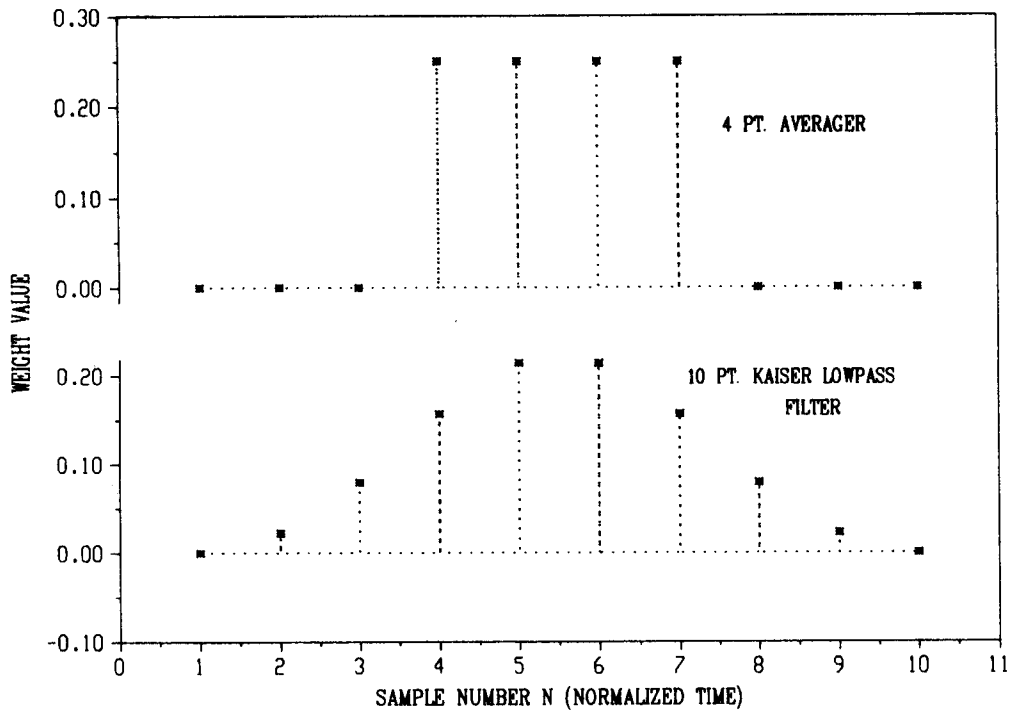


Figure 8.5 Impulse Response of Lowpass Filters

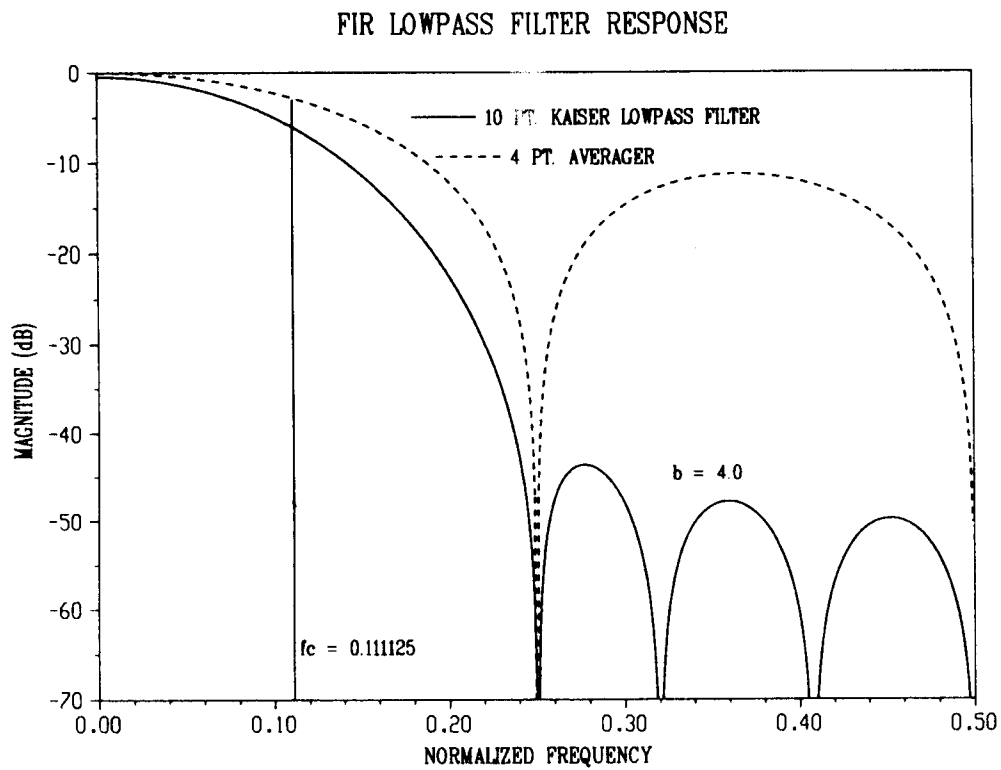


Figure 8.6 Frequency Response of Lowpass Filters

The transition band,  $F_t$ , is the difference between the passband and the stopband frequencies:

$$F_t = F_2 - F_1$$

As described previously, these parameters can be used to design a lowpass filter with any desired frequency response. The 3 dB cutoff of a filter,  $f_0$ , is the frequency where the response is at half power (0.707107). The (3 dB) cutoff of an “ $n$ -point” averager is

$$f_0 = 0.4445/n$$

The frequency responses of these two lowpass filters are shown in Figure 8.6. The stopband parameter,  $b = 4$ , sets the stopband attenuation of the Kaiser lowpass filter to about  $-45$  dB. The cutoff frequency,  $f_c = 0.111125$ , of the Kaiser filter was set equal to the 3 dB cutoff of the 4-point averager,  $f_0 = 0.4445/4$ . The frequency response of the Kaiser lowpass filter at  $f_c$  is attenuated by approximately 6 dB. Note that the 10-point Kaiser filter has a much better lowpass response than the 4-point averager.

**See Function:** `lowpass()`.

**Highpass Filtering.** The “2-point” difference filter is the simplest highpass filter. This filter is an antisymmetric (odd) function. It is implemented by replacing each data point with the difference of itself and its nearest neighbor. For a comparison, the impulse response of a 2-point difference filter is shown in Figure 8.7 along with the impulse response of a 25-point Kaiser–Bessel highpass filter. The upper plot is the response of a “2-point” difference filter, and the lower plot is the response of a Kaiser highpass filter.

Digital highpass filters are characterized by a passband,  $(F_2, F_s/2)$ , over which the filter’s response is reasonably close to unity gain. They also often have a stopband,  $(0, F_1)$ , over which frequencies are attenuated by some specified amount (i. e., the gain in the stopband is less than  $d_s < 1$ ). The maximum attenuation is at zero frequency (DC).

The frequency parameter,  $f_c$ , sets the nominal passband ( $-6$  dB) of the Kaiser lowpass filter, since it is the average of  $F_1$  and  $F_2$ :

$$f_c = (F_1 + F_2)/2$$

The transition band,  $F_t$ , is the difference between the passband and the stopband frequencies:

$$F_t = F_2 - F_1$$

As described previously, these parameters can be used to design a highpass filter with any desired frequency response.

It is instructive to compare the frequency response of the “2-point” difference filter with the Kaiser highpass filter. The frequency responses are plotted in Figure 8.8. The 3 dB cutoff of a “2-point” difference filter is:

$$f_0 = \sin^{-1}(0.707107)/3.14159265 = 0.25$$

The 6 dB cutoff frequency of the Kaiser filter is  $f_c = 0.10$ . The stopband parameter,  $b = 2.16$ , sets the stopband attenuation of the Kaiser lowpass filter to about  $-32$  dB. Note that the 25-point Kaiser filter has a much better highpass response than the 2-point difference filter.

**See Function:** `highpass()`.

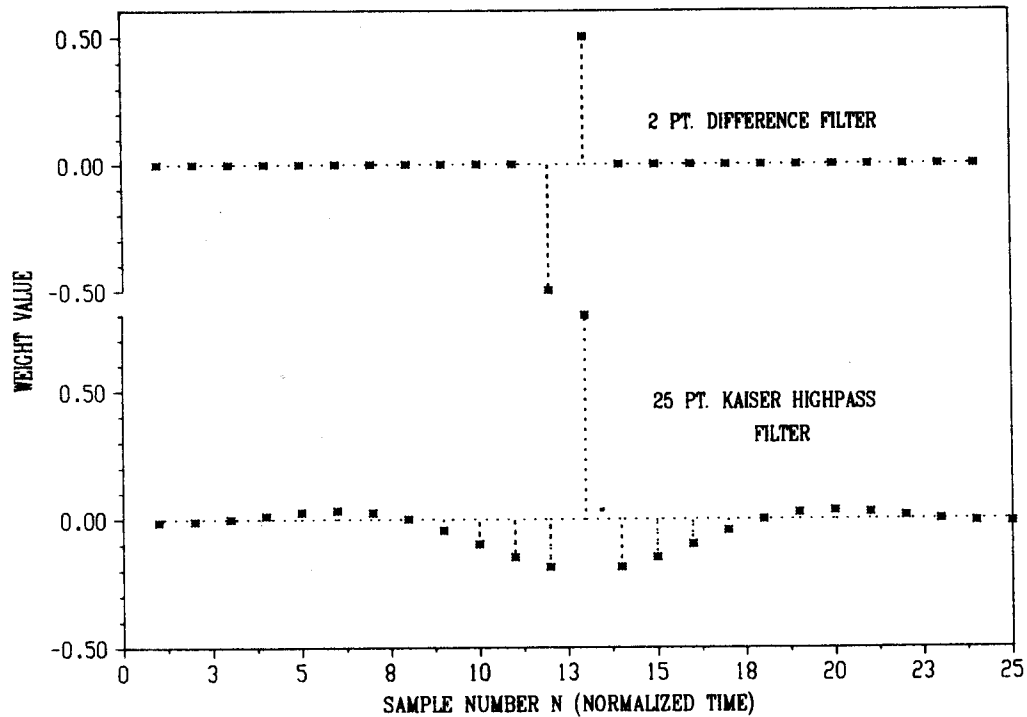


Figure 8.7 Impulse Response of Highpass Filters

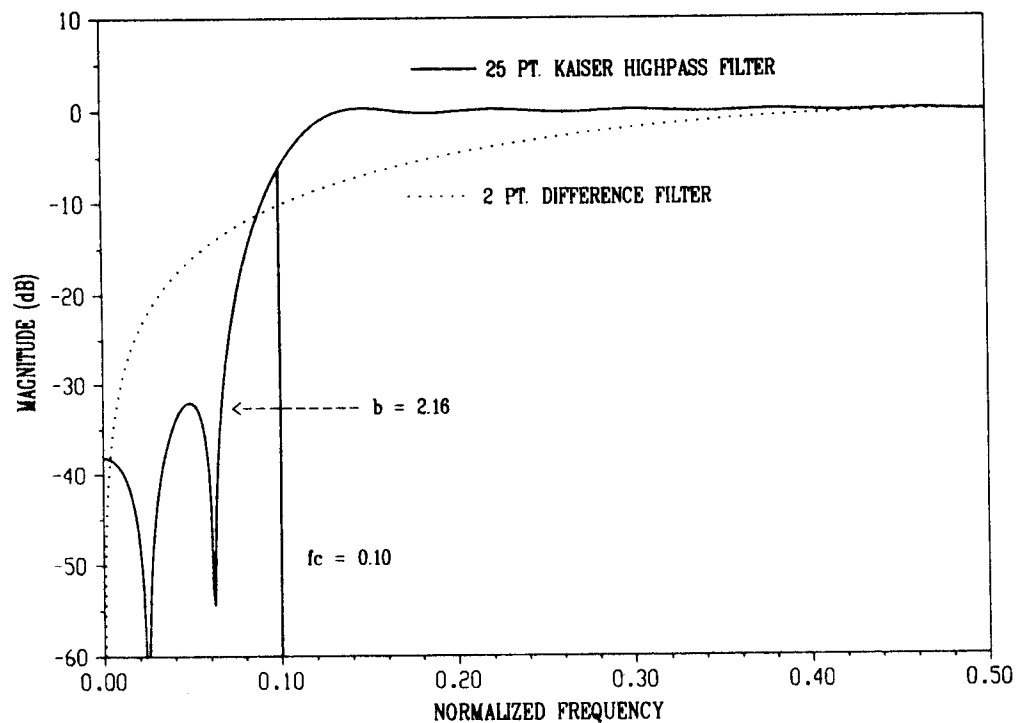


Figure 8.8 Frequency Response of Highpass Filters

**Bandpass Filtering.** Digital bandpass filters are characterized by one passband,  $(F_1, F_2)$ , over which the filter's response is reasonably close to unity gain. The impulse response of a 31-point Kaiser bandpass filter is shown in Figure 8.9. Bandpass

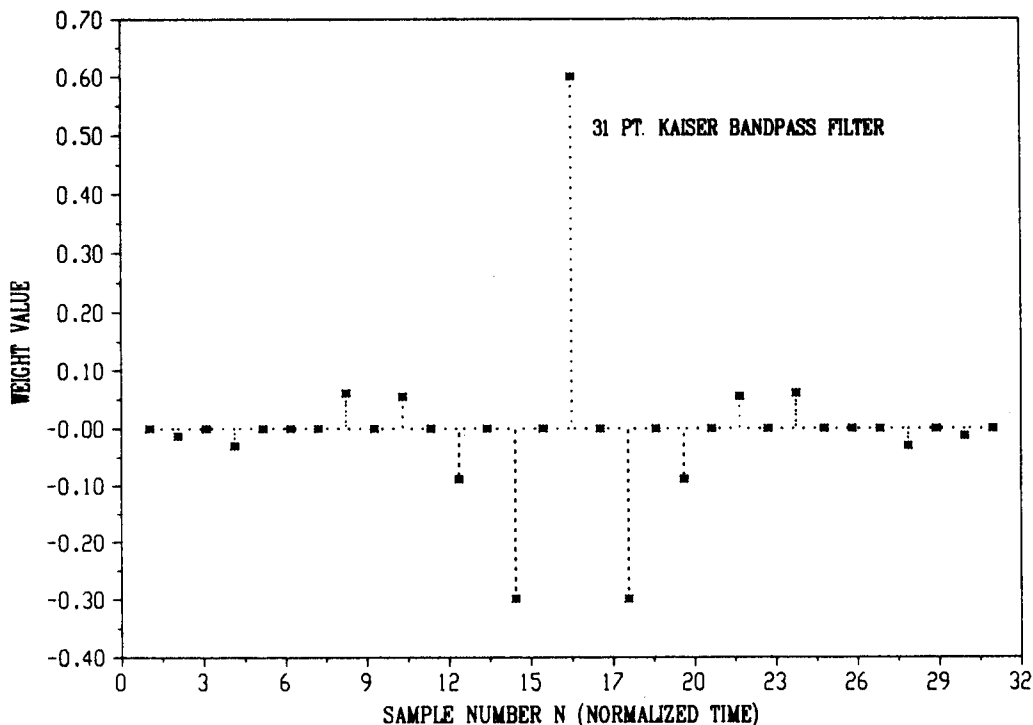


Figure 8.9 Impulse Response of a Bandpass Filter

filters have two stopbands,  $(0, F_{s1})$ , and  $(F_{s2}, F_s/2)$ , over which frequencies are attenuated by some specified amount (i. e., the gain in the stopband is less than  $d_s < 1$ ).

The cutoff frequencies,  $fl$  and  $fh$ , set the nominal passband (and stopband) of the Kaiser window filter, since they are averages of the “corner” frequencies:

$$fl = (F_{s1} + F_1)/2$$

$$fh = (F_2 + F_{s2})/2$$

The response of a Kaiser bandpass filter at the cutoff frequencies,  $fl$  and  $fh$ , is attenuated by approximately 6 dB.

The transition band,  $F_t$ , is the difference between the passband and the stopband frequencies:

$$F_t = F_{s2} - F_2 = F_1 - F_{s1}$$

Frequently, the gain in the stopband is specified in *decibels* (dB):

$$D_s dB = 20 \log_{10}(d_s)$$

As described previously, these parameters can be used to design filters with any desired frequency response.

The length of the bandpass filter,  $N$ , needed to achieve the specified values of  $A$  and  $F_t$  is given by

$$N = (A - 8)/(7.1785F_t)$$

Note that bandpass filters are about twice as long as highpass and lowpass designs for comparable transition bandwidths and stopband attenuations.

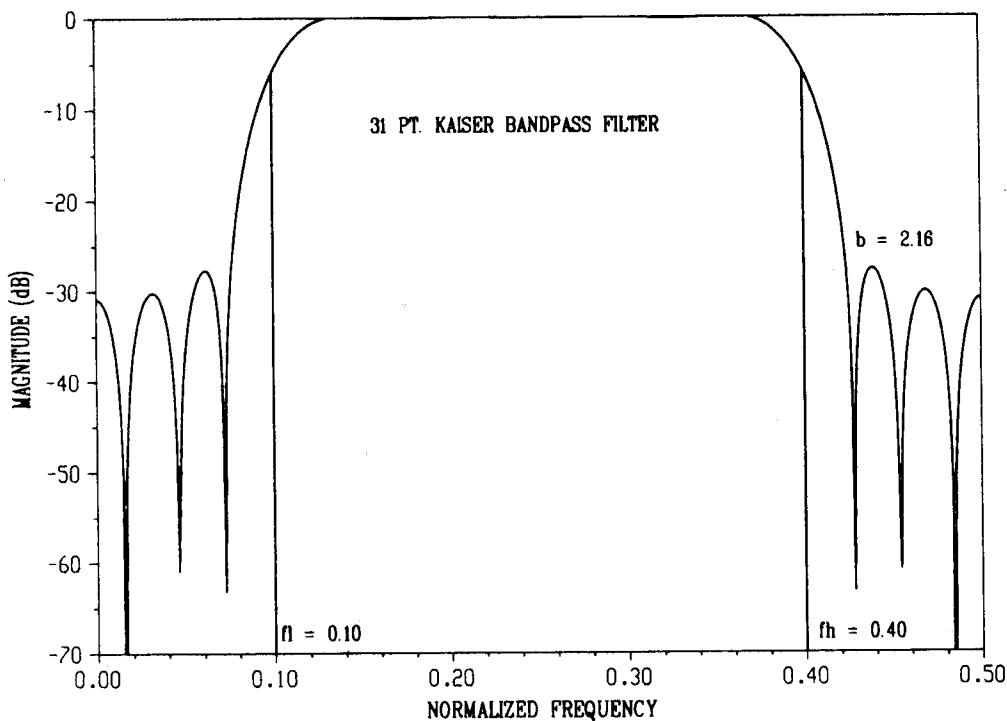


Figure 8.10 Bandpass Frequency Response

The frequency response of the Kaiser bandpass filter has the symmetric shape shown in Figure 8.10. The  $-6$  dB cutoff frequencies for the filter are  $fl = 0.10$  and  $fh = 0.40$ . The stopband attenuation is about  $-27$  dB ( $b = 2.16$ ).

See Function: `bandpass()`.

### 8.2.2 Infinite Impulse Response Filters

An **infinite impulse response (IIR)** filter is a digital filter that theoretically “rings” forever when excited by an impulse. However, in practice the responses of these filters can be considered finite after some nominal time interval. The general IIR filter is described by the following difference equation:

$$y(n) = \sum_{r=1}^N b_r y(n-r) + \sum_{k=0}^M a_k x(n-k)$$

The  $z$ -transform of this difference equation is given by

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^M a_k z^{-k}}{1 - \sum_{r=1}^N b_r z^{-r}}$$

The **frequency response** of this filter is determined by evaluating the above  $z$ -transform on the unit circle  $z = e^{j\omega t_s}$ :

$$H(e^{j\omega t_s}) = \frac{\sum_{k=0}^M a_k e^{jk\omega t_s}}{1 - \sum_{r=1}^N b_r e^{jr\omega t_s}}$$

IIR filters are often implemented as a cascade of second order **biquad** sections, as shown in Figure 8.11. The transfer function of each biquad stage can be expressed as a ratio of second order polynomials:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1 z^{-1} - a_2 z^{-2}}{1 - b_1 z^{-1} - b_2 z^{-2}}$$

Choosing the poles in each stage to be complex conjugate pairs reduces the errors due to coefficient truncation.

The classical techniques of analog filter design can also be used to design IIR filters. Several standard techniques exist for transforming analog filter designs into IIR designs. **Impulse invariance** and **bilinear transformation** are the most popular.

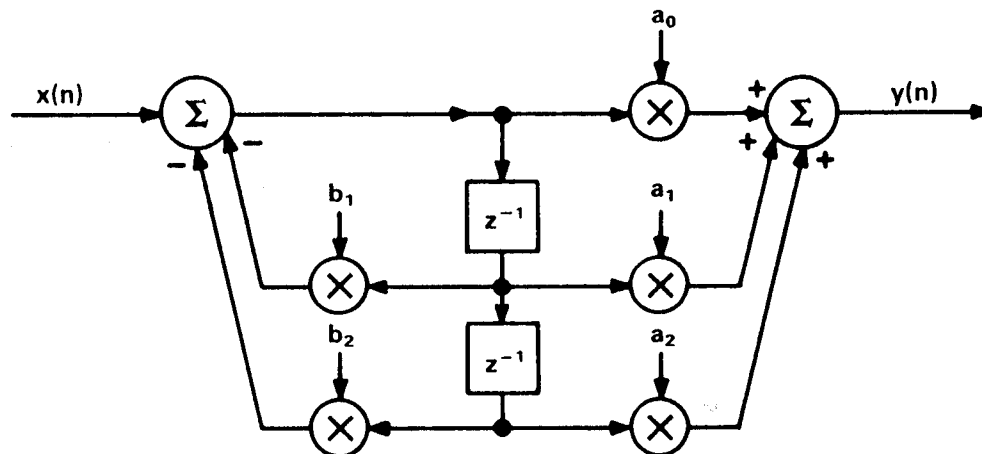


Figure 8.11 Biquad IIR Filter

See Function: `iirfilt()`.

**The Bilinear Transform Method.** The **bilinear transform (BLT)** is the most popular technique for translating analog filter designs into IIR designs. The following substitution defines the bilinear transformation from the *s*-plane to the *z*-plane:

$$s = \frac{z - 1}{z + 1}$$

This mapping can be used to translate analog filter transfer functions into digital IIR filter transfer functions. However, the critical frequencies of the analog design must be prewarped to compensate for the nonlinear mapping of the analog frequency *f<sub>A</sub>* axis onto the normalized digital frequency *f<sub>D</sub>* axis:

$$f_A = \tan(\pi * f_D)$$

where  $0 < f_D < 1/2$  and  $\pi = 3.14159 \dots$

**Example:**

Given the following analog lowpass filter, use the BLT to design an equivalent IIR lowpass filter, assuming a sampling rate of 500 radians/sec.

$$H(s) = \frac{100}{s + 100}$$

**Solution:** The cutoff frequency (100 radians/sec) must be prewarped to

$$\omega_A = \tan(\pi * 100/500) = 0.726542528$$

The resulting frequency-scaled Laplace transfer function is

$$\hat{H}(s) = \frac{0.726542528}{s + 0.726542528}$$

Letting  $s = (z - 1)/(z + 1)$ , the desired IIR filter is given by

$$\begin{aligned} H(z) &= \frac{0.726542528 + 0.726542528z}{(0.726542528 - 1) + (0.726542528 + 1)z} \\ &= \frac{0.726542528 + 0.726542528z^{-1}}{1.726542528 - 0.273457472z^{-1}} \end{aligned}$$

**See Function:** `bilinear()`.

### 8.2.3 Data Smoothing

Data smoothing is a lowpass filtering process that reduces the high frequency noise, thus enhancing the low frequency signals. The noise typically consists of both high and low frequencies, while it is assumed that the signal has mainly low frequencies.

The design concerns for smoothing filters are quite similar to those of decimation filters. It could be said that a smoothing filter is simply a decimation filter, where the data rate has not been reduced.

Each new signal and noise combination must be considered separately in the smoothing filter design. It is impossible to design a smoothing filter that will work for all situations.

All smoothing processes remove noise at the expense of introducing some errors in the signal. For example, if the signal has high frequency information, some of this energy will be lost. Special care must be taken to ensure that the noise reduction process does not lead to excessive signal loss. In smoothing by large amounts, it is best that the signals of interest are sufficiently oversampled or they will be filtered out with the noise.

**See Function:** `smooth()`.

### 8.2.4 Median Filtering

**Median filtering** is a nonlinear technique that is ideal for removing shot-noise or noise from digital dropout sometimes found in images. This technique has also been used in

speech processing for smoothing estimates of zero crossings and pitch contours. In its simplest form the median filter is a sliding window that extends over an odd number of data points. At each new window position, the central data point is replaced by the median of the window. The effect of this processing is to filter out large (high frequency) noise spikes without significantly degrading the desired data. A fast sorting algorithm (e. g., Heapsort) can be used to speed up the filtering. Median smoothing is ideal for correcting isolated errors due to random data loss.

**See Function:** median().

## 8.3 SAMPLE RATE CONVERSION

Frequently in digital signal processing, it is necessary to match systems that are sampled at different rates. **Sample rate conversion** is the process of converting a signal sampled at one rate to any other rate. When the new sampling rate is lower, this process is called **decimation**. When the new sampling rate is higher, this process is called **interpolation**. The following sections present some of the fundamentals of sample rate conversion.

### 8.3.1 Decimation

Originally, the term **decimation** was used in reference to a 10 percent reduction in sampling rate. Currently this signal processing term is used for reductions of arbitrary amounts. Lowering the sampling rate of a data set can be very useful in many data processing applications. A common case occurs with graphics packages, which are often designed to handle some maximum number of data points. One may wish to plot a much larger number of points, and may effectively do so by using decimation techniques.

When the sampling rate is reduced by an integer amount, the resampling is called **integer decimation**. The output sequence is formed by “hopping” a lowpass filter across the input samples in integral steps that correspond to the rate reduction factor. It should be emphasized that the output sequence represents all of the low-frequency information of the original input sequence, and is not simply a subset formed by deleting the intermediate input points.

When decimating by large factors, one must take care that the signals of interest are sufficiently oversampled, or they will be filtered out with the noise. It is important to note that even if the input data are appropriately oversampled, all resampling approaches introduce some errors (resampling ambiguities). These errors can be controlled by adjusting the stopband attenuation,  $d_s$ , of the lowpass filter to match the accuracy that is needed to represent the signal. For example, if the signal needs to be represented to only three-place accuracy, the stopband attenuation of the lowpass filter should be set to  $d_s = 0.001$  (or  $20 \cdot \log(d_s) = -60dB$ ). Unnecessarily high accuracies should be avoided since they lead to resampling filters with long impulse responses, which exaggerate filter edge effects.

**See Function:** resample(), downsamp().

### 8.3.2 Interpolation

Interpolation is the opposite of the decimation process just described (i. e., the sampling rate is increased). The most common form of interpolation is encountered in the interpretation of tabulated data. Before the advent of the calculator and the computer, scientists and mathematicians had to rely on volumes of tables for computations that involved special functions. Thus, interpolation approaches evolved over hundreds of years from the need to generate and evaluate the points “in between” mathematical tables. The famous Lagrange interpolation has already been described in the chapter on numerical analysis.

The interpolation of digital signal processing is similar to the classical interpolation methods. For an excellent comparison of these approaches, the reader is encouraged to consult the paper in the references by R. W. Schafer and L. R. Rabiner (1973). There are some basic differences in the digital signal processing approach to interpolation:

1. The  $n$ -point interpolation is implemented by inserting  $n - 1$  zeros between the original time samples and filtering the result with an FIR lowpass filter.
2. The interpolated output sequence is, in general, equally spaced.
3. The interpolation at the input data samples is not exact. In Lagrange interpolation, the appropriate output samples are always equal to the original input values.
4. The DSP techniques cannot interpolate at the edges of data sets.
5. For a sufficiently high sampling rate, the accuracy of the interpolation is determined by the design of the lowpass filter. Often this allows the interpolation to be arbitrarily accurate.

**See Functions:** `resample()`, `interp1()`, `interp()`, `spline()`.

## 8.4 SPECTRAL ANALYSIS

Spectral analysis is the determination of the frequency content of a signal. In low-noise environments, the harmonic content of signals with constant amplitude and phase is well characterized by the **fast Fourier transform (FFT)**. The FFT is simply an efficient approach to evaluating the previously described discrete Fourier transform.

**Power spectral analysis** provides a valuable solution to the problem of estimating the harmonic content of random signals. Most real-world signals are not well represented by constant amplitude and constant phase sinusoids. For example, speech signals often resemble noise, more than sinusoids. Due to their inherent randomness, such signals are better characterized statistically by their mean (average value) and variance (average power) over the observation interval of interest.

### 8.4.1 The Fast Fourier Transform

The discrete Fourier transform (DFT) plays an important role in digital signal analysis. Yet it was not until the development of the fast Fourier transform (FFT) that the DFT

became computationally feasible [J. W. Cooley and J. W. Tukey (1965)]. Although the FFT is most commonly used for spectral estimation, its efficiency can be exploited to perform a variety of signal processing tasks such as convolution and correlation. This makes the FFT one of the most valuable tools in digital signal processing.

With most FFT approaches, all the calculations can be done “in place.” This means that the desired transform is written over the original input data. Many FFT algorithms require that the length of the transform,  $N$ , must be a “power of two”:

$$N = 2^M$$

where  $M$  is some integer.

Direct evaluation of the DFT requires a number of computations that is proportional to  $N^2$ , where  $N$  is the length of the transform. Yet with FFT approaches, the number of required computations is only proportional to  $N * \log_2(N) = NM$ . Thus, the relative increase in efficiency is proportional to

$$N / \log_2(N) = N/M$$

which becomes quite large as  $N$  increases.

“Radix-2” FFT algorithms decompose the full transform into  $M$  stages of  $N/2$  2-point transforms. “Radix-4” FFT algorithms are implemented with  $M/2$  stages of  $N/4$  4-point transforms. “Radix-4 + 2” algorithms break the transform down into as many “radix-4” stages as possible, and finish with a “radix-2” stage if necessary.

The increase in efficiency that results from using an FFT can be quantified by counting the total number of required operations (real adds and multiplies). The following table compares the total number of operations required for “radix-2” and “radix-4” FFT algorithms with direct DFT evaluation.

| $N$   | Direct DFT Computation | Radix-2 FFT  | Radix -4 FFT |
|-------|------------------------|--------------|--------------|
| 16    | 1.860e + 003           | 2.300e + 002 | 1.820e + 002 |
| 64    | 3.200e + 004           | 1.542e + 003 | 1.254e + 003 |
| 256   | 5.212e + 005           | 8.710e + 003 | 7.174e + 003 |
| 1024  | 8.376e + 006           | 4.506e + 004 | 3.738e + 004 |
| 4096  | 1.342e + 008           | 2.212e + 005 | 1.843e + 005 |
| 16384 | 2.147e + 009           | 1.049e + 006 | 8.765e + 005 |

Note the tremendous reduction in number of operations required by the FFT algorithms for all of the transform lengths ( $N$ ) shown. This is especially true as  $N$  increases. Note also that the “radix-4” algorithm requires on the average about 18 percent fewer operations than the “radix-2” FFT.

**See Functions:** `fftrad2()`, `fft42()`, `fftrealm()`, `fftr_inv()`.

### 8.4.2 The Power Spectrum

A power spectrum is a display of the average power (variance) of a signal as a function of frequency. While there are several methods of computing the power spectrum, the periodogram approach is the most common. Most modern spectrum analyzers use this approach. Periodogram analysis exploits the computational efficiency of the fast Fourier transform (FFT). The signal length is divided into equal subintervals (possibly overlapping) where the subinterval length is a “power of two.” Each segment is Fourier transformed, and all of the transforms are averaged, yielding the desired power spectrum.

Welch (1970) found that the maximum benefit of overlapping occurs when the overlapped length is half the segment length. The variance of the spectral estimate is about a factor of two less than it is for the nonoverlapping case. However, twice as much computation is required.

Periodograms allow a convenient tradeoff between the frequency resolution and the variance of the spectral estimate. If more frequency resolution is desired (and more noise variance is tolerable), simply divide the signal length into fewer segments and perform longer Fourier transforms.

Time domain windows can be used to help reduce degradations in spectral estimates caused by finite data length. Multiplying the time data by tapered windows lowers the sidelobes that cause “spectral-leakage” in the frequency domain.

**See Functions:** `powspec()`, `Cpowspec()`.

### 8.4.3 Correlation and Covariance

In many DSP applications it is necessary to compare two random signals. It may be suspected that signals are similar, though not identical. **Correlation** procedures provide a quantitative measure of waveform similarity and are particularly useful in random noise environments. Correlation is important in matched filtering applications and power spectral estimation.

**Autocorrelation** (sometimes called autocovariance) is a useful tool in random signal and power spectral analysis. The autocorrelation function and the power spectrum are Fourier transform pairs.

The autocorrelation function  $R_{xx}(m)$  of the **zero-mean** real sequence  $x(n)$  is given by:

$$R_{xx}(m) = \frac{1}{N - |m|} \sum_{n=0}^{N-|m|-1} x(n)x(n+m) \quad \text{where } |m| < N$$

Although the autocorrelation is defined for  $2 * N - 1$  lags, the central 10 percent are the most accurate estimates of the true autocorrelation. The autocorrelation function is symmetric for real data.

The **crosscorrelation**  $R_{xy}(m)$  (sometimes called covariance) between two real **zero-mean** sequences  $x(n)$  and  $y(n)$  is

$$R_{xy}(m) = \frac{1}{N-m} \sum_{n=0}^{N-m-1} x(n)y(n+m) \quad 0 \leq m < N$$

$$R_{xy}(-m) = \frac{1}{N-m} \sum_{n=0}^{N-m-1} x(n+m)y(n) \quad 0 \leq m < N$$

Again, the correlation estimate is most accurate for the central (e. g., 10 percent) lags. Unlike autocorrelation, the **crosscorrelation** function is not generally symmetric.

Both autocorrelation and crosscorrelation can be efficiently implemented with the FFT.

**See Functions:** `crosscor()`, `autocor()`, `autofft()`, `crossfft()`.

**Two-Dimensional Correlation.** Two-dimensional correlation can be used to measure the spatial misregistration between two similar images. The definitions for autocorrelation and crosscorrelation can be directly extended to two dimensions. The crosscorrelation  $R_{xy}(n_1, n_2)$  between two real **zero-mean** 2-D sequences  $x(n_1, n_2)$  and  $y(n_1, n_2)$  has the following four quadrant structure:

$$R_{xy}(n_1, n_2) = \frac{1}{N_1 - n_1} \frac{1}{N_2 - n_2} \sum_{k_1=0}^{N_1-n_1-1} \sum_{k_2=0}^{N_2-n_2-1} x(n_1, n_2)y(n_1 + k_1, n_2 + k_2)$$

$$R_{xy}(n_1, -n_2) = \frac{1}{N_1 - n_1} \frac{1}{N_2 - n_2} \sum_{k_1=0}^{N_1-n_1-1} \sum_{k_2=0}^{N_2-n_2-1} x(n_1, n_2 + k_2)y(n_1 + k_1, n_2)$$

$$R_{xy}(-n_1, n_2) = \frac{1}{N_1 - n_1} \frac{1}{N_2 - n_2} \sum_{k_1=0}^{N_1-n_1-1} \sum_{k_2=0}^{N_2-n_2-1} x(n_1 + k_1, n_2)y(n_1, n_2 + k_2)$$

$$R_{xy}(-n_1, -n_2) = \frac{1}{N_1 - n_1} \frac{1}{N_2 - n_2} \sum_{k_1=0}^{N_1-n_1-1} \sum_{k_2=0}^{N_2-n_2-1} x(n_1 + k_1, n_2 + k_2)y(n_1, n_2)$$

where  $0 \leq n_1 < N_1$  and  $0 \leq n_2 < N_2$ .

The autocorrelation formulas are a trivial modification of the crosscorrelation definitions. To save space, the autocorrelation formulas are not given. However, it may be worth noting an additional symmetry that results for the 2-D autocorrelation function:

$$R_{xx}(-n_1, -n_2) = R_{xx}(n_1, n_2)$$

As in the one-dimensional case, the estimates of the central lags are the most accurate.

The 2-D FFT can be used to efficiently correlate moderately large sets of 2-D data.

**See Functions:** `auto2dft()`, `cross2dft()`.

## 8.5 ADAPTIVE SIGNAL PROCESSING

**Adaptive filters** automatically change their coefficients based on the signal and noise statistics to enhance the signal. The adaptive filters presented in this section are all of the **transversal (FIR)** type, as shown in Figure 8.12. The circular symbols with arrows are meant to indicate that the filter weights  $h(i)$  are not fixed, but adapt depending on the signal and noise environment. There are two basic adaptive modes of operation: **open-loop** and **closed-loop**. Batch filtering processes like the Wiener filter are often used for the open-loop mode. The closed-loop mode of operation is efficiently implemented with the **LMS** (least mean square) algorithm developed by Widrow. In the steady state, both modes of operation approach the Wiener solution.

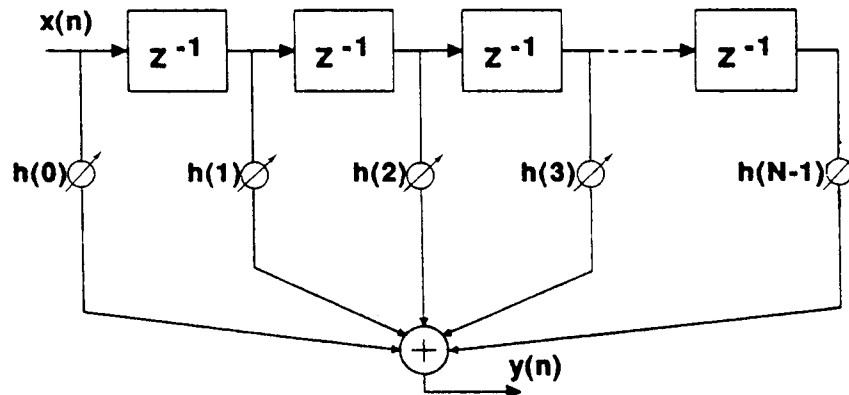


Figure 8.12 Adaptive Transversal Filter

### 8.5.1 Wiener Filtering

Norbert Wiener's work in adaptive filtering originally appeared in a book of limited circulation in February 1942. This work detailed an intricate mathematical procedure for solving the transcendental problem needed to derive optimal prediction filters. Wiener's results sparked several related military reports that dealt with gunnery prediction and fire control. However, Wiener is perhaps remembered most for developing an approximate (and more computationally feasible) solution to the optimal filter problem. This solution is the famous **Wiener-Hopf** equation expressed in matrix form:

$$\begin{bmatrix} r_{xx}(0) & r_{xx}(1) & r_{xx}(2) & \cdots & r_{xx}(N-1) \\ r_{xx}(1) & r_{xx}(0) & r_{xx}(1) & \cdots & r_{xx}(N-2) \\ r_{xx}(2) & r_{xx}(1) & r_{xx}(0) & \cdots & r_{xx}(N-3) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{xx}(N-1) & r_{xx}(N-2) & r_{xx}(N-3) & \cdots & r_{xx}(0) \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \\ h_{N-1} \end{bmatrix} = \begin{bmatrix} r_{xx}(0) \\ r_{xx}(1) \\ r_{xx}(2) \\ \vdots \\ r_{xx}(N-1) \end{bmatrix}$$

where

$$R_{xx}(m) = [r_{xx}(0), r_{xx}(1), \dots, r_{xx}(N-1)]$$

is the autocorrelation function of the signal of interest  $X(n)$  and

$$H_{LMS}(m) = [h_0, h_1, \dots, h_{N-1}]^T$$

is the desired **Wiener filter**. The Wiener filter is a transversal matched filter. Under the proper assumptions, the Wiener solution can be shown to be minimum mean square.

The Wiener–Hopf equation can be solved directly using standard matrix techniques. However, Levinson developed a much more efficient approach that exploits the Toeplitz symmetry of the autocorrelation matrix.

**See Function:** levinson().

**The Wiener Prediction Filter.** Prediction theory has many applications ranging from digital data compression to military gunfire control. The Wiener–Hopf equation can be extended to include the prediction of future values of a signal. Suppose we are interested in estimating the future value of a signal at time  $t + k/f_s$ , where  $f_s$  is the sampling rate. The **Wiener prediction filter** is obtained by solving the following modified set of normal equations:

$$\begin{bmatrix} r_{xx}(0) & r_{xx}(1) & r_{xx}(2) & \cdots & r_{xx}(N-1) \\ r_{xx}(1) & r_{xx}(0) & r_{xx}(1) & \cdots & r_{xx}(N-2) \\ r_{xx}(2) & r_{xx}(1) & r_{xx}(0) & \cdots & r_{xx}(N-3) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{xx}(N-1) & r_{xx}(N-2) & r_{xx}(N-3) & \cdots & r_{xx}(0) \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \\ h_{N-1} \end{bmatrix} = \begin{bmatrix} r_{xx}(0+k) \\ r_{xx}(1+k) \\ r_{xx}(2+k) \\ \vdots \\ r_{xx}(N-1+k) \end{bmatrix}$$

Note the only change is in the constraining vector  $R_{xx}(m+k)$ , and that for  $k = 0$ , the above equation simplifies to the original Wiener filter problem.

**Example Program: Design a Wiener prediction filter for a sine wave.**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mathlib.h"
main()
{
Real_Vector signal, sig_noise, filter, sig_covar, covar_ahead, y;
int i, ny, predict_time = 4; /* desired positive prediction time in samples */

    (continued on next page)
```

(continued)

```

double noise, amplitude = 50.0;
unsigned n=128,nfilt=16;
Real w[16];

/* main program designs a Wiener prediction filter to estimate future values of
a signal in background noise */

    signal = valloc(NULL, n);
    sig_noise = valloc(NULL, n);
    covar_ahead = valloc(NULL, nfilt);
    for (i=0; i<n; i++) {
        signal[i] = amplitude*sin(twopi_*i*0.14);
        noise = ((double) rand()/RAND_MAX - 0.5); /* uniform +0.5, -0.5 */
        sig_noise[i] = signal[i] + noise;
    }
    sig_covar = autocor(signal, n, nfilt+predict_time);
    for(i=0; i<nfilt; i++) covar_ahead[i] = sig_covar[i+predict_time];
    filter = levinson(sig_covar, covar_ahead, nfilt);
    for (i=0; i<nfilt; i++) w[i] = filter[i];
    y = convolve(sig_noise, n, w, nfilt, 1, 1, 0, &ny);
    printf(" true signal      Wiener predicted signal, %d samples ahead\n",
           predict_time);
    for (i=predict_time; i<predict_time+17; i++) {
        printf("%2d %8.4f      %8.4f\n", i, signal[i-predict_time],y[i]);
    }
    return 0;
}

```

**Program Output:**

|    | true signal | Wiener predicted signal, 4 samples ahead |
|----|-------------|--|
| 4  | 0.0000      | -0.4050                                  |
| 5  | 38.5257     | 38.9296                                  |
| 6  | 49.1144     | 50.0329                                  |
| 7  | 24.0877     | 23.9791                                  |
| 8  | -18.4062    | -18.3570                                 |
| 9  | -47.5528    | -48.0411                                 |
| 10 | -42.2164    | -42.1333                                 |
| 11 | -6.2667     | -6.6494                                  |
| 12 | 34.2274     | 33.8501                                  |
| 13 | 49.9013     | 50.6411                                  |
| 14 | 29.3893     | 30.3072                                  |
| 15 | -12.4345    | -12.1562                                 |
| 16 | -45.2414    | -45.3712                                 |
| 17 | -45.2414    | -44.7656                                 |
| 18 | -12.4345    | -12.6516                                 |
| 19 | 29.3893     | 30.2071                                  |
| 20 | 49.9013     | 49.9910                                  |

**See Function:** levinson().

### 8.5.2 The LMS Algorithm

The goal of all adaptive filtering is to approximate the Wiener–Hopf equation as closely as possible. Even with Levison’s recursion and exact correlation estimates, the direct matrix approach can be computationally intensive. Furthermore, for pseudo-stationary signals, the elements of the correlation matrix are slowly time-varying and cannot be estimated perfectly. The LMS (least mean square) algorithm has several advantages over other adaptive estimation approaches:

1. The iterative LMS minimization formula is much simpler than that of gradient approaches.
2. The LMS method does not require explicit calculation of the correlation matrices.
3. The LMS technique does not require a matrix inversion.

The details of the derivation of the LMS method are not presented here. It is recommended that the interested reader consult the excellent references listed at the end of this section.

The LMS algorithm uses a gradient estimate that is based on the instantaneous error

$$e_k = d_k - y_k$$

where  $d_k$  is the desired response and  $y_k$  is the output of the adaptive filter. The filter update equation is

$$H_{k+1} = H_k + 2u_k e_k X_k$$

where  $H_k$  is the vector of  $N$  filter coefficients at the  $k$ th iteration and  $X_k$  is the vector of input signal values

$$X_k = [x_k, x_{k-1}, \dots, x_{k-N+1}]^T$$

The convergence parameter  $u_k$  is included to smooth the noise problems associated with the instantaneous error estimation. The stable range for  $u_k$  varies with the input signal power  $s^2$

$$u_k = u / (N * s^2), \quad 0 < u < 1$$

where  $N$  is the number of filter coefficients. The normalized convergence parameter  $u$  controls the speed and accuracy of convergence. The goal is to select  $u$  such that the algorithm quickly converges to the Wiener–Hopf solution. Experimentation is often required to avoid choosing  $u$  to be too small or too large.

If the signal power is changing, it is sometimes convenient to allow the signal power estimate to adapt according to

$$\hat{s}_k^2 = a * x_k^2 + (1 - a) * \hat{s}_{k-1}^2$$

where  $0 \leq a < 1$ .

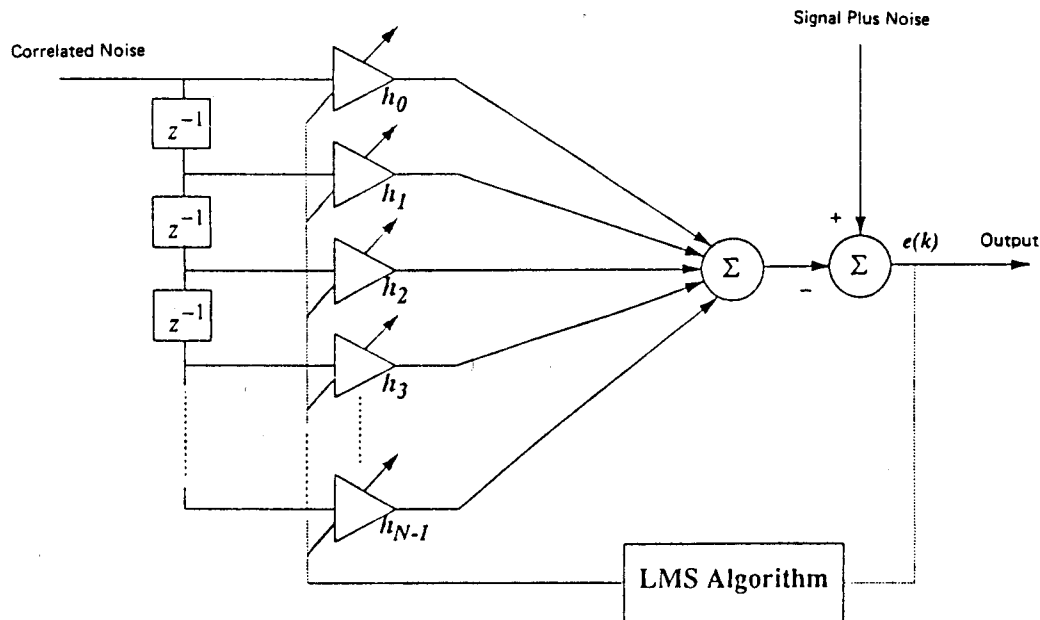
The following sections discuss the application of the LMS algorithm to the noise canceling.

**References.** B. Widrow and S. D. Stearns (1985); S. D. Stearns and R. A. David (1988).

**See Function:** `lmsadapt()`.

### 8.5.3 Adaptive Interference Cancelling

The general block diagram for **adaptive interference cancelling** is shown in Figure 8.13. It is assumed that the signal has been corrupted with noise known to be correlated with a given noise source. Provided the signal is uncorrelated with the noise source, the LMS algorithm may be used to enhance the desired signal. Note that the output of the adaptive filter is a best estimate of the noise that corrupts the signal of interest. A program that demonstrates the use of the LMS algorithm for interference canceling is given along with the description of the *lmsadapt* function.



**Figure 8.13** Adaptive Interference Cancelling

**See Function:** `lmsadapt()`.

### 8.5.4 Adaptive Line Enhancement

Unlike interference canceling, the adaptive filter output is an estimate of the desired signal for **adaptive line enhancement**. The general approach to adaptive line enhancement is illustrated in Figure 8.14. The desired signal may be wideband, and is shown in this figure to be a finite sum of sinusoids. A constant delay is chosen so that the noise before the delay block is uncorrelated with the noise that is input to the adaptive filter. A one-sample delay is adequate for white noise. For bandlimited noise, a delay of at least the reciprocal of the bandwidth (i. e., at least two samples) is required.

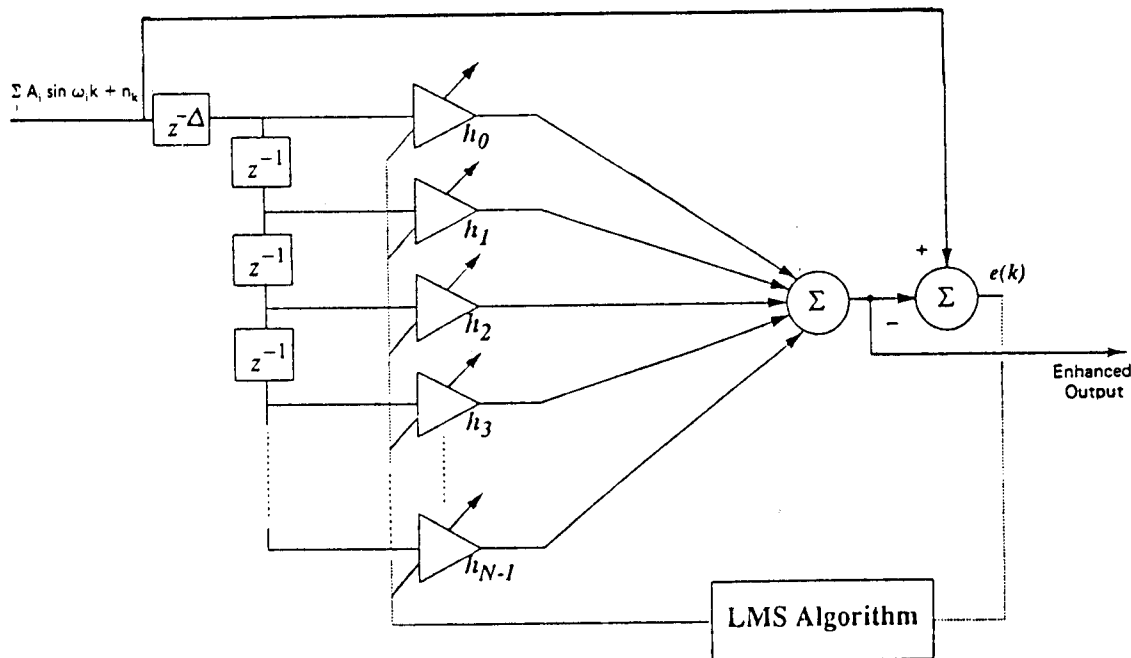


Figure 8.14 Adaptive Line Enhancement

**Example Program: Design an Adaptive Line Enhancer for a sine wave.**

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mathlib.h"
main()
{
Real_Vector desired, source, corrupted, error;
int k, ndelays=20, ndata=250;
double err_before, err_after, freq, tmp, rms_desired, noise;
double mu = 0.01, sig_power = 0.5, rho = 0.0;

/* Adaptive Line Enhancement with the LMS algorithm : Given a sinusoid in
unknown (white) noise, cancel the unknown noise and enhance the sinusoid */

source = valloc(NULL, ndata);
corrupted = valloc(NULL, ndata);
desired = valloc(NULL, ndata);

freq = 2*pi_/30.; /* for this example, the desired signal is a sinusoid */
source[0] = 0.0;
for(k = 0; k<ndata-1; k++) {
noise = 4.0*((double) rand()/RAND_MAX - 0.5);
desired[k] = sin(freq*k);
corrupted[k] = desired[k] + noise;
source[k+1] = corrupted[k];
}
}

```

(continued on next page)

(continued)

```

    }
    /* a delay of one sample is enough for white noise to decorrelate      */
    /* at least a two sample delay would be needed for bandlimited noise    */
    noise = 4.0*((double) rand()/RAND_MAX - 0.5);
    desired[k] = sin(freq*(ndata-1));
    corrupted[k] = desired[ndata-1] + noise;

    /* enhance the sinusoid with the LMS algorithm: */
    error = lmsadapt(source, corrupted, ndata, ndelays, mu, rho, sig_power);

    err_after = err_before = rms_desired = 0.0;
    for(k = 0; k<ndata; k++) {
        tmp = desired[k] - corrupted[k];
        err_before += tmp*tmp;
        tmp = desired[k] - source[k];
        err_after += tmp*tmp;
        rms_desired += desired[k]*desired[k];
    }
    printf("error before LMS filtering = %.4f\n", err_before/rms_desired);
    printf("error after LMS filtering = %.4f\n", err_after/rms_desired);
    mathfree();
    return 0;
}

```

**Program Output:**

```

error before LMS filtering = 2.6136
error after LMS filtering = 0.4735

```

**See Function:** lmsadapt().