CHAPTER

14

Introduction to Digital Filters

Digital filters are used for two general purposes: (1) separation of signals that have been combined, and (2) restoration of signals that have been distorted in some way. Analog (electronic) filters can be used for these same tasks; however, digital filters can achieve far superior results. The most popular digital filters are described and compared in the next seven chapters. This introductory chapter describes the parameters you want to look for when learning about each of these filters.

Filter Basics

Digital filters are a very important part of DSP. In fact, their extraordinary performance is one of the key reasons that DSP has become so popular. As mentioned in the introduction, filters have two uses: signal *separation* and signal *restoration*. Signal separation is needed when a signal has been contaminated with interference, noise, or other signals. For example, imagine a device for measuring the electrical activity of a baby's heart (EKG) while still in the womb. The raw signal will likely be corrupted by the breathing and heartbeat of the mother. A filter might be used to separate these signals so that they can be individually analyzed.

Signal restoration is used when a signal has been distorted in some way. For example, an audio recording made with poor equipment may be filtered to better represent the sound as it actually occurred. Another example is the deblurring of an image acquired with an improperly focused lens, or a shaky camera.

These problems can be attacked with either analog or digital filters. Which is better? Analog filters are cheap, fast, and have a large dynamic range in both amplitude and frequency. Digital filters, in comparison, are vastly superior in the level of performance that can be achieved. For example, a low-pass digital filter presented in Chapter 16 has a gain of 1 +/- 0.0002 from DC to 1000 hertz, and a gain of less than 0.0002 for frequencies above

1001 hertz. The entire transition occurs within only 1 hertz. Don't expect this from an op amp circuit! Digital filters can achieve *thousands* of times better performance than analog filters. This makes a dramatic difference in how filtering problems are approached. With analog filters, the emphasis is on handling limitations of the electronics, such as the accuracy and stability of the resistors and capacitors. In comparison, digital filters are so good that the performance of the filter is frequently ignored. The emphasis shifts to the limitations of the *signals* and the *theoretical* issues regarding their processing.

It is common in DSP to say that a filter's input and output signals are in the *time domain*. This is because signals are usually created by sampling at regular intervals of *time*. But this is not the only way sampling can take place. The second most common way of sampling is at equal intervals in *space*. For example, imagine taking simultaneous readings from an array of strain sensors mounted at one centimeter increments along the length of an aircraft wing. Many other domains are possible; however, time and space are by far the most common. When you see the term *time domain* in DSP, remember that it may actually refer to samples taken over time, or it may be a general reference to any domain that the samples are taken in.

As shown in Fig. 14-1, every linear filter has an **impulse response**, a **step response** and a **frequency response.** Each of these responses contains complete information about the filter, but in a different form. If one of the three is specified, the other two are fixed and can be directly calculated. All three of these representations are important, because they describe how the filter will react under different circumstances.

The most straightforward way to implement a digital filter is by *convolving* the input signal with the digital filter's *impulse response*. All possible linear filters can be made in this manner. (This should be obvious. If it isn't, you probably don't have the background to understand this section on filter design. Try reviewing the previous section on DSP fundamentals.) When the *impulse response* is used in this way, filter designers give it a special name: the **filter kernel**.

There is also another way to make digital filters, called **recursion**. When a filter is implemented by convolution, each sample in the output is calculated by *weighting* the samples in the input, and adding them together. Recursive filters are an extension of this, using previously calculated values from the *output*, besides points from the *input*. Instead of using a filter kernel, recursive filters are defined by a set of **recursion coefficients**. This method will be discussed in detail in Chapter 19. For now, the important point is that all linear filters have an impulse response, even if you don't use it to implement the filter. To find the impulse response of a recursive filter, simply feed in an impulse, and see what comes out. The impulse responses of recursive filters are composed of sinusoids that exponentially decay in amplitude. In principle, this makes their impulse responses *infinitely long*. However, the amplitude eventually drops below the round-off noise of the system, and the remaining samples can be ignored. Because

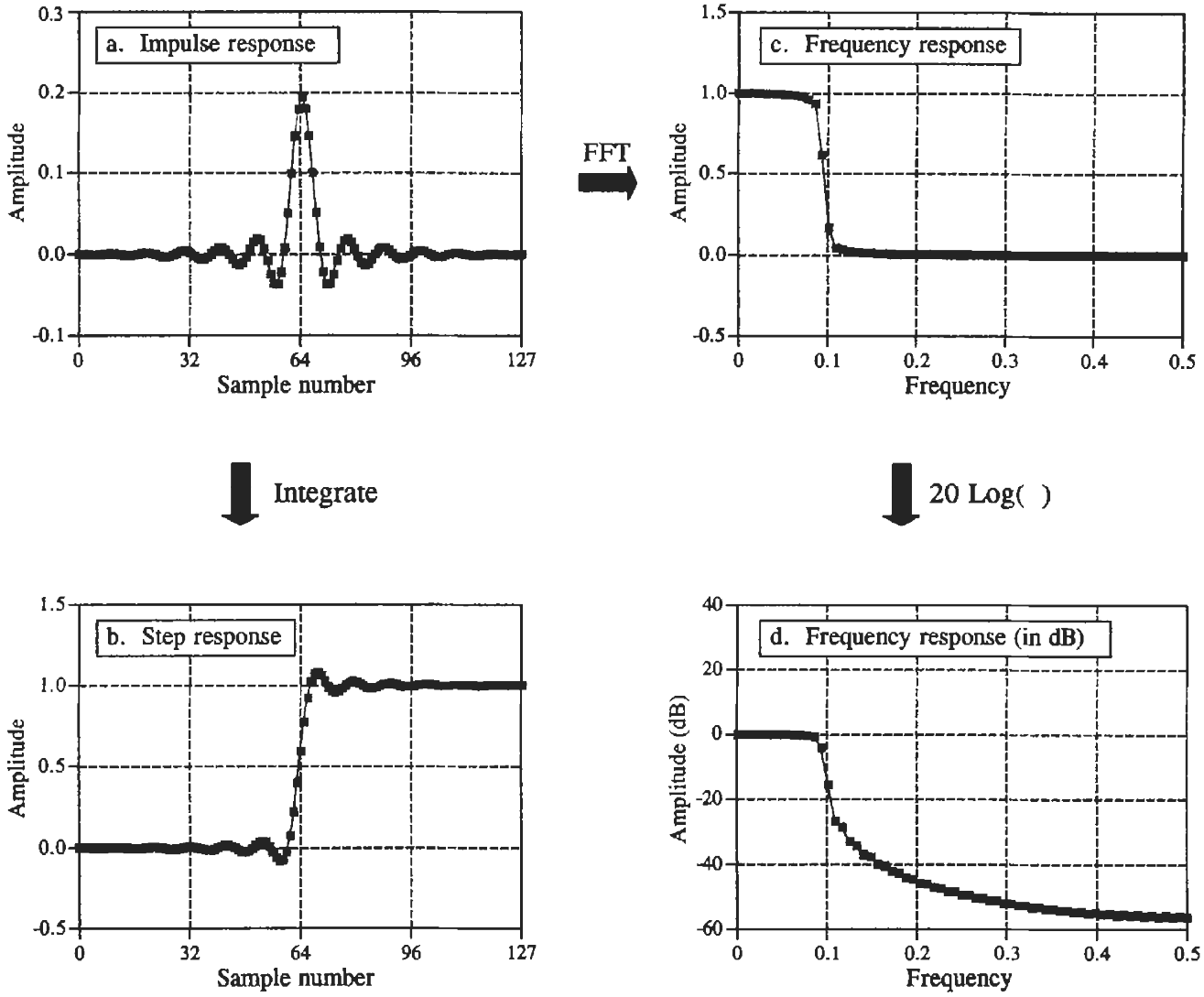

FIGURE 14-1
Filter parameters. Every linear filter has an impulse response, a step response, and a frequency response. The step response, (b), can be found by discrete integration of the impulse response, (a). The frequency response can be found from the impulse response by using the Fast Fourier Transform (FFT), and can be displayed either on a linear scale, (c), or in decibels, (d).

of this characteristic, recursive filters are also called **Infinite Impulse Response or IIR** filters. In comparison, filters carried out by convolution are called **Finite Impulse Response or FIR** filters.

As you know, the *impulse response* is the output of a system when the input is an *impulse*. In this same manner, the *step response* is the output when the input is a *step* (also called an *edge*, and an *edge response*). Since the step is the integral of the impulse, the step response is the integral of the impulse response. This provides two ways to find the step response: (1) feed a step waveform into the filter and see what comes out, or (2) integrate the impulse response. (To be mathematically correct: *integration* is used with continuous signals, while *discrete integration*, i.e., a running sum, is used with discrete signals.) The frequency response can be found by taking the DFT (using the FFT algorithm) of the impulse response. This will be reviewed later in this

chapter. The frequency response can be plotted on a linear vertical axis, such as in (c), or on a logarithmic scale (decibels), as shown in (d). The linear scale is best at showing the passband ripple and roll-off, while the decibel scale is needed to show the stopband attenuation.

Don't remember decibels? Here is a quick review. A **bel** (in honor of Alexander Graham Bell) means that the power is changed by a *factor of ten.* For example, an electronic circuit that has 3 bels of amplification produces an output signal with $10 \times 10 \times 10 = 1000$ times the power of the input. A **decibel (dB)** is one-tenth of a bel. Therefore, the decibel values of: -20dB, -10dB, 0dB, 10dB and 20dB mean the power ratios: 0.01, 0.1, 1, 10, and 100, respectively. In other words, every *ten* decibels mean that the power has changed by a factor of ten.

Here's the catch: you usually want to work with a signal's *amplitude*, not its *power*. For example, imagine an amplifier with 20dB of gain. By definition, this means that the power in the signal has increased by a factor of 100. Since amplitude is proportional to the square-root of power, the amplitude of the output is 10 times the amplitude of the input. While 20dB means a factor of 100 in power, it only means a factor of 10 in amplitude. Every *twenty* decibels mean that the amplitude has changed by a factor of ten. In equation form:

EQUATION 14-1
Definition of decibels. Decibels are a way of expressing a *ratio* between two signals. Ratios of power (P_1 & P_2) use a different equation from ratios of amplitude (A_1 & A_2).

$$\text{dB} = 10 \log_{10} \frac{P_2}{P_1}$$

$$\text{dB} = 20 \log_{10} \frac{A_2}{A_1}$$

The above equations use the base 10 logarithm; however, many computer languages only provide a function for the base *e* logarithm (the natural log, written $\log_e x$ or $\ln x$). The natural log can be used by modifying the above equations: $\text{dB} = 4.342945 \log_e(P_2/P_1)$ and $\text{dB} = 8.685890 \log_e(A_2/A_1)$.

Since decibels are a way of expressing the ratio between two signals, they are ideal for describing the gain of a system, i.e., the ratio between the output and the input signal. However, engineers also use decibels to specify the amplitude (or power) of a *single* signal, by referencing it to some standard. For example, the term: **dBV** means that the signal is being referenced to a 1 volt rms signal. Likewise, **dBm** indicates a reference signal producing 1 mW into a 600 ohms load (about 0.78 volts rms).

If you understand nothing else about decibels, remember two things: First, –3dB means that the amplitude is reduced to 0.707 (and the power is

therefore reduced to 0.5). Second, memorize the following conversions between decibels and *amplitude* ratios:

60dB	=	1000
40dB	=	100
20dB	=	10
0dB	=	1
-20dB	=	0.1
-40dB	=	0.01
-60dB	=	0.001

How Information is Represented in Signals

The most important part of any DSP task is understanding how *information* is contained in the signals you are working with. There are many ways that information can be contained in a signal. This is especially true if the signal is manmade. For instance, consider all of the modulation schemes that have been devised: AM, FM, single-sideband, pulse-code modulation, pulse-width modulation, etc. The list goes on and on. Fortunately, there are only two ways that are common for information to be represented in naturally occurring signals. We will call these: **information represented in the time domain**, and **information represented in the frequency domain.**

Information represented in the time domain describes when something occurs and what the amplitude of the occurrence is. For example, imagine an experiment to study the light output from the sun. The light output is measured and recorded once each second. Each sample in the signal indicates what is happening at that instant, and the level of the event. If a solar flare occurs, the signal directly provides information on the time it occurred, the duration, the development over time, etc. Each sample contains information that is interpretable without reference to any other sample. Even if you have only one sample from this signal, you still know something about what you are measuring. This is the simplest way for information to be contained in a signal.

In contrast, information represented in the frequency domain is more indirect. Many things in our universe show periodic motion. For example, a wine glass struck with a fingernail will vibrate, producing a ringing sound; the pendulum of a grandfather clock swings back and forth; stars and planets rotate on their axis and revolve around each other, and so forth. By measuring the frequency, phase, and amplitude of this periodic motion, information can often be obtained about the system producing the motion. Suppose we sample the sound produced by the ringing wine glass. The fundamental frequency and harmonics of the periodic vibration relate to the mass and elasticity of the material. A single sample, in itself, contains no information about the periodic motion, and therefore no information about the wine glass. The information is contained in the *relationship* between many points in the signal.

This brings us to the importance of the step and frequency responses. The *step response* describes how information represented in the *time domain* is being modified by the system. In contrast, the *frequency response* shows how information represented in the *frequency domain* is being changed. This distinction is absolutely critical in filter design because it is not possible to optimize a filter for both applications. Good performance in the time domain results in poor performance in the frequency domain, and vice versa. If you are designing a filter to remove noise from an EKG signal (information represented in the time domain), the step response is the important parameter, and the frequency response is of little concern. If your task is to design a digital filter for a hearing aid (with the information in the frequency domain), the frequency response is all important, while the step response doesn't matter. Now let's look at what makes a filter optimal for time domain or frequency domain applications.

Time Domain Parameters

It may not be obvious why the step response is of such concern in time domain filters. You may be wondering why the impulse response isn't the important parameter. The answer lies in the way that the human mind understands and processes information. Remember that the step, impulse and frequency responses all contain identical information, just in different arrangements. The step response is useful in time domain analysis because it matches the way humans view the information contained in the signals.

For example, suppose you are given a signal of some unknown origin and asked to analyze it. The first thing you will do is divide the signal into regions of similar characteristics. You can't stop from doing this; your mind will do it automatically. Some of the regions may be smooth; others may have large amplitude peaks; others may be noisy. This segmentation is accomplished by identifying the points that separate the regions. This is where the step function comes in. The step function is the purest way of representing a division between two dissimilar regions. It can mark when an event starts, or when an event ends. It tells you that whatever is on the *left* is somehow different from whatever is on the *right*. This is how the human mind views time domain information: a group of step functions dividing the information into regions of similar characteristics. The step response, in turn, is important because it describes how the dividing lines are being modified by the filter.

The step response parameters that are important in filter design are shown in Fig. 14-2. To distinguish events in a signal, the duration of the step response must be shorter than the spacing of the events. This dictates that the step response should be as *fast* (the DSP jargon) as possible. This is shown in Figs. (a) and (b). The most common way to specify the **risetime** (more jargon) is to quote the number of samples between the 10% and 90% amplitude levels. Why isn't a very fast risetime always possible? There are many reasons, noise reduction, inherent limitations of the data acquisition system, avoiding aliasing, etc.

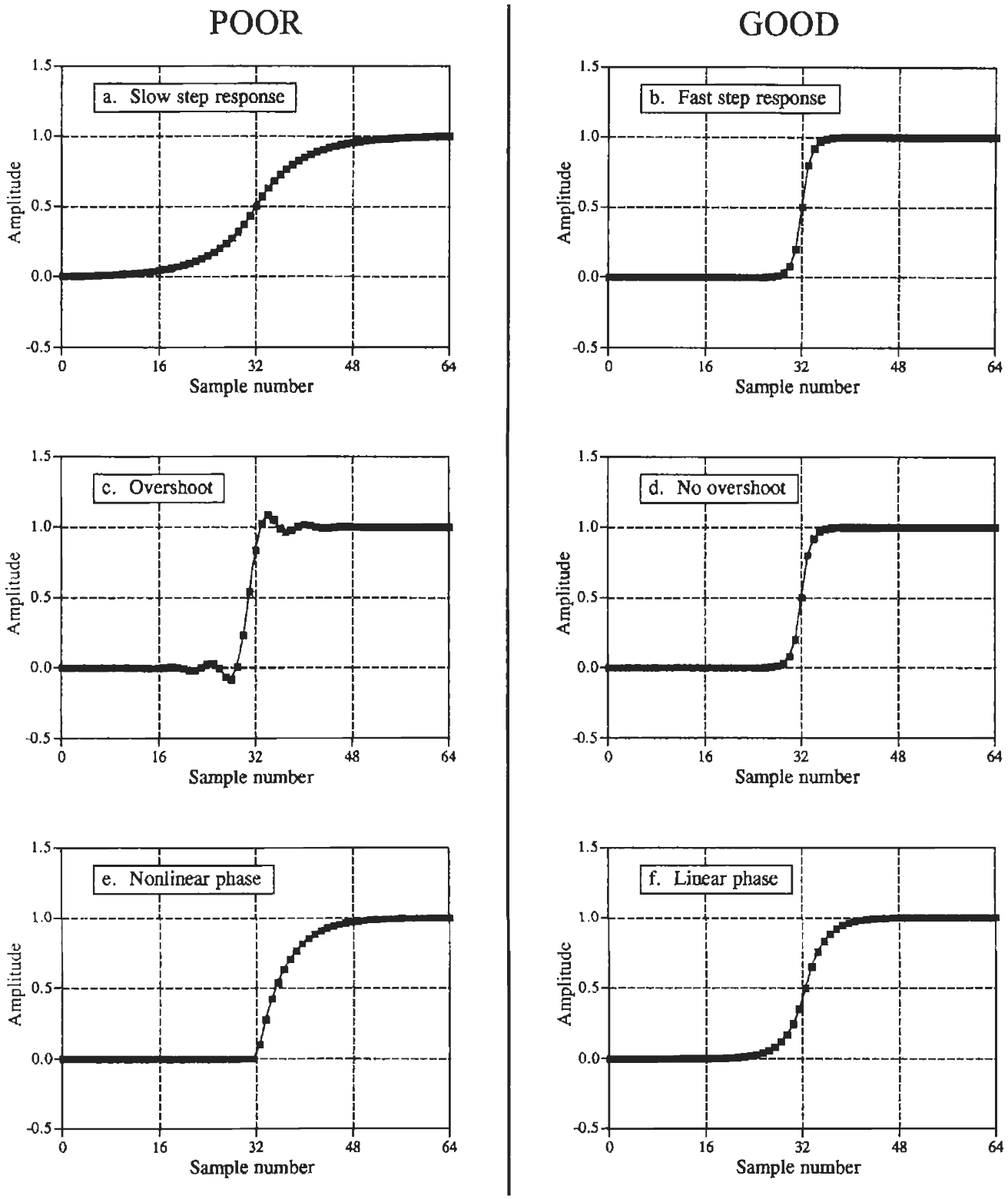

FIGURE 14-2
Parameters for evaluating *time domain* performance. The step response is used to measure how well a filter performs in the time domain. Three parameters are important: (1) transition speed (risetime), shown in (a) and (b), (2) overshoot, shown in (c) and (d), and (3) phase linearity (symmetry between the top and bottom halves of the step), shown in (e) and (f).

Figures (c) and (d) show the next parameter that is important: **overshoot** in the step response. Overshoot must generally be eliminated because it changes the amplitude of samples in the signal; this is a basic distortion of the information contained in the time domain. This can be summed up in

one question: Is the overshoot you observe in a signal coming from the thing you are trying to measure, or from the filter you have used?

Finally, it is often desired that the upper half of the step response be symmetrical with the lower half, as illustrated in (e) and (f). This symmetry is needed to make the *rising edges* look the same as the *falling edges*. This symmetry is called **linear phase**, because the frequency response has a phase that is a straight line (discussed in Chapter 19). Make sure you understand these three parameters; they are the key to evaluating time domain filters.

Frequency Domain Parameters

Figure 14-3 shows the four basic frequency responses. The purpose of these filters is to allow some frequencies to pass unaltered, while completely blocking other frequencies. The **passband** refers to those frequencies that are passed, while the **stopband** contains those frequencies that are blocked. The **transition band** is between. A **fast roll-off** means that the transition band is very narrow. The division between the passband and transition band is called the **cutoff frequency.** In analog filter design, the cutoff frequency is usually defined to be where the amplitude is reduced to 0.707 (i.e., -3dB). Digital filters are less standardized, and it is common to see 99%, 90%, 70.7%, and 50% amplitude levels defined to be the cutoff frequency.

Figure 14-4 shows three parameters that measure how well a filter performs in the frequency domain. To separate closely spaced frequencies, the filter must have a **fast roll-off**, as illustrated in (a) and (b). For the passband frequencies to move through the filter unaltered, there must be no **passband ripple**, as shown in (c) and (d). Lastly, to adequately block the stopband frequencies, it is necessary to have good **stopband attenuation**, displayed in (e) and (f).

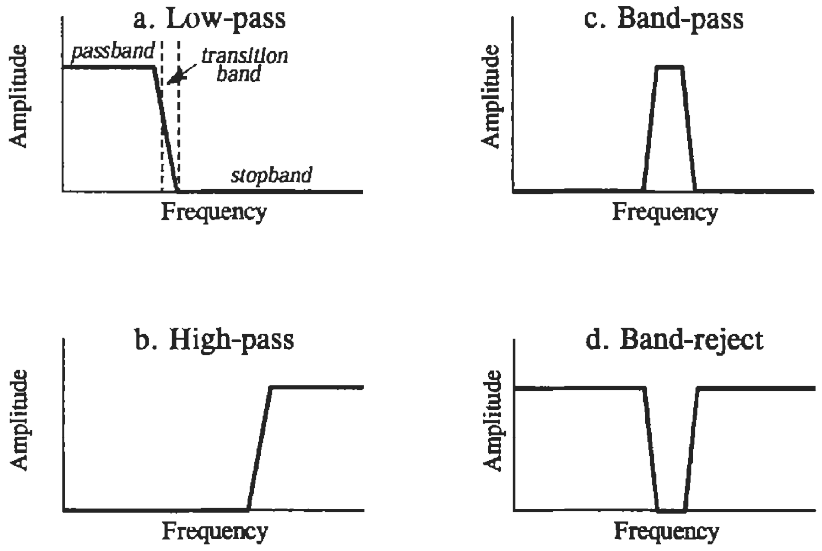

FIGURE 14-3
The four common frequency responses. Frequency domain filters are generally used to pass certain frequencies (the *passband*), while blocking others (the *stopband*). Four responses are the most common: low-pass, high-pass, band-pass, and band-reject.

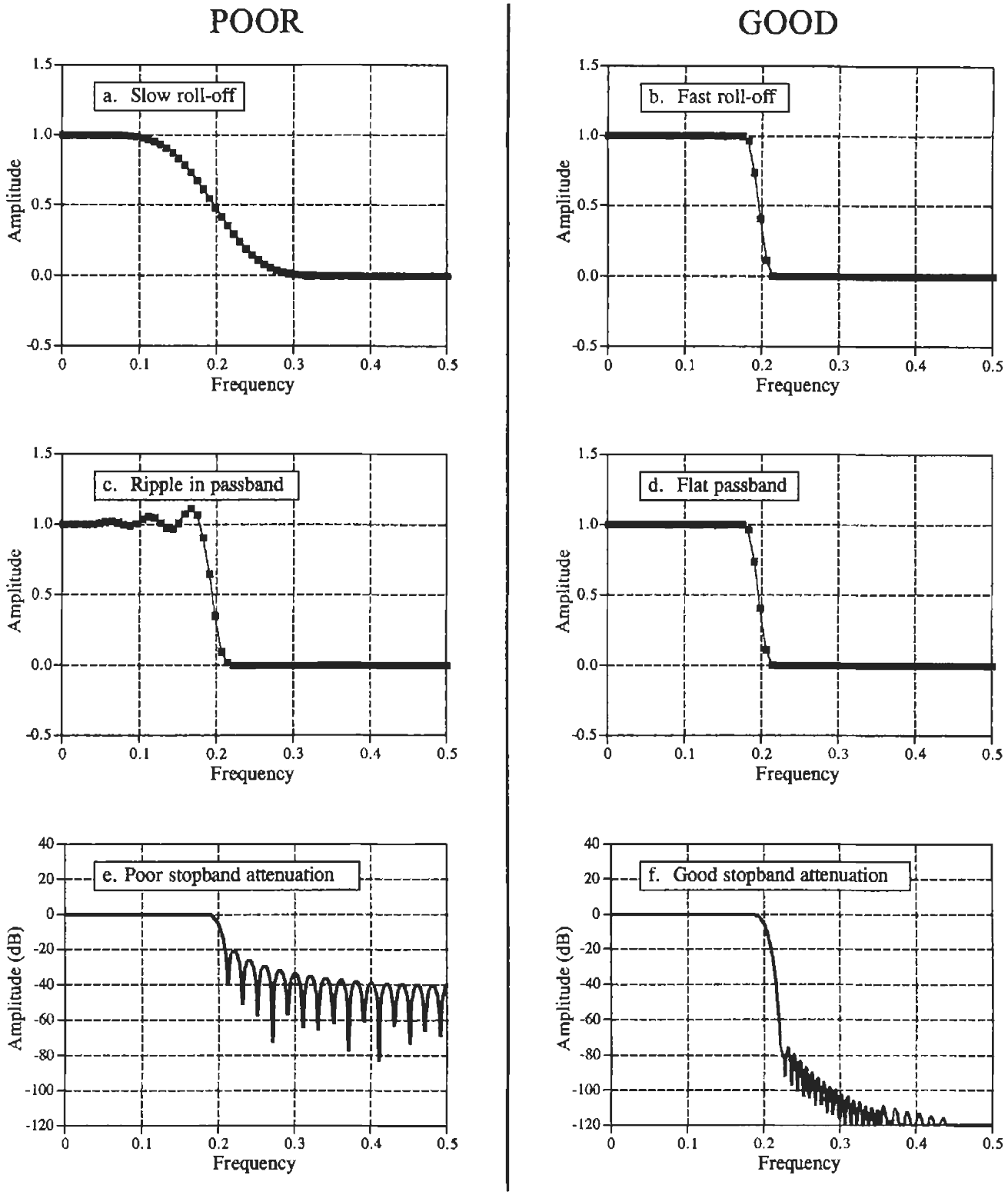

FIGURE 14-4
Parameters for evaluating *frequency domain* performance. The frequency responses shown are for low-pass filters. Three parameters are important: (1) roll-off sharpness, shown in (a) and (b), (2) passband ripple, shown in (c) and (d), and (3) stopband attenuation, shown in (e) and (f).

Why is there nothing about the *phase* in these parameters? First, the phase isn't important in most frequency domain applications. For example, the phase of an audio signal is almost completely random, and contains little useful information. Second, if the phase is important, it is very easy to make digital

filters with a *perfect* phase response, i.e., all frequencies pass through the filter with a zero phase shift (also discussed in Chapter 19). In comparison, analog filters are ghastly in this respect.

Previous chapters have described how the DFT converts a system's impulse response into its frequency response. Here is a brief review. The quickest way to calculate the DFT is by means of the FFT algorithm presented in Chapter 12. Starting with a filter kernel N samples long, the FFT calculates the frequency spectrum consisting of an N point *real part* and an N point *imaginary part*. Only samples 0 to $N/2$ of the FFT's real and imaginary parts contain useful information; the remaining points are duplicates (negative frequencies) and can be ignored. Since the real and imaginary parts are difficult for humans to understand, they are usually converted into polar notation as described in Chapter 8. This provides the magnitude and phase signals, each running from sample 0 to sample $N/2$ (i.e., $N/2+1$ samples in each signal). For example, an impulse response of 256 points will result in a frequency response running from point 0 to 128. Sample 0 represents DC, i.e., zero frequency. Sample 128 represents one-half of the sampling rate. Remember, no frequencies higher than one-half of the sampling rate can appear in sampled data.

The number of samples used to represent the impulse response can be arbitrarily large. For instance, suppose you want to find the frequency response of a filter kernel that consists of 80 points. Since the FFT only works with signals that are a power of two, you need to add 48 zeros to the signal to bring it to a length of 128 samples. This *padding with zeros* does not change the impulse response. To understand why this is so, think about what happens to these added zeros when the input signal is convolved with the system's impulse response. The added zeros simply *vanish* in the convolution, and do not affect the outcome.

Taking this a step further, you could add *many* zeros to the impulse response to make it, say, 256, 512, or 1024 points long. The important idea is that longer impulse responses result in a closer spacing of the data points in the frequency response. That is, there are more samples spread between DC and one-half of the sampling rate. Taking this to the extreme, if the impulse response is padded with an *infinite* number of zeros, the data points in the frequency response are infinitesimally close together, i.e., a continuous line. In other words, the frequency response of a filter is really a *continuous* signal between DC and one-half of the sampling rate. The output of the DFT is a *sampling* of this continuous line. What length of impulse response should you use when calculating a filter's frequency response? As a first thought, try $N=1024$, but don't be afraid to change it if needed (such as insufficient resolution or excessive computation time).

Keep in mind that the "good" and "bad" parameters discussed in this chapter are only generalizations. Many signals don't fall neatly into categories. For example, consider an EKG signal contaminated with 60-hertz interference. The information is encoded in the *time domain*, but the interference is best dealt with in the *frequency domain*. The best design for this application is

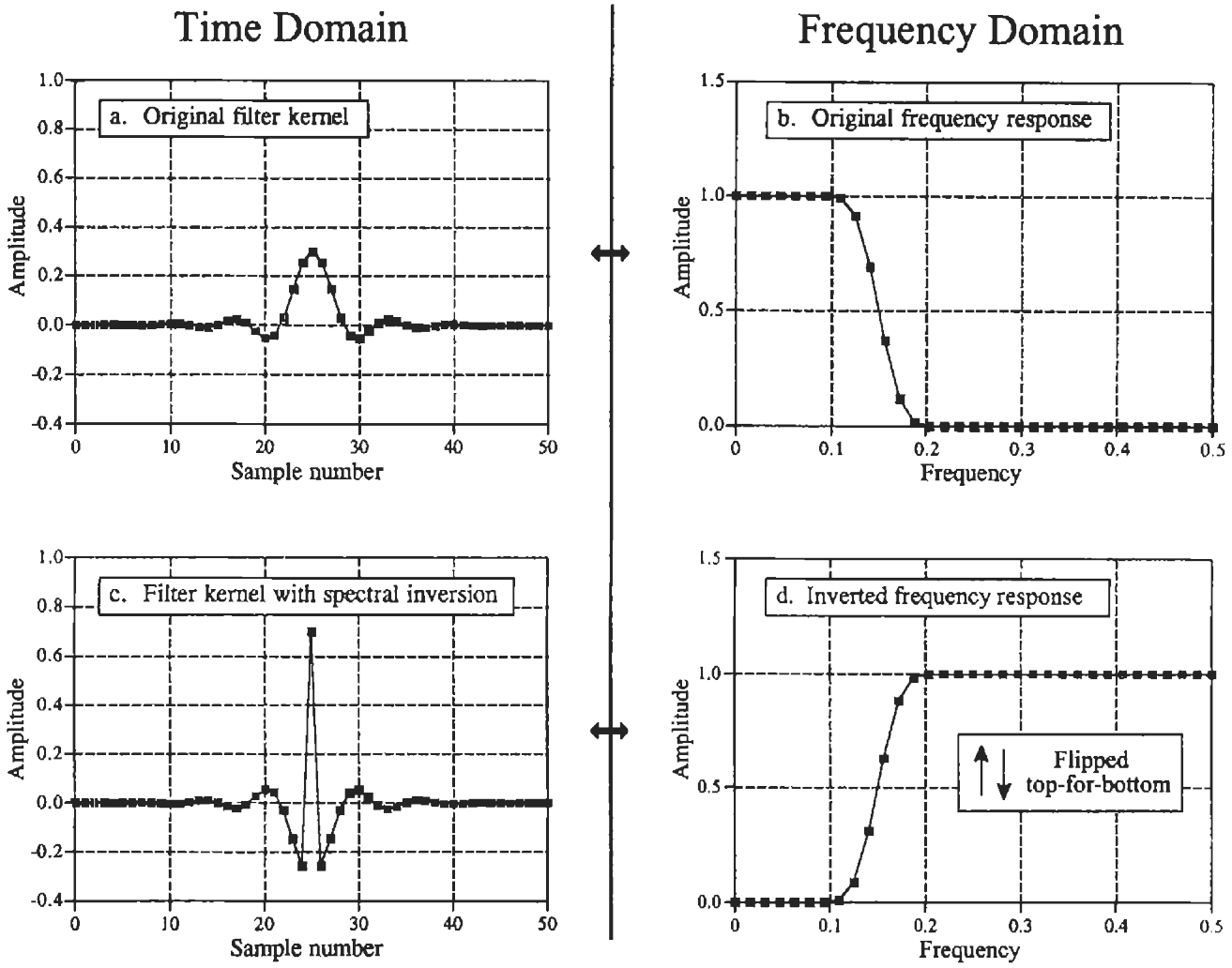

FIGURE 14-5
Example of spectral inversion. The low-pass filter kernel in (a) has the frequency response shown in (b). A high-pass filter kernel, (c), is formed by changing the sign of each sample in (a), and adding one to the sample at the center of symmetry. This action in the time domain *inverts* the frequency spectrum (i.e., flips it top-for-bottom), as shown by the high-pass frequency response in (d).

bound to have trade-offs, and might go against the conventional wisdom of this chapter. Remember the number one rule of education: *A paragraph in a book doesn't give you a license to stop thinking.*

High-Pass, Band-Pass and Band-Reject Filters

High-pass, band-pass and band-reject filters are designed by starting with a low-pass filter, and then converting it into the desired response. For this reason, most discussions on filter design only give examples of low-pass filters. There are two methods for the low-pass to high-pass conversion: **spectral inversion** and **spectral reversal**. Both are equally useful.

An example of *spectral inversion* is shown in 14-5. Figure (a) shows a low-pass filter kernel called a windowed-sinc (the topic of Chapter 16). This filter kernel is 51 points in length, although many of samples have a value so small that they appear to be zero in this graph. The corresponding

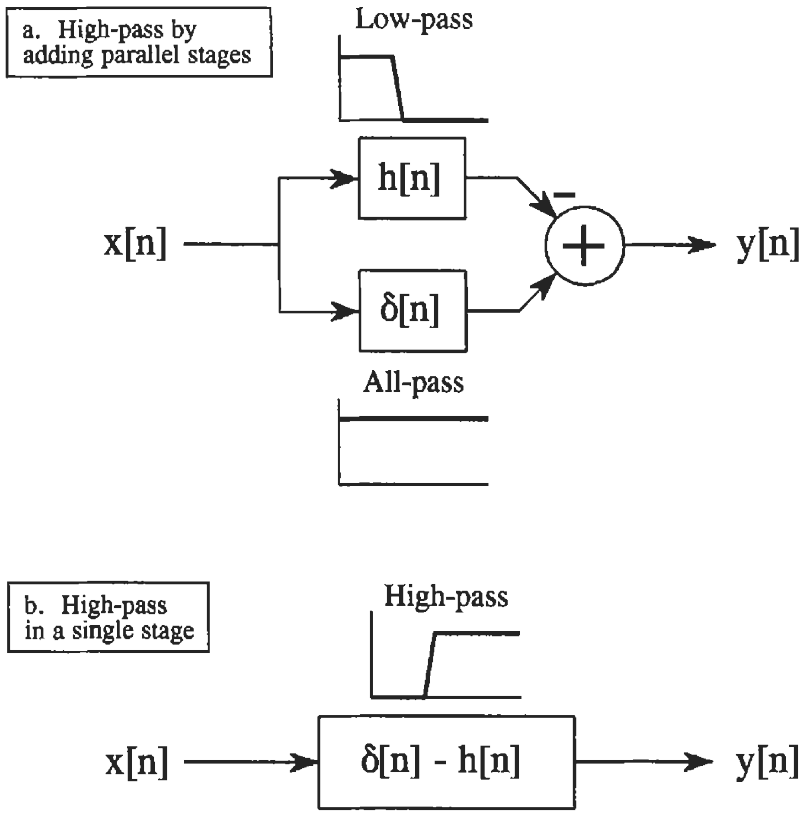

FIGURE 14-6
Block diagram of spectral inversion. In (a), the input signal, $x[n]$, is applied to two systems in parallel, having impulse responses of $h[n]$ and $\delta[n]$. As shown in (b), the combined system has an impulse response of $\delta[n] - h[n]$. This means that the frequency response of the combined system is the *inversion* of the frequency response of $h[n]$.

frequency response is shown in (b), found by adding 13 zeros to the filter kernel and taking a 64 point FFT. Two things must be done to change the low-pass filter kernel into a high-pass filter kernel. First, change the sign of each sample in the filter kernel. Second, add *one* to the sample at the center of symmetry. This results in the high-pass filter kernel shown in (c), with the frequency response shown in (d). Spectral inversion *flips* the frequency response *top-for-bottom*, changing the passbands into stopbands, and the stopbands into passbands. In other words, it changes a filter from low-pass to high-pass, high-pass to low-pass, band-pass to band-reject, or band-reject to band-pass.

Figure 14-6 shows why this two step modification to the time domain results in an inverted frequency spectrum. In (a), the input signal, $x[n]$, is applied to two systems in parallel. One of these systems is a low-pass filter, with an impulse response given by $h[n]$. The other system does *nothing* to the signal, and therefore has an impulse response that is a delta function, $\delta[n]$. The overall output, $y[n]$, is equal to the output of the all-pass system *minus* the output of the low-pass system. Since the low frequency components are subtracted from the original signal, only the high frequency components appear in the output. Thus, a high-pass filter is formed.

This could be performed as a two step operation in a computer program: run the signal through a low-pass filter, and then subtract the filtered signal from the original. However, the entire operation can be performed in a signal stage by combining the two filter kernels. As described in Chapter

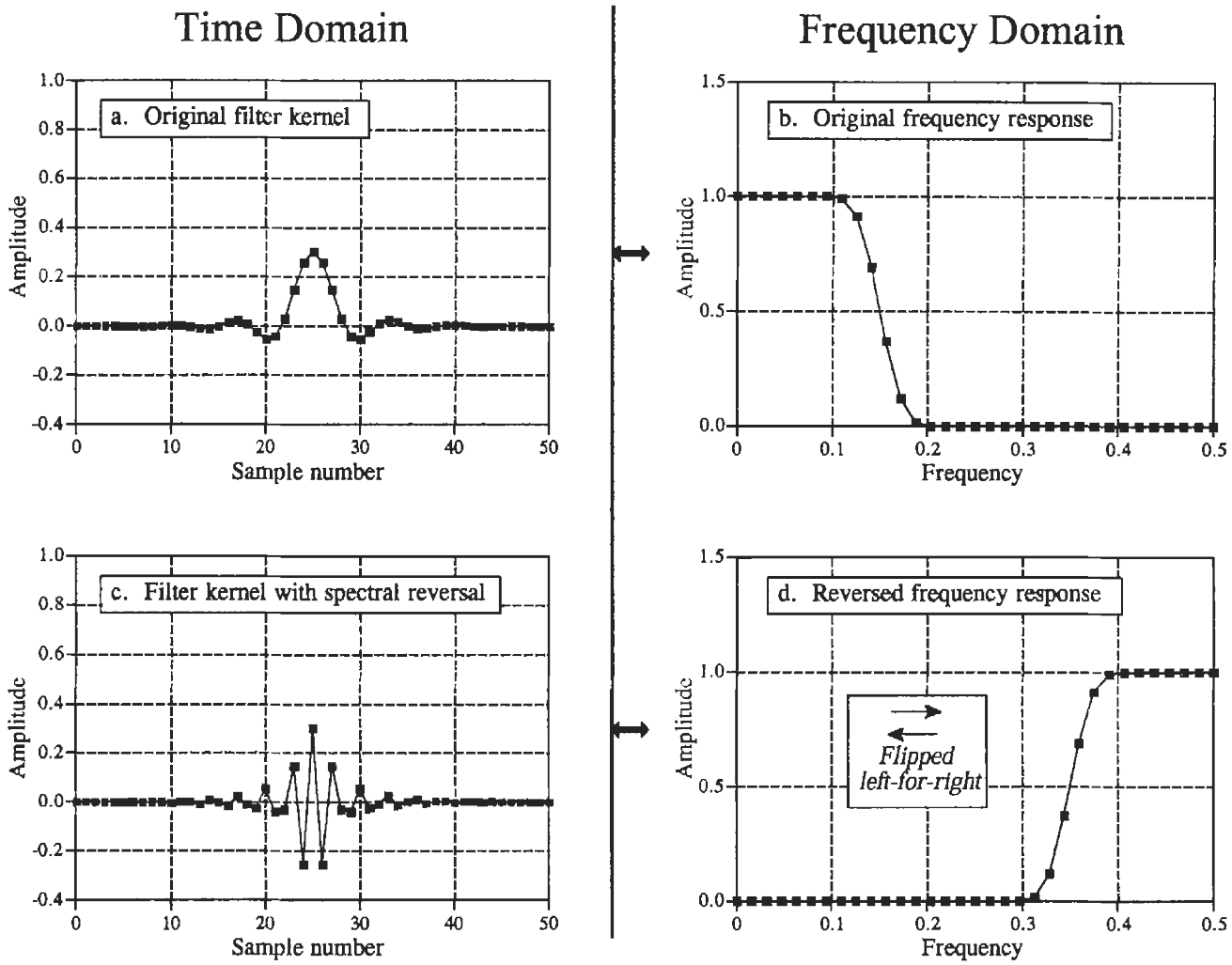

FIGURE 14-7
Example of spectral reversal. The low-pass filter kernel in (a) has the frequency response shown in (b). A high-pass filter kernel, (c), is formed by changing the sign of every other sample in (a). This action in the time domain results in the frequency domain being flipped *left-for-right*, resulting in the high-pass frequency response shown in (d).

7, parallel systems with added outputs can be combined into a single stage by adding their impulse responses. As shown in (b), the filter kernel for the high-pass filter is given by: $\delta[n] - h[n]$. That is, change the sign of all the samples, and then add one to the sample at the center of symmetry.

For this technique to work, the low-frequency components exiting the low-pass filter must have the same phase as the low-frequency components exiting the all-pass system. Otherwise a complete subtraction cannot take place. This places two restrictions on the method: (1) the original filter kernel must have left-right symmetry (i.e., a zero or linear phase), and (2) the impulse must be added at the center of symmetry.

The second method for low-pass to high-pass conversion, *spectral reversal*, is illustrated in Fig. 14-7. Just as before, the low-pass filter kernel in (a) corresponds to the frequency response in (b). The high-pass filter kernel, (c), is formed by *changing the sign of every other sample* in (a). As shown in (d), this flips the frequency domain *left-for-right*: 0 becomes 0.5 and 0.5

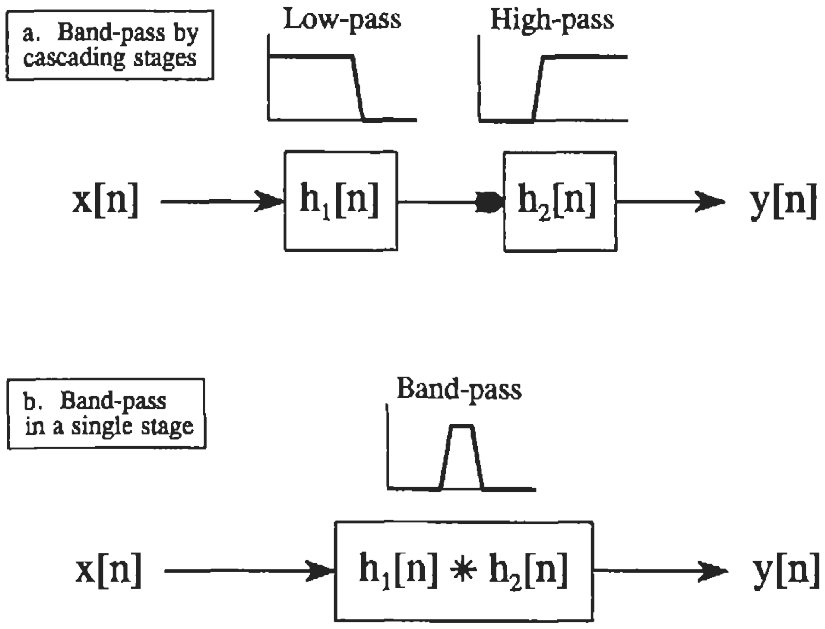

FIGURE 14-8
Designing a band-pass filter. As shown in (a), a band-pass filter can be formed by cascading a low-pass filter and a high-pass filter. This can be reduced to a single stage, shown in (b). The filter kernel of the single stage is equal to the *convolution* of the low-pass and high-pass filter kernels.

becomes 0. The cutoff frequency of the example low-pass filter is 0.15, resulting in the cutoff frequency of the high-pass filter being 0.35.

Changing the sign of every other sample is equivalent to multiplying the filter kernel by a sinusoid with a frequency of 0.5. As discussed in Chapter 10, this has the effect of *shifting* the frequency domain by 0.5. Look at (b) and imagine the negative frequencies between -0.5 and 0 that are of mirror image of the frequencies between 0 and 0.5. The frequencies that appear in (d) are the negative frequencies from (b) shifted by 0.5.

Lastly, Figs. 14-8 and 14-9 show how low-pass and high-pass filter kernels can be combined to form band-pass and band-reject filters. In short, *adding* the filter kernels produces a *band-reject* filter, while *convolving* the filter kernels produces a *band-pass* filter. These are based on the way cascaded and parallel systems are be combined, as discussed in Chapter 7. Multiple combination of these techniques can also be used. For instance, a band-pass filter can be designed by adding the two filter kernels to form a stop-pass filter, and then use *spectral inversion* or *spectral reversal* as previously described. All these techniques work very well with few surprises.

Filter Classification

Table 14-1 summarizes how digital filters are classified by their *use* and by their *implementation*. The use of a digital filter can be broken into three categories: *time domain*, *frequency domain* and *custom*. As previously described, time domain filters are used when the information is encoded in the shape of the signal's waveform. Time domain filtering is used for such actions as: smoothing, DC removal, waveform shaping, etc. In contrast, frequency domain filters are used when the information is contained in the

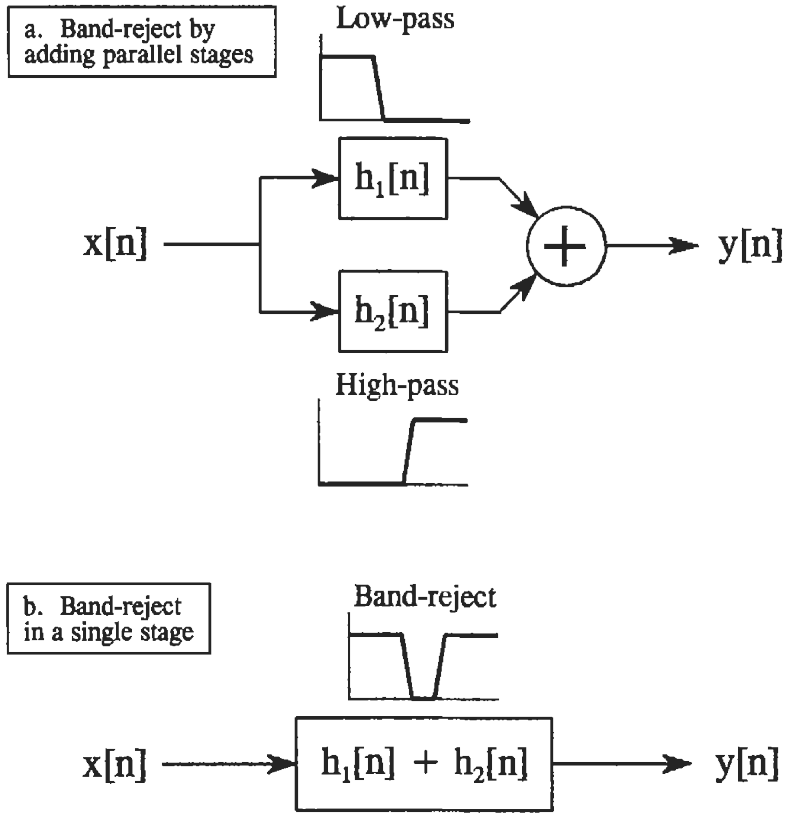

FIGURE 14-9
Designing a band-reject filter. As shown in (a), a band-reject filter is formed by the parallel combination of a low-pass filter and a high-pass filter with their outputs added. Figure (b) shows this reduced to a single stage, with the filter kernel found by *adding* the low-pass and high-pass filter kernels.

amplitude, frequency, and phase of the component sinusoids. The goal of these filters is to separate one band of frequencies from another. Custom filters are used when a special action is required by the filter, something more elaborate than the four basic responses (high-pass, low-pass, band-pass and band-reject). For instance, Chapter 17 describes how custom filters can be used for *deconvolution*, a way of counteracting an unwanted convolution.

FILTER USED FOR:	FILTER IMPLEMENTED BY: Convolution *Finite Impulse Response (FIR)*	Recursion *Infinite Impulse Response (IIR)*
Time Domain *(smoothing, DC removal)*	Moving average (Ch. 15)	Single pole (Ch. 19)
Frequency Domain *(separating frequencies)*	Windowed-sinc (Ch. 16)	Chebyshev (Ch. 20)
Custom *(Deconvolution)*	FIR custom (Ch. 17)	Iterative design (Ch. 26)

TABLE 14-1
Filter classification. Filters can be divided by their *use*, and how they are *implemented*.

Digital filters can be implemented in two ways, by *convolution* (also called *finite impulse response* or *FIR*) and by *recursion* (also called *infinite impulse response* or *IIR*). Filters carried out by convolution can have far better performance than filters using recursion, but execute much more slowly.

The next six chapters describe digital filters according to the classifications in Table 14-1. First, we will look at filters carried out by convolution. The *moving average* (Chapter 15) is used in the time domain, the *windowed-sinc* (Chapter 16) is used in the frequency domain, and *FIR custom* (Chapter 17) is used when something special is needed. To finish the discussion of FIR filters, Chapter 18 presents a technique called FFT convolution. This is an algorithm for increasing the speed of convolution, allowing FIR filters to execute faster.

Next, we look at recursive filters. The *single pole* recursive filter (Chapter 19) is used in the time domain, while the *Chebyshev* (Chapter 20) is used in the frequency domain. Recursive filters having a custom response are designed by *iterative techniques*. For this reason, we will delay their discussion until Chapter 26, where they will be presented with another type of iterative procedure: the neural network.

As shown in Table 14-1, *convolution* and *recursion* are rival techniques; you must use one or the other for a particular application. How do you choose? Chapter 21 presents a head-to-head comparison of the two, in both the time and frequency domains.

CHAPTER

15

Moving Average Filters

The moving average is the most common filter in DSP, mainly because it is the easiest digital filter to understand and use. In spite of its simplicity, the moving average filter is *optimal* for a common task: reducing random noise while retaining a sharp step response. This makes it the premier filter for time domain encoded signals. However, the moving average is the *worst* filter for frequency domain encoded signals, with little ability to separate one band of frequencies from another. Relatives of the moving average filter include the Gaussian, Blackman, and multiple-pass moving average. These have slightly better performance in the frequency domain, at the expense of increased computation time.

Implementation by Convolution

As the name implies, the moving average filter operates by averaging a number of points from the input signal to produce each point in the output signal. In equation form, this is written:

EQUATION 15-1
Equation of the moving average filter. In this equation, $x[\ ]$ is the input signal, $y[\ ]$ is the output signal, and M is the number of points used in the moving average. This equation only uses points on *one side* of the output sample being calculated.

$$y[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i+j]$$

Where $x[\ ]$ is the input signal, $y[\ ]$ is the output signal, and M is the number of points in the average. For example, in a 5-point moving average filter, point 80 in the output signal is given by:

$$y[80] = \frac{x[80] + x[81] + x[82] + x[83] + x[84]}{5}$$

As an alternative, the group of points from the input signal can be chosen *symmetrically* around the output point:

$$y[80] = \frac{x[78] + x[79] + x[80] + x[81] + x[82]}{5}$$

This corresponds to changing the summation in Eq. 15-1 from: $j = 0$ to $M-1$, to: $j = -(M-1)/2$ to $(M-1)/2$. For instance, in a 10-point moving average filter, the index, j, can run from 0 to 11 (one side averaging) or -5 to 5 (symmetrical averaging). Symmetrical averaging requires that M be an *odd* number. Programming is slightly easier with the points on only one side; however, this produces a relative shift between the input and output signals.

You should recognize that the moving average filter is a *convolution* using a very simple filter kernel. For example, a 5-point filter has the filter kernel: $\cdots 0, 0, 1/5, 1/5, 1/5, 1/5, 1/5, 0, 0 \cdots$. That is, the moving average filter is a convolution of the input signal with a *rectangular pulse* having an area of *one*. Table 15-1 shows a program to implement the moving average filter.

```
100 'MOVING AVERAGE FILTER
110 'This program filters 5000 samples with a 101 point moving
120 'average filter, resulting in 4900 samples of filtered data.
130 '
140 DIM X[4999]                    'X[ ] holds the input signal
150 DIM Y[4999]                    'Y[ ] holds the output signal
160 '
170 GOSUB XXXX                     'Mythical subroutine to load X[ ]
180 '
190 FOR I% = 50 TO 4949            'Loop for each point in the output signal
200   Y[I%] = 0                    'Zero, so it can be used as an accumulator
210   FOR J% = -50 TO 50           'Calculate the summation
220     Y[I%] = Y[I%] + X(I%+J%]
230   NEXT J%
240   Y[I%] = Y[I%]/101            'Complete the average by dividing
250 NEXT I%
260 '
270 END
```

TABLE 15-1

Noise Reduction vs. Step Response

Many scientists and engineers feel guilty about using the moving average filter. Because it is so very simple, the moving average filter is often the first thing tried when faced with a problem. Even if the problem is completely solved, there is still the feeling that something more should be done. This situation is truly ironic. Not only is the moving average filter very good for many applications, it is *optimal* for a common problem, reducing random white noise while keeping the sharpest step response.

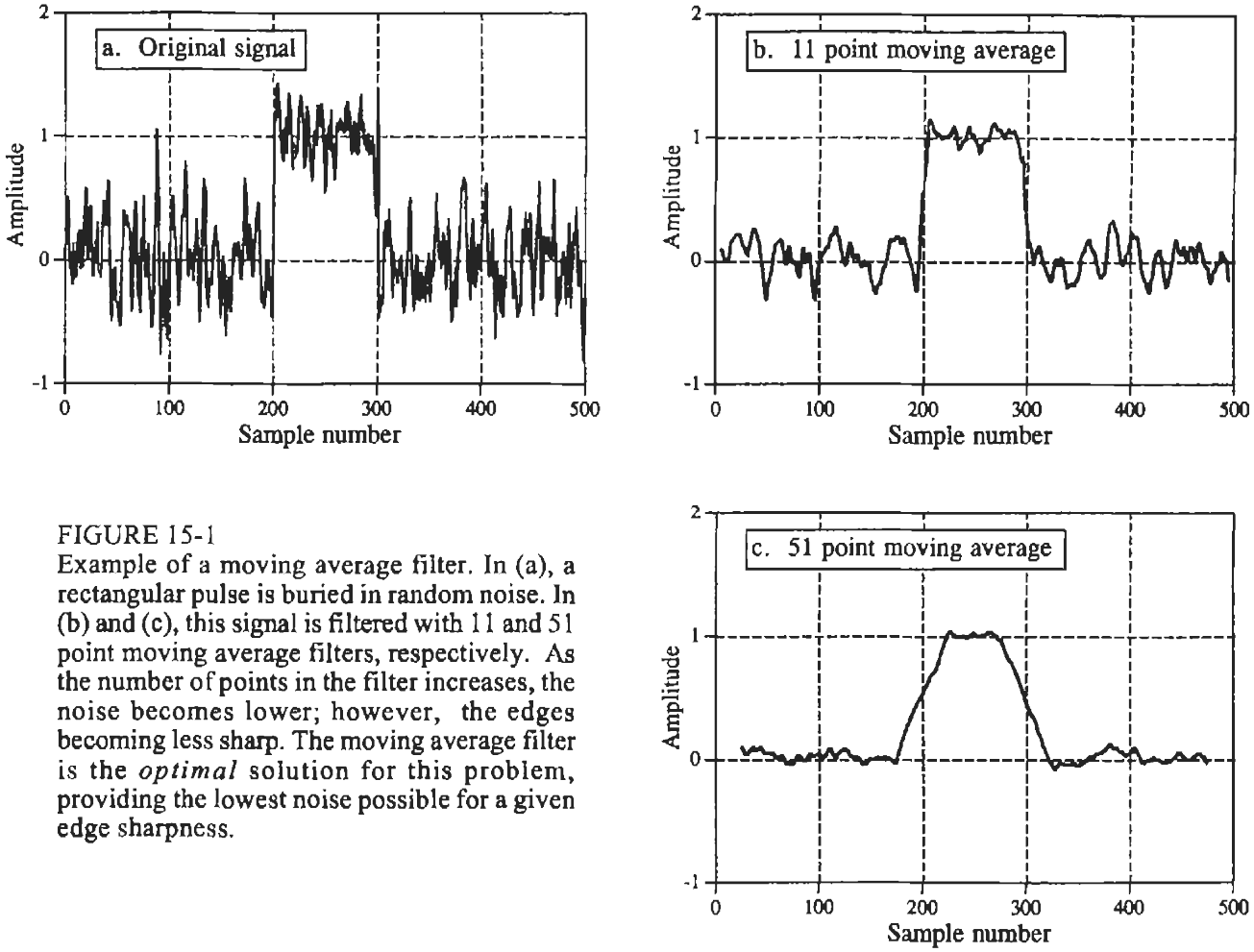

FIGURE 15-1
Example of a moving average filter. In (a), a rectangular pulse is buried in random noise. In (b) and (c), this signal is filtered with 11 and 51 point moving average filters, respectively. As the number of points in the filter increases, the noise becomes lower; however, the edges becoming less sharp. The moving average filter is the *optimal* solution for this problem, providing the lowest noise possible for a given edge sharpness.

Figure 15-1 shows an example of how this works. The signal in (a) is a pulse buried in random noise. In (b) and (c), the smoothing action of the moving average filter decreases the amplitude of the random noise (good), but also reduces the sharpness of the edges (bad). Of all the possible linear filters that could be used, the moving average produces the lowest noise for a given edge sharpness. The amount of noise reduction is equal to the square-root of the number of points in the average. For example, a 100-point moving average filter reduces the noise by a factor of 10.

To understand why the moving average is the best solution, imagine we want to design a filter with a fixed edge sharpness. For example, let's assume we fix the edge sharpness by specifying that there are eleven points in the rise of the step response. This requires that the filter kernel have eleven points. The optimization question is: how do we choose the eleven values in the filter kernel to minimize the noise on the output signal? Since the noise we are trying to reduce is random, none of the input points is special; each is just as noisy as its neighbor. Therefore, it is useless to give preferential treatment to any one of the input points by assigning it a larger coefficient in the filter kernel. The lowest noise is obtained when all the input samples are treated equally, i.e., the moving average filter. (Later in this chapter we show that other filters are essentially *as* good. The point is, no filter is *better* than the simple moving average.)

Frequency Response

Figure 15-2 shows the frequency response of the moving average filter. It is mathematically described by the Fourier transform of the rectangular pulse, as discussed in Chapter 11:

EQUATION 15-2
Frequency response of an M point moving average filter. The frequency, f, runs between 0 and 0.5. For $f = 0$, use: $H[f] = 1$

$$H[f] = \frac{\sin(\pi f M)}{M \sin(\pi f)}$$

The roll-off is very slow and the stopband attenuation is ghastly. Clearly, the moving average filter cannot separate one band of frequencies from another. Remember, good performance in the time domain results in poor performance in the frequency domain, and vice versa. In short, the moving average is an exceptionally good *smoothing filter* (the action in the time domain), but an exceptionally bad *low-pass filter* (the action in the frequency domain).

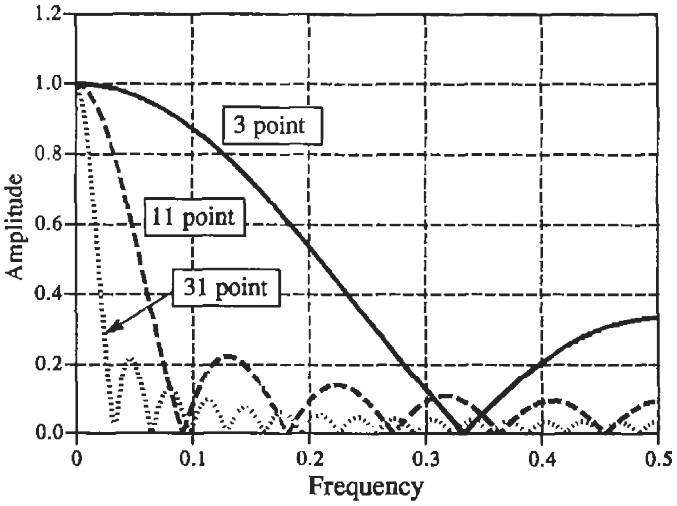

FIGURE 15-2
Frequency response of the moving average filter. The moving average is a very poor low-pass filter, due to its slow roll-off and poor stopband attenuation. These curves are generated by Eq. 15-2.

Relatives of the Moving Average Filter

In a perfect world, filter designers would only have to deal with time domain *or* frequency domain encoded information, but never a mixture of the two in the same signal. Unfortunately, there are some applications where both domains are simultaneously important. For instance, television signals fall into this nasty category. Video information is encoded in the time domain, that is, the shape of the waveform corresponds to the patterns of brightness in the image. However, during transmission the video signal is treated according to its frequency composition, such as its total bandwidth, how the carrier waves for sound and color are added, elimination & restoration of the DC component, etc. As another example, electromagnetic interference is best understood in the frequency domain, even if

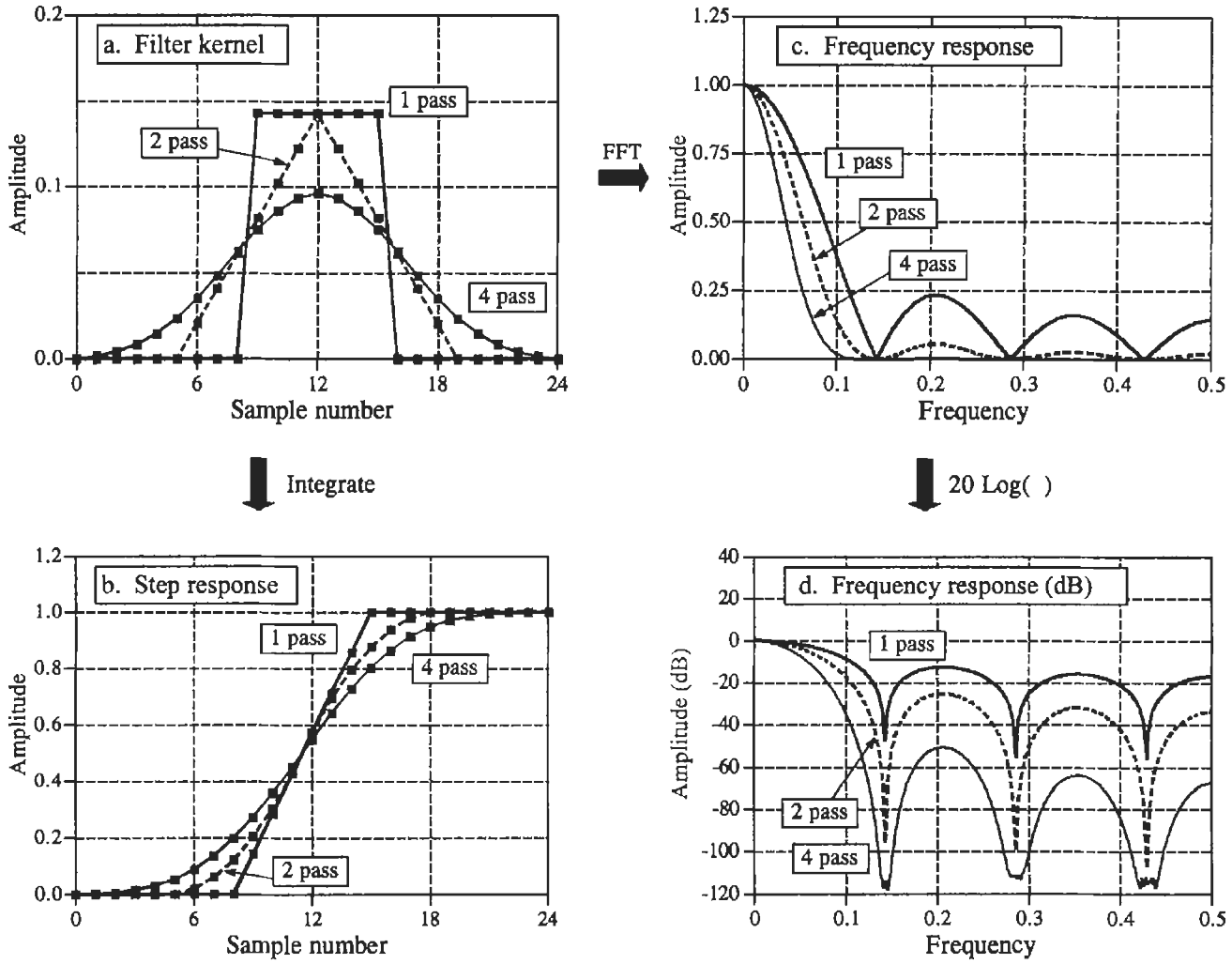

FIGURE 15-3
Characteristics of multiple-pass moving average filters. Figure (a) shows the filter kernels resulting from passing a seven point moving average filter over the data once, twice and four times. Figure (b) shows the corresponding step responses, while (c) and (d) show the corresponding frequency responses.

the signal's information is encoded in the time domain. For instance, the temperature monitor in a scientific experiment might be contaminated with 60 hertz from the power lines, 30 kHz from a switching power supply, or 1320 kHz from a local AM radio station. Relatives of the moving average filter have better frequency domain performance, and can be useful in these mixed domain applications.

Multiple-pass moving average filters involve passing the input signal through a moving average filter two or more times. Figure 15-3a shows the overall filter kernel resulting from one, two and four passes. Two passes are equivalent to using a *triangular* filter kernel (a rectangular filter kernel convolved with itself). After four or more passes, the equivalent filter kernel looks like a *Gaussian* (recall the Central Limit Theorem). As shown in (b), multiple passes produce an "s" shaped step response, as compared to the straight line of the single pass. The frequency responses in (c) and (d) are given by Eq. 15-2 *multiplied* by itself for each pass. That is, each time domain convolution results in a multiplication of the frequency spectra.

Figure 15-4 shows the frequency response of two other relatives of the moving average filter. When a pure **Gaussian** is used as a filter kernel, the frequency response is also a Gaussian, as discussed in Chapter 11. The Gaussian is important because it is the impulse response of many natural and manmade systems. For example, a brief pulse of light entering a long fiber optic transmission line will exit as a Gaussian pulse, due to the different paths taken by the photons within the fiber. The Gaussian filter kernel is also used extensively in *image processing* because it has unique properties that allow fast two-dimensional convolutions (see Chapter 24). The second frequency response in Fig. 15-4 corresponds to using a **Blackman window** as a filter kernel. (The term *window* has no meaning here; it is simply part of the accepted name of this curve). The exact shape of the Blackman window is given in Chapter 16 (Eq. 16-2, Fig. 16-2); however, it looks much like a Gaussian.

How are these relatives of the moving average filter better than the moving average filter itself? Three ways: First, and most important, these filters have better *stopband attenuation* than the moving average filter. Second, the filter kernels *taper* to a smaller amplitude near the ends. Recall that each point in the output signal is a weighted sum of a group of samples from the input. If the filter kernel tapers, samples in the input signal that are farther away are given less weight than those close by. Third, the step responses are *smooth* curves, rather than the abrupt straight line of the moving average. These last two are usually of limited benefit, although you might find applications where they are genuine advantages.

The moving average filter and its relatives are all about the same at reducing random noise while maintaining a sharp step response. The ambiguity lies in how the *risetime* of the step response is measured. If the risetime is measured from 0% to 100% of the step, the moving average filter is the best you can do, as previously shown. In comparison, measuring the risetime from 10% to 90% makes the Blackman window *better* than the moving average filter. The point is, this is just theoretical squabbling; consider these filters equal in this parameter.

The biggest difference in these filters is *execution speed*. Using a recursive algorithm (described next), the moving average filter will run like lightning in your computer. In fact, it is the *fastest* digital filter available. Multiple passes of the moving average will be correspondingly slower, but still very quick. In comparison, the Gaussian and Blackman filters are excruciatingly slow, because they must use convolution. Think a factor of ten times the number of points in the filter kernel (based on multiplication being about 10 times slower than addition). For example, expect a 100 point Gaussian to be 1000 times slower than a moving average using recursion.

Recursive Implementation

A tremendous advantage of the moving average filter is that it can be implemented with an algorithm that is very fast. To understand this

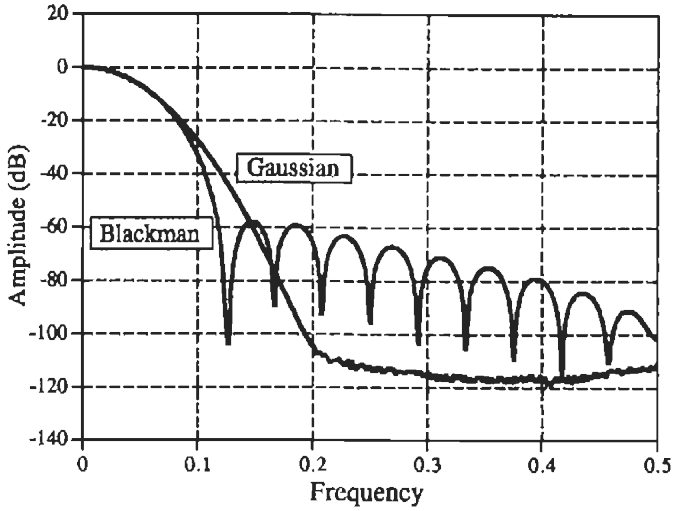

FIGURE 15-4
Frequency response of the Blackman window and Gaussian filter kernels. Both these filters provide better stopband attenuation than the moving average filter. This has no advantage in removing random noise from time domain encoded signals, but it can be useful in mixed domain problems. The disadvantage of these filters is that they must use *convolution*, a terribly slow algorithm.

algorithm, imagine passing an input signal, $x[\ ]$, through a seven-point moving average filter to form an output signal, $y[\ ]$. Now look at how two adjacent output points, $y[50]$ and $y[51]$, are calculated:

$$y[50] = x[47] + x[48] + x[49] + x[50] + x[51] + x[52] + x[53]$$

$$y[51] = x[48] + x[49] + x[50] + x[51] + x[52] + x[53] + x[54]$$

These are nearly the same calculation; points $x[48]$ through $x[53]$ must be added for $y[50]$, and again for $y[51]$. If $y[50]$ has already been calculated, the most *efficient* way to calculate $y[51]$ is:

$$y[51] = y[50] + x[54] - x[47]$$

Once $y[51]$ has been found using $y[50]$, then $y[52]$ can be calculated from sample $y[51]$, and so on. After the first point is calculated in $y[\ ]$, all of the other points can be found with only a single addition and subtraction per point. This can be expressed in the equation:

EQUATION 15-3
Recursive implementation of the moving average filter. In this equation, $x[\ ]$ is the input signal, $y[\ ]$ is the output signal, M is the number of points in the moving average (an odd number). Before this equation can be used, the first point in the signal must be calculated using a standard summation.

$$y[i] = y[i-1] + x[i+p] - x[i-q]$$

where: $p = (M-1)/2$

$q = p + 1$

Notice that this equation use two sources of data to calculate each point in the output: points from the input *and* previously calculated points from the output. This is called a **recursive** equation, meaning that the result of one calculation

is used in *future* calculations. (The term "recursive" also has other meanings, especially in computer science.) Chapter 19 discusses a variety of recursive filters in more detail. Be aware that the moving average recursive filter is very different from typical recursive filters. In particular, most recursive filters have an infinitely long impulse response (IIR), composed of sinusoids and exponentials. The impulse response of the moving average is a rectangular pulse (finite impulse response, or FIR).

This algorithm is faster than other digital filters for several reasons. First, there are only two computations per point, regardless of the length of the filter kernel. Second, addition and subtraction are the only math operations needed, while most digital filters require time-consuming multiplication. Third, the indexing scheme is very simple. Each index in Eq. 15-3 is found by adding or subtracting integer constants that can be calculated before the filtering starts (i.e., p and q). Fourth, the entire algorithm can be carried out with integer representation. Depending on the hardware used, integers can be more than an order of magnitude faster than floating point.

Surprisingly, integer representation works *better* than floating point with this algorithm, in addition to being *faster*. The round-off error from floating point arithmetic can produce unexpected results if you are not careful. For example, imagine a 10,000 sample signal being filtered with this method. The last sample in the filtered signal contains the accumulated error of 10,000 additions and 10,000 subtractions. This appears in the output signal as a drifting offset. Integers don't have this problem because there is no round-off error in the arithmetic. If you must use floating point with this algorithm, the program in Table 15-2 shows how to use a double precision accumulator to eliminate this drift.

```
100 'MOVING AVERAGE FILTER IMPLEMENTED BY RECURSION
110 'This program filters 5000 samples with a 101 point moving
120 'average filter, resulting in 4900 samples of filtered data.
130 'A double precision accumulator is used to prevent round-off drift.
140 '
150 DIM X[4999]                    'X[ ] holds the input signal
160 DIM Y[4999]                    'Y[ ] holds the output signal
170 DEFDBL  ACC                    'Define the variable ACC to be double precision
180 '
190 GOSUB XXXX                     'Mythical subroutine to load X[ ]
200 '
210 ACC = 0                        'Find Y[50] by averaging points X[0] to X[100]
220 FOR I% = 0 TO 100
230   ACC = ACC + X[I%]
240 NEXT I%
250 Y[50] = ACC/101
260 '                              'Recursive moving average filter (Eq. 15-3)
270 FOR I% = 51 TO 4949
280   ACC = ACC + X[I%+50] - X[I%-51]
290   Y[I%] = ACC/101
300 NEXT I%
310 '
320 END
```

TABLE 15-2

CHAPTER

16

Windowed-Sinc Filters

Windowed-sinc filters are used to separate one band of frequencies from another. They are very stable, produce few surprises, and can be pushed to incredible performance levels. These exceptional frequency domain characteristics are obtained at the expense of poor performance in the time domain, including excessive ripple and overshoot in the step response. When carried out by standard convolution, windowed-sinc filters are easy to program, but slow to execute. Chapter 18 shows how the FFT can be used to dramatically improve the computational speed of these filters.

Strategy of the Windowed-Sinc

Figure 16-1 illustrates the idea behind the windowed-sinc filter. In (a), the frequency response of the *ideal* low-pass filter is shown. All frequencies below the cutoff frequency, f_C, are passed with unity amplitude, while all higher frequencies are blocked. The passband is perfectly flat, the attenuation in the stopband is infinite, and the transition between the two is infinitesimally small.

Taking the inverse Fourier transform of this ideal frequency response produces the ideal filter kernel (impulse response) shown in (b). As previously discussed (see Chapter 11, Eq. 11-4), this curve is of the general form: $\sin(x)/x$, called the **sinc function**, given by:

$$h[i] = \frac{\sin(2\pi f_C i)}{i\pi}$$

Convolving an input signal with this filter kernel provides a *perfect* low-pass filter. The problem is, the sinc function continues to both negative and positive infinity without dropping to zero amplitude. While this infinite length is not a problem for *mathematics*, it is a show stopper for *computers*.

To get around this problem, we will make two modifications to the sinc function in (b), resulting in the waveform shown in (c). First, it is truncated to $M+1$ points, symmetrically chosen around the main lobe, where M is an even number. All samples outside these $M+1$ points are set to zero, or simply ignored. Second, the entire sequence is shifted to the right so that it runs from 0 to M. This allows the filter kernel to be represented using only *positive* indexes. While many programming languages allow *negative* indexes, they are a nuisance to use. The sole effect of this $M/2$ shift in the filter kernel is to shift the output signal by the same amount.

Since the modified filter kernel is only an approximation to the ideal filter kernel, it will not have an ideal frequency response. To find the frequency response that is obtained, the Fourier transform can be taken of the signal in (c), resulting in the curve in (d). It's a mess! There is excessive ripple in the passband and poor attenuation in the stopband (recall the Gibbs effect discussed in Chapter 11). These problems result from the abrupt discontinuity at the ends of the truncated sinc function. Increasing the length of the filter kernel does not reduce these problems; the discontinuity is significant no matter how long M is made.

Fortunately, there is a simple method of improving this situation. Figure (e) shows a smoothly tapered curve called a **Blackman window**. Multiplying the truncated-sinc, (c), by the Blackman window, (e), results in the **windowed-sinc** filter kernel shown in (f). The idea is to reduce the abruptness of the truncated ends and thereby improve the frequency response. Figure (g) shows this improvement. The passband is now flat, and the stopband attenuation is so good it cannot be seen in this graph.

Several different windows are available, most of them named after their original developers in the 1950s. Only two are worth using, the **Hamming window** and the **Blackman window**. These are given by:

EQUATION 16-1
The Hamming window. These windows run from $i = 0$ to M, for a total of $M + 1$ points.

$$w[i] = 0.54 - 0.46\cos(2\pi i/M)$$

EQUATION 16-2
The Blackman window.

$$w[i] = 0.42 - 0.5\cos(2\pi i/M) + 0.08\cos(4\pi i/M)$$

Figure 16-2a shows the shape of these two windows for $M = 50$ (i.e., 51 total points in the curves). Which of these two windows should you use? It's a trade-off between parameters. As shown in Fig. 16-2b, the Hamming window has about a 20% faster *roll-off* than the Blackman. However,

FIGURE 16-1 (facing page)
Derivation of the windowed-sinc filter kernel. The frequency response of the ideal low-pass filter is shown in (a), with the corresponding filter kernel in (b), a sinc function. Since the sinc is infinitely long, it must be truncated to be used in a computer, as shown in (c). However, this truncation results in undesirable changes in the frequency response, (d). The solution is to multiply the truncated-sinc with a smooth window, (e), resulting in the windowed-sinc filter kernel, (f). The frequency response of the windowed-sinc, (g), is smooth and well behaved. These figures are not to scale.

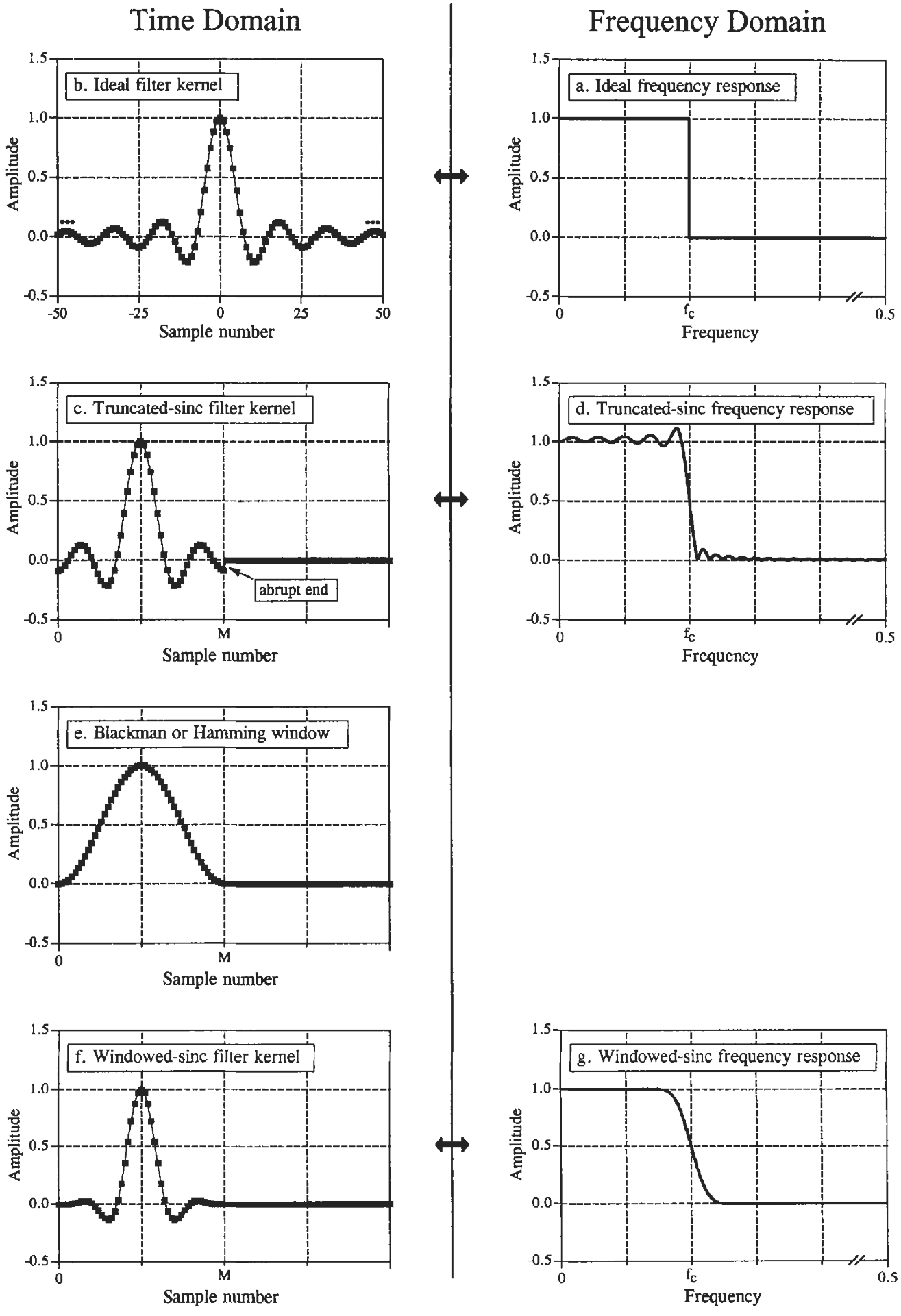
Time Domain
Frequency Domain
b. Ideal filter kernel
a. Ideal frequency response
c. Truncated-sinc filter kernel
d. Truncated-sinc frequency response
abrupt end
e. Blackman or Hamming window
f. Windowed-sinc filter kernel
g. Windowed-sinc frequency response
Amplitude
Sample number
Frequency
1.5
1.0
0.5
0.0
-0.5
-50
-25
0
25
50
M
f_c

FIGURE 16-1

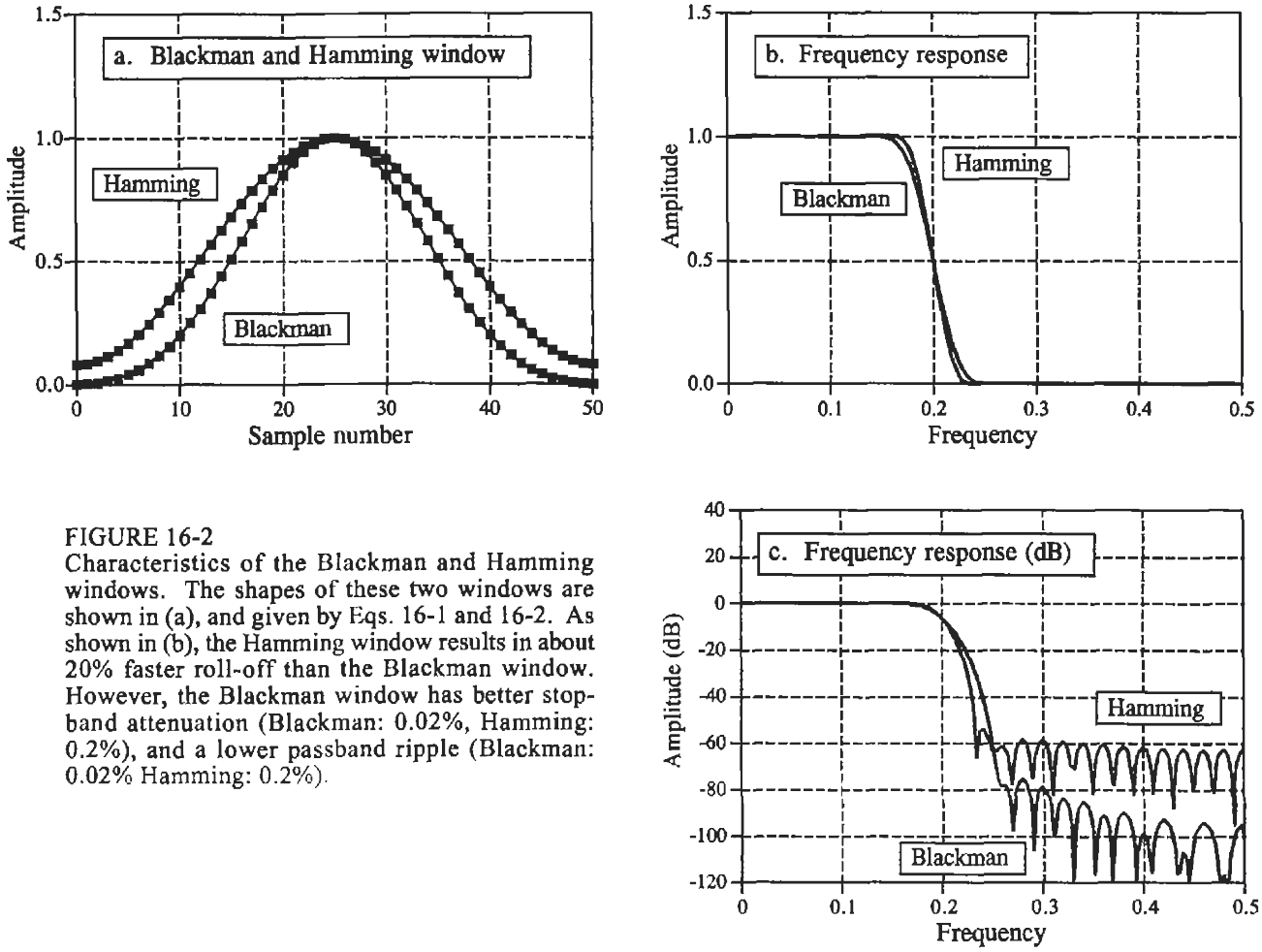

FIGURE 16-2
Characteristics of the Blackman and Hamming windows. The shapes of these two windows are shown in (a), and given by Eqs. 16-1 and 16-2. As shown in (b), the Hamming window results in about 20% faster roll-off than the Blackman window. However, the Blackman window has better stopband attenuation (Blackman: 0.02%, Hamming: 0.2%), and a lower passband ripple (Blackman: 0.02% Hamming: 0.2%).

(c) shows that the Blackman has a better *stopband attenuation*. To be exact, the stopband attenuation for the Blackman is -74dB (~0.02%), while the Hamming is only -53dB (~0.2%). Although it cannot be seen in these graphs, the Blackman has a *passband ripple* of only about 0.02%, while the Hamming is typically 0.2%. In general, the Blackman should be your first choice; a slow roll-off is easier to handle than poor stopband attenuation.

There are other windows you might hear about, although they fall short of the Blackman and Hamming. The **Bartlett window** is a triangle, using straight lines for the taper. The **Hanning window,** also called the **raised cosine window**, is given by: $w[i] = 0.5 - 0.5\cos(2\pi i/M)$. These two windows have about the same roll-off speed as the Hamming, but worse stopband attenuation (Bartlett: -25dB or 5.6%, Hanning -44dB or 0.63%). You might also hear of a **rectangular window**. This is the same as *no* window, just a truncation of the tails (such as in Fig. 16-1c). While the roll-off is ~2.5 times faster than the Blackman, the stopband attenuation is only -21dB (8.9%).

Designing the Filter

To design a windowed-sinc, two parameters must be selected: the cutoff frequency, f_C, and the length of the filter kernel, M. The cutoff frequency

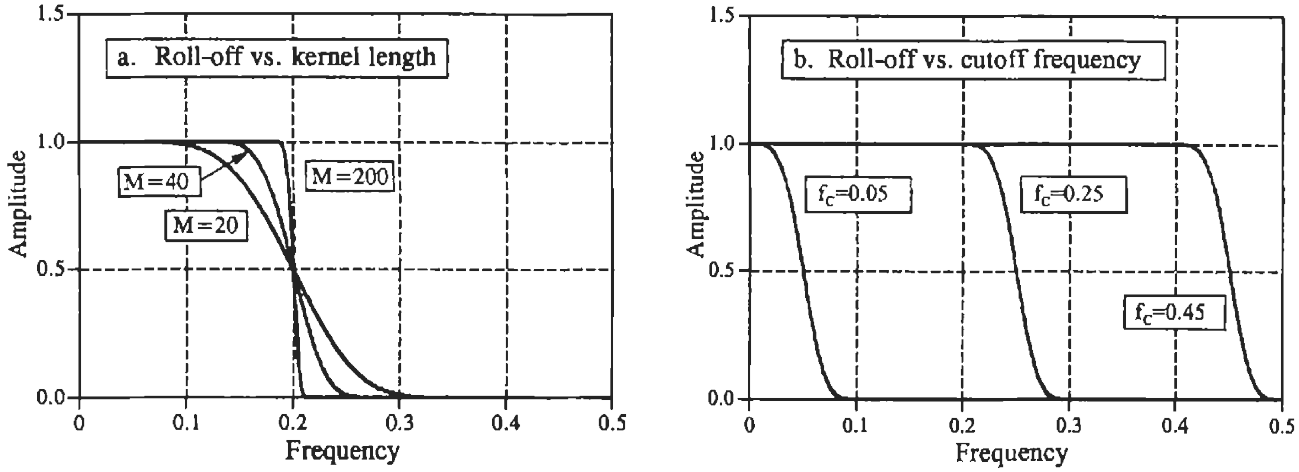

FIGURE 16-3
Filter length vs. roll-off of the windowed-sinc filter. As shown in (a), for $M = 20$, 40, and 200, the transition bandwidths are $BW = 0.2$, 0.1, and 0.02 of the sampling rate, respectively. As shown in (b), the shape of the frequency response does not change with different cutoff frequencies. In (b), $M = 60$.

is expressed as a fraction of the sampling rate, and therefore must be between 0 and 0.5. The value for M sets the *roll-off* according to the approximation:

EQUATION 16-3
Filter length vs. roll-off. The length of the filter kernel, M, determines the transition bandwidth of the filter, BW. This is only an approximation since roll-off depends on the particular window being used.

$$M \approx \frac{4}{BW}$$

where BW is the width of the transition band, measured from where the curve just barely leaves one, to where it almost reaches zero (say, 99% to 1% of the curve). The transition bandwidth is also expressed as a fraction of the sampling frequency, and must between 0 and 0.5. Figure 16-3a shows an example of how this approximation is used. The three curves shown are generated from filter kernels with: M = 20, 40, and 200. From Eq. 16-3, the transition bandwidths are: BW = 0.2, 0.1, and 0.02, respectively. Figure (b) shows that the shape of the frequency response does not depend on the cutoff frequency selected.

Since the time required for a convolution is proportional to the length of the signals, Eq. 16-3 expresses a trade-off between *computation time* (depends on the value of M) and *filter sharpness* (the value of BW). For instance, the 20% slower roll-off of the Blackman window (as compared with the Hamming) can be compensated for by using a filter kernel 20% longer. In other words, it could be said that the Blackman window is 20% slower to execute than an equivalent roll-off Hamming window. This is important because the execution speed of windowed-sinc filters is already terribly slow.

As also shown in Fig. 16-3b, the cutoff frequency of the windowed-sinc filter is measured at the *one-half amplitude* point. Why use 0.5 instead of the

standard 0.707 (-3dB) used in analog electronics and other digital filters? This is because the windowed-sinc's frequency response is *symmetrical* between the passband and the stopband. For instance, the Hamming window results in a passband ripple of 0.2%, and an *identical* stopband attenuation (i.e., ripple in the stopband) of 0.2%. Other filters do not show this symmetry, and therefore have no advantage in using the one-half amplitude point to mark the cutoff frequency. As shown later in this chapter, this symmetry makes the windowed-sinc ideal for *spectral inversion*.

After f_C and M have been selected, the filter kernel is calculated from the relation:

$$h[i] = K \frac{\sin(2\pi f_c(i - M/2))}{i - M/2} \left[0.42 - 0.5\cos\left(\frac{2\pi i}{M}\right) + 0.08\cos\left(\frac{4\pi i}{M}\right) \right]$$

EQUATION 16-4
The windowed-sinc filter kernel. The cutoff frequency, f_C, is expressed as a fraction of the sampling rate, a value between 0 and 0.5. The length of the filter kernel is determined by M, which must be an even integer. The sample number i, is an integer that runs from 0 to M, resulting in $M+1$ total points in the filter kernel. The constant, K, is chosen to provide unity gain at zero frequency. To avoid a divide-by-zero error, for $i = M/2$, use $h[i] = 2\pi f_C K$.

Don't be intimidated by this equation! Based on the previous discussion, you should be able to identify three components: the *sinc function*, the *M/2 shift*, and the *Blackman window*. For the filter to have unity gain at DC, the constant K must be chosen such that the sum of all the samples is equal to one. In practice, ignore K during the calculation of the filter kernel, and then *normalize* all of the samples as needed. The program listed in Table 16-1 shows how this is done. Also notice how the calculation is handled at the center of the sinc, $i = M/2$, which involves a division by zero.

This equation may be long, but it is easy to use; simply type it into your computer program and forget it. Let the computer handle the calculations. If you find yourself trying to evaluate this equation by hand, you are doing something very very wrong.

Let's be specific about where the filter kernel described by Eq. 16-4 is located in your computer array. As an example, M will be chosen to be 100. Remember, M must be an even number. The first point in the filter kernel is in array location 0, while the last point is in array location 100. This means that the entire signal is 101 points long. The center of symmetry is at point 50, i.e., $M/2$. The 50 points to the left of point 50 are symmetrical with the 50 points to the right. Point 0 is the same value as point 100, and point 49 is the same as point 51. If you must have a specific number of samples in the filter kernel, such as to use the FFT, simply add zeros to one end or the other. For example, with $M = 100$, you could make samples 101 through 127 equal to zero, resulting in a filter kernel 128 points long.

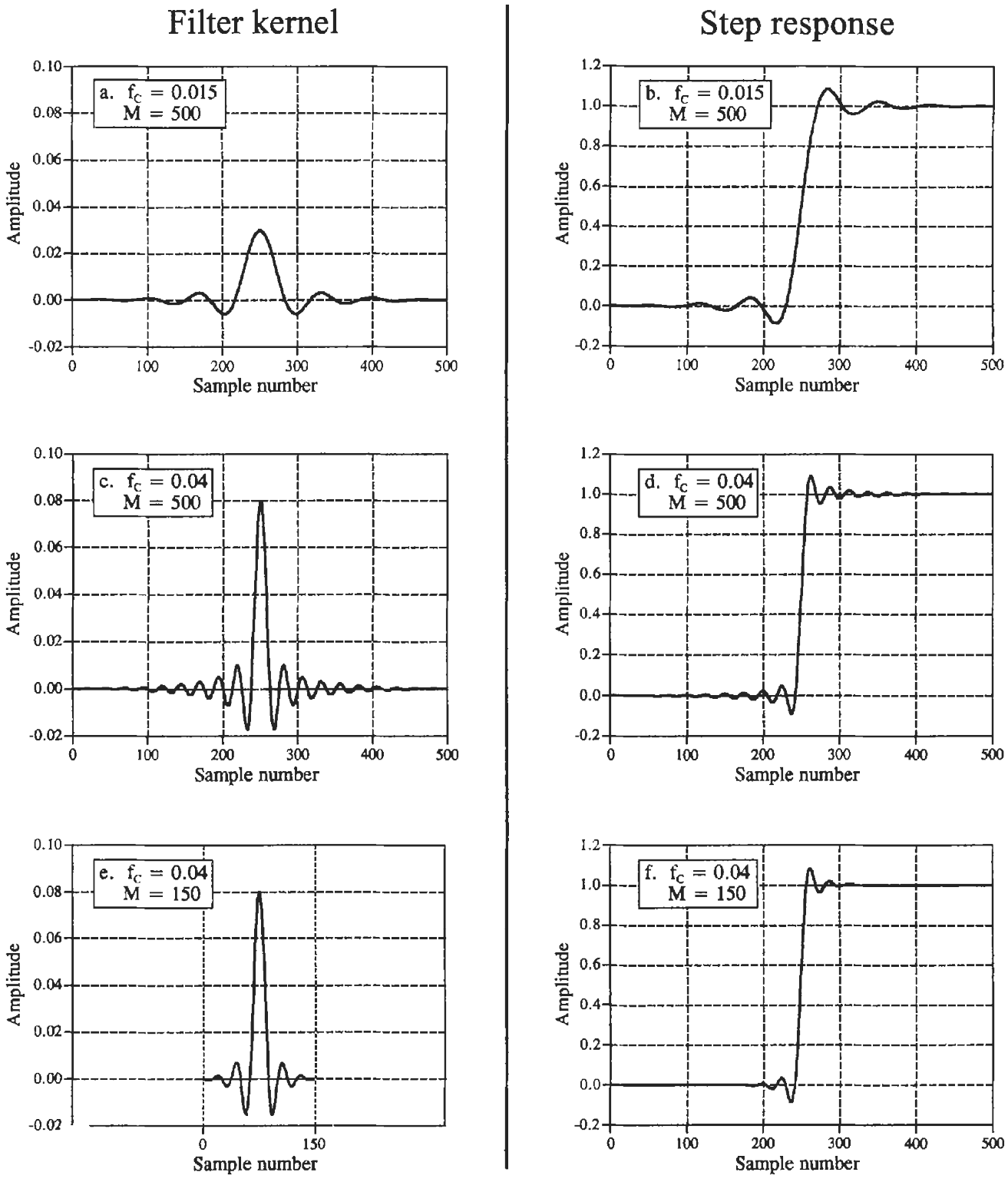

FIGURE 16-4
Example filter kernels and the corresponding step responses. The frequency of the sinusoidal oscillation is approximately equal to the cutoff frequency, f_C, while M determines the kernel length.

Figure 16-4 shows examples of windowed-sinc filter kernels, and their corresponding step responses. The samples at the beginning and end of the filter kernels are so small that they can't even be seen in the graphs. Don't make the mistake of thinking they are unimportant! These samples may be small in value; however, they collectively have a large effect on the

performance of the filter. This is also why floating point representation is typically used to implement windowed-sinc filters. Integers usually don't have enough dynamic range to capture the large variation of values contained in the filter kernel. How does the windowed-sinc filter perform in the time domain? Terrible! The step response has overshoot and ringing; this is *not* a filter for signals with information encoded in the time domain.

Examples of Windowed-Sinc Filters

An electroencephalogram, or EEG, is a measurement of the electrical activity of the brain. It can be detected as millivolt level signals appearing on electrodes attached to the surface of the head. Each nerve cell in the brain generates small electrical pulses. The EEG is the combined result of an enormous number of these electrical pulses being generated in a (hopefully) coordinated manner. Although the relationship between thought and this electrical coordination is very poorly understood, different frequencies in the EEG can be identified with specific mental states. If you close your eyes and relax, the predominant EEG pattern will be a slow oscillation between about 7 and 12 hertz. This waveform is called the *alpha rhythm*, and is associated with contentment and a decreased level of attention. Opening your eyes and looking around causes the EEG to change to the *beta rhythm*, occurring between about 17 and 20 hertz. Other frequencies and waveforms are seen in children, different depths of sleep, and various brain disorders such as epilepsy.

In this example, we will assume that the EEG signal has been amplified by analog electronics, and then digitized at a sampling rate of 100 samples per second. Acquiring data for 50 seconds produces a signal of 5,000 points. Our goal is to separate the alpha from the beta rhythms. To do this, we will design a digital low-pass filter with a cutoff frequency of 14 hertz, or 0.14

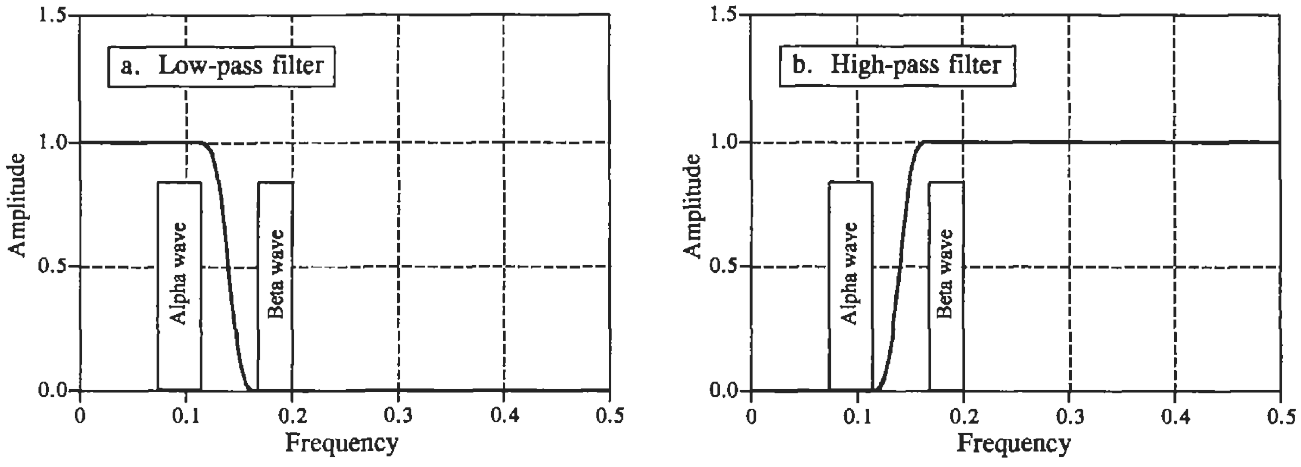

FIGURE 16-5
Example of windowed-sinc filters. The alpha and beta rhythms in an EEG are separated by low-pass and high-pass filters with $M = 100$. The program to implement the low-pass filter is shown in Table 16-1. The program for the high-pass filter is identical, except for a *spectral inversion* of the low-pass filter kernel.

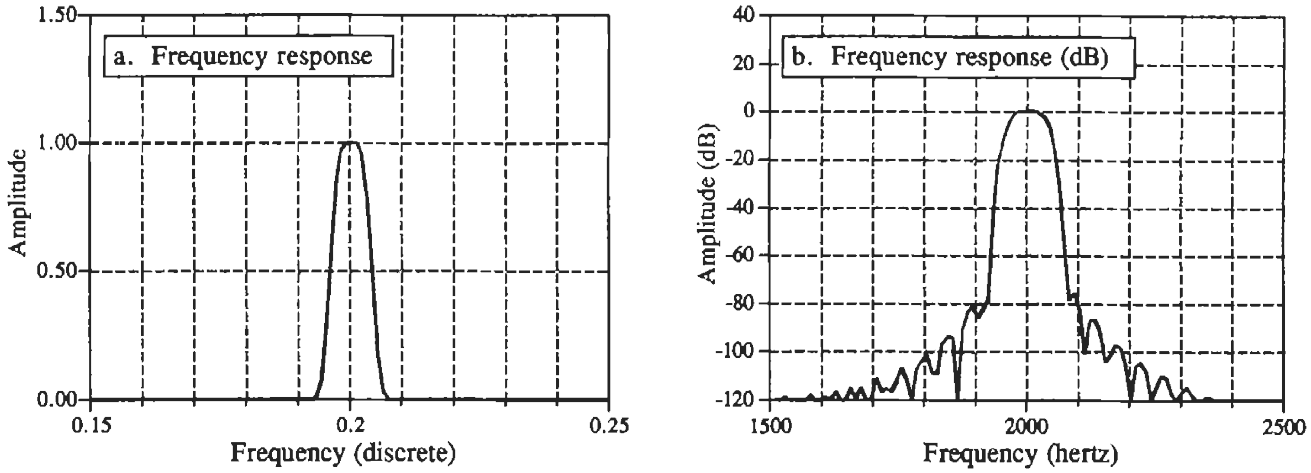

FIGURE 16-6
Example of a windowed-sinc band-pass filter. This filter was designed for a sampling rate of 10 kHz. When referenced to the analog signal, the center frequency of the passband is at 2 kHz, the passband is 80 hertz, and the transition bands are 50 hertz. The windowed-sinc uses 801 points in the filter kernel to achieve this roll-off, and a Blackman window for good stopband attenuation. Figure (a) shows the resulting frequency response on a linear scale, while (b) shows it in decibels. The frequency axis in (a) is expressed as a fraction of the sampling frequency, while (b) is expressed in terms of the analog signal before digitization.

of the sampling rate. The transition bandwidth will be set at 4 hertz, or 0.04 of the sampling rate. From Eq. 16-3, the filter kernel needs to be about 101 points long, and we will arbitrarily choose to use a Hamming window. The program in Table 16-1 shows how the filter is carried out. The frequency response of the filter, obtained by taking the Fourier Transform of the filter kernel, is shown in Fig. 16-5.

In a second example, we will design a *band-pass filter* to isolate a *signaling tone* in an audio signal, such as when a button on a telephone is pressed. We will assume that the signal has been digitized at 10 kHz, and the goal is to isolate an 80 hertz band of frequencies centered on 2 kHz. In terms of the sampling rate, we want to block all frequencies below 0.196 and above 0.204 (corresponding to 1960 hertz and 2040 hertz, respectively). To achieve a transition bandwidth of 50 hertz (0.005 of the sampling rate), we will make the filter kernel 801 points long, and use a Blackman window. Table 16-2 contains a program for calculating the filter kernel, while Fig. 16-6 shows the frequency response. The design involves several steps. First, *two* low-pass filters are designed, one with a cutoff at 0.196, and the other with a cutoff at 0.204. This second filter is then *spectrally inverted*, making it a high-pass filter (see Chapter 14, Fig. 14-6). Next, the two filter kernels are added, resulting in a band-reject filter (see Fig. 14-8). Finally, another *spectral inversion* makes this into the desired band-pass filter.

Pushing it to the Limit

The windowed-sinc filter can be pushed to incredible performance levels without nasty surprises. For instance, suppose you need to isolate a 1 *millivolt* signal riding on a 120 *volt* power line. The low-pass filter will need

```
100 'LOW-PASS WINDOWED-SINC FILTER
110 'This program filters 5000 samples with a 101 point windowed-sinc filter,
120 'resulting in 4900 samples of filtered data.
130 '
140 DIM X[4999]                    'X[ ] holds the input signal
150 DIM Y[4999]                    'Y[ ] holds the output signal
160 DIM H[100]                     'H[ ] holds the filter kernel
170 '
180 PI = 3.14159265
190 FC = .14                       'Set the cutoff frequency (between 0 and 0.5)
200 M% = 100                       'Set filter length (101 points)
210 '
220 GOSUB XXXX                     'Mythical subroutine to load X[ ]
230 '
240 '                              'Calculate the low-pass filter kernel via Eq. 16-4
250 FOR I% = 0 TO 100
260   IF (I%-M%/2) = 0 THEN H[I%] = 2*PI*FC
270   IF (I%-M%/2) <> 0  THEN H[I%] = SIN(2*PI*FC * (I%-M%/2)) / (I%-M%/2)
280   H[I%] = H[I%] * (0.54 - 0.46*COS(2*PI*I%/M%) )
290 NEXT I%
300 '
310 SUM = 0                        'Normalize the low-pass filter kernel for
320 FOR I% = 0 TO 100              'unity gain at DC
330   SUM = SUM + H[I%]
340 NEXT I%
350 '
360 FOR I% = 0 TO 100
370   H[I%] = H[I%] / SUM
380 NEXT I%
390 '
400 FOR J% = 100 TO 4999           'Convolve the input signal & filter kernel
410   Y[J%] = 0
420   FOR I% = 0 TO 100
430     Y[J%] = Y[J%] + X[J%-I%] * H[I%]
440   NEXT I%
450 NEXT J%
460 '
470 END
```

TABLE 16-1

a stopband attenuation of at least -120dB (one part in one-million for those that refuse to learn decibels). As previously shown, the Blackman window only provides -74dB (one part in five-thousand). Fortunately, greater stopband attenuation is easy to obtain. The input signal can be filtered using a conventional windowed-sinc filter kernel, providing an intermediate signal. The intermediate signal can then be passed through the filter a second time, further increasing the stopband attenuation to -148dB (1 part in 30 million, wow!). It is also possible to combine the two stages into a single filter. The kernel of the combined filter is equal to the *convolution* of the filter kernels of the two stages. This also means that convolving any filter kernel *with itself* results in a filter kernel with a much improved stopband attenuation. The price you pay is a longer filter kernel and a slower roll-off. Figure 16-7a shows the frequency response of a 201 point low-pass filter, formed by convolving a 101 point Blackman windowed-sinc with itself. Amazing performance! (If you really need more than -100dB of stopband attenuation, you should use double precision. Single precision

```
100 'BAND-PASS WINDOWED-SINC FILTER
110 'This program calculates an 801 point band-pass filter kernel
120 '
130 DIM A[800]                      'A[ ] workspace for the lower cutoff
140 DIM B[800]                      'B[ ] workspace for the upper cutoff
150 DIM H[800]                      'H[ ] holds the final filter kernel
160 '
170 PI = 3.1415926
180 M% = 800                        'Set filter kernel length (801 points)
190 '
200 '                               'Calculate the first low-pass filter kernel via Eq. 16-4,
210 FC = 0.196                      'with a cutoff frequency of 0.196, store in A[ ]
220 FOR I% = 0 TO 800
230   IF (I%-M%/2) = 0 THEN A[I%] = 2*PI*FC
240   IF (I%-M%/2) <> 0  THEN A[I%] = SIN(2*PI*FC * (I%-M%/2)) / (I%-M%/2)
250   A[I%] = A[I%] * (0.42 - 0.5*COS(2*PI*I%/M%) + 0.08*COS(4*PI*I%/M%))
260 NEXT I%
270 '
280 SUM = 0                         'Normalize the first low-pass filter kernel for
290 FOR I% = 0 TO 800               'unity gain at DC
300   SUM = SUM + A[I%]
310 NEXT I%
320 '
330 FOR I% = 0 TO 800
340   A[I%] = A[I%] / SUM
350 NEXT I%
360 '                               'Calculate the second low-pass filter kernel via Eq. 16-4,
370 FC = 0.204                      'with a cutoff frequency of 0.204, store in B[ ]
380 FOR I% = 0 TO 800
390   IF (I%-M%/2) = 0 THEN B[I%] = 2*PI*FC
400   IF (I%-M%/2) <> 0  THEN B[I%] = SIN(2*PI*FC * (I%-M%/2)) / (I%-M%/2)
410   B[I%] = B[I%] * (0.42 - 0.5*COS(2*PI*I%/M%) + 0.08*COS(4*PI*I%/M%))
420 NEXT I%
430 '
440 SUM = 0                         'Normalize the second low-pass filter kernel for
450 FOR I% = 0 TO 800               'unity gain at DC
460   SUM = SUM + B[I%]
470 NEXT I%
480 '
490 FOR I% = 0 TO 800
500   B[I%] = B[I%] / SUM
510 NEXT I%
520 '
530 FOR I% = 0 TO 800               'Change the low-pass filter kernel in B[ ] into a high-pass
540  B[I%] = - B[I%]                'filter kernel using spectral inversion (as in Fig. 14-5)
550 NEXT I%
560 B[400] =  B[400] + 1
570 '
580 '
590 FOR I% = 0 TO 800               'Add the low-pass filter kernel in A[ ], to the high-pass
600   H[I%] = A[I%] + B[I%]         'filter kernel in B[ ], to form a band-reject filter kernel
610 NEXT I%                         'stored in H[ ] (as in Fig. 14-8)
620 '
630 FOR I% = 0 TO 800               'Change the band-reject filter kernel into a band-pass
640   H[I%] = -H[I%]                'filter kernel by using spectral inversion
650 NEXT I%
660 H[400] =  H[400] + 1
670 '                               'The band-pass filter kernel now resides in H[ ]
680 END
```

TABLE 16-2

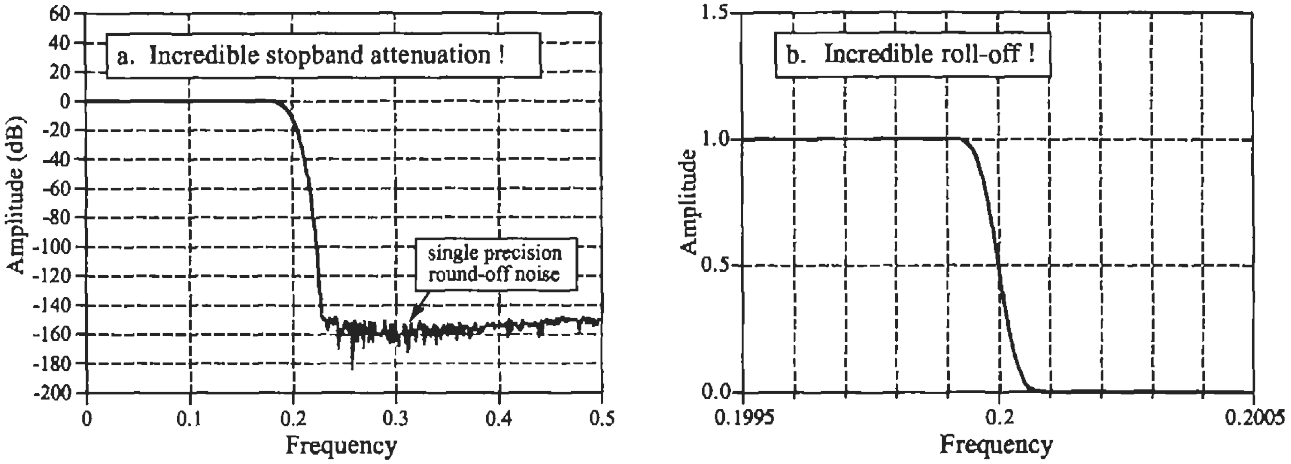

FIGURE 16-7
The incredible performance of the windowed-sinc filter. Figure (a) shows the frequency response of a windowed-sinc filter with increased stopband attenuation. This is achieved by convolving a windowed-sinc filter kernel with itself. Figure (b) shows the very rapid roll-off a 32,001 point windowed-sinc filter.

round-off noise on signals in the *passband* can erratically appear in the *stopband* with amplitudes in the -100dB to -120dB range).

Figure 16-7b shows another example of the windowed-sinc's incredible performance: a low-pass filter with 32,001 points in the kernel. The frequency response appears as expected, with a roll-off of 0.000125 of the sampling rate. How good is this filter? Try building an analog electronic filter that passes signals from DC to 1000 hertz with less than a 0.02% variation, and blocks all frequencies above 1001 hertz with less than 0.02% residue. Now that's a filter! If you really want to be impressed, remember that both the filters in Fig. 16-7 use *single precision*. Using *double precision* allows these performance levels to be extended by a *million* times.

The strongest limitation of the windowed-sinc filter is the *execution time*; it can be unacceptably long if there are many points in the filter kernel and standard convolution is used. A high-speed algorithm for this filter (FFT convolution) is presented in Chapter 18. Recursive filters (Chapter 19) also provide good frequency separation and are a reasonable alternative to the windowed-sinc filter.

Is the windowed-sinc the optimal filter kernel for separating frequencies? No, filter kernels resulting from more sophisticated techniques can be better. But beware! Before you jump into this very mathematical field, you should consider exactly what you hope to gain. The windowed-sinc will provide *any* level of performance that you could possibly need. What the advanced filter design methods may provide is a slightly shorter filter kernel for a given level of performance. This, in turn, may mean a slightly faster execution speed. Be warned that you may get little return for the effort expended.

CHAPTER

17

Custom Filters

Most filters have one of the four standard frequency responses: low-pass, high-pass, band-pass or band-reject. This chapter presents a general method of designing digital filters with an *arbitrary* frequency response, tailored to the needs of your particular application. DSP excels in this area, solving problems that are far above the capabilities of analog electronics. Two important uses of custom filters are discussed in this chapter: *deconvolution*, a way of restoring signals that have undergone an unwanted convolution, and *optimal filtering*, the problem of separating signals with overlapping frequency spectra. This is DSP at its best.

Arbitrary Frequency Response

The approach used to derive the windowed-sinc filter in the last chapter can also be used to design filters with virtually *any* frequency response. The only difference is how the desired response is moved from the frequency domain into the time domain. In the windowed-sinc filter, the frequency response and the filter kernel are both represented by *equations*, and the conversion between them is made by evaluating the *mathematics* of the Fourier transform. In the method presented here, both signals are represented by *arrays of numbers*, with a *computer program* (the FFT) being used to find one from the other.

Figure 17-1 shows an example of how this works. The frequency response we want the filter to produce is shown in (a). To say the least, it is very irregular and would be virtually impossible to obtain with analog electronics. This ideal frequency response is *defined* by an array of numbers that have been selected, not some mathematical equation. In this example, there are 513 samples spread between 0 and 0.5 of the sampling rate. More points could be used to better represent the desired frequency response, while a smaller number may be needed to reduce the computation time during the filter design. However, these concerns are usually small, and 513 is a good length for most applications.

Besides the desired *magnitude* array shown in (a), there must be a corresponding *phase* array of the same length. In this example, the phase of the desired frequency response is entirely *zero* (this array is not shown in Fig. 17-1). Just as with the magnitude array, the phase array can be loaded with any arbitrary curve you would like the filter to produce. However, remember that the first and last samples (i.e., 0 and 512) of the phase array must have a value of *zero* (or a multiple of 2π, which is the same thing). The frequency response can also be specified in rectangular form by defining the array entries for the *real* and *imaginary parts*, instead of using the magnitude and phase.

The next step is to take the inverse DFT to move the filter into the time domain. The quickest way to do this is to convert the frequency domain to rectangular form, and then use the inverse FFT. This results in a 1024 sample signal running from 0 to 1023, as shown in (b). This is the impulse response that corresponds to the frequency response we want; however, it is not suitable for use as a filter kernel (more about this shortly). Just as in the last chapter, it needs to be *shifted*, *truncated*, and *windowed*. In this example, we will design the filter kernel with $M = 40$, i.e., 41 points running from sample 0 to sample 40. Table 17-1 shows a computer program that converts the signal in (b) into the filter kernel shown in (c). As with the windowed-sinc filter, the points near the ends of the filter kernel are so small that they appear to be zero when plotted. Don't make the mistake of thinking they can be deleted!

```
100 'CUSTOM FILTER DESIGN
110 'This program converts an aliased 1024 point impulse response into an M+1 point
120 'filter kernel (such as Fig. 17-1b being converted into Fig. 17-1c)
130 '
140 DIM REX[1023]                   'REX[ ] holds the signal being converted
150 DIM T[1023]                     'T[ ] is a temporary storage buffer
160 '
170 PI = 3.14159265
180 M% = 40                         'Set filter kernel length (41 total points)
190 '
200 GOSUB XXXX                      'Mythical subroutine to load REX[ ] with impulse response
210 '
220 FOR I% = 0 TO 1023   'Shift (rotate) the signal M/2 points to the right
230   INDEX% = I% + M%/2
240   IF INDEX% > 1023 THEN INDEX% = INDEX%-1024
250   T[INDEX%] = REX[I%]
260 NEXT I%
270 '
280 FOR I% = 0 TO 1023
290   REX[I%] = T[I%]
300 NEXT I%
310 '                               'Truncate and window the signal
320 FOR I% = 0 TO 1023
330   IF I% <= M% THEN REX[I%] = REX[I%] * (0.54 - 0.46 * COS(2*PI*I%/M%))
340   IF I% > M%  THEN REX[I%] = 0
350 NEXT I%
360 '                               'The filter kernel now resides in REX[0] to REX[40]
370 END
```

TABLE 17-1

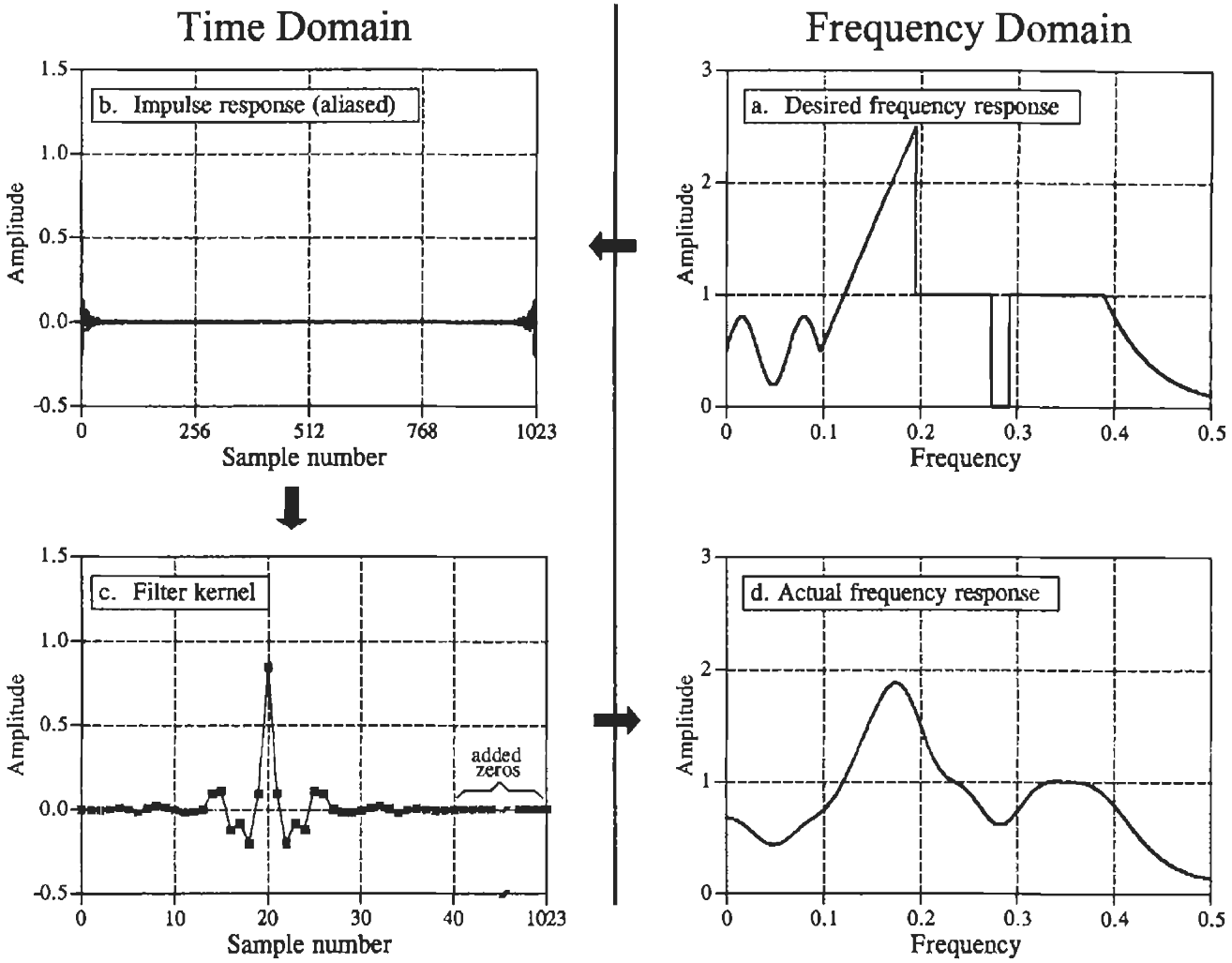

FIGURE 17-1
Example of FIR filter design. Figure (a) shows the desired frequency response, with 513 samples running between 0 to 0.5 of the sampling rate. Taking the inverse DFT results in (b), an *aliased* impulse response composed of 1024 samples. To form the filter kernel, (c), the aliased impulse response is truncated to $M+1$ samples, shifted to the right by $M/2$ samples, and multiplied by a Hamming or Blackman window. In this example, M is 40. The program in Table 17-1 shows how this is done. The filter kernel is tested by padding it with zeros and taking the DFT, providing the actual frequency response of the filter, (d).

The last step is to *test* the filter kernel. This is done by taking the DFT (using the FFT) to find the actual frequency response, as shown in (d). To obtain better resolution in the frequency domain, pad the filter kernel with zeros before the FFT. For instance, using 1024 total samples (41 in the filter kernel, plus 983 zeros), results in 513 samples between 0 and 0.5.

As shown in Fig. 17-2, the length of the filter kernel determines how well the *actual* frequency response matches the *desired* frequency response. The exceptional performance of FIR digital filters is apparent; virtually any frequency response can be obtained if a long enough filter kernel is used.

This is the entire design method; however, there is a subtle *theoretical* issue that needs to be clarified. Why isn't it possible to directly use the impulse response shown in 17-1b as the filter kernel? After all, if (a) is the Fourier transform of (b), wouldn't convolving an input signal with (b) produce the *exact* frequency response we want? The answer is *no*, and here's why.

When designing a custom filter, the desired frequency response is defined by the values in an array. Now consider this: what does the frequency response do *between* the specified points? For simplicity, two cases can be imagined, one "good" and one "bad." In the "good" case, the frequency response is a smooth curve between the defined samples. In the "bad" case, there are wild fluctuations between. As luck would have it, the impulse response in (b) corresponds to the "bad" frequency response. This can be shown by padding it with a large number of zeros, and then taking the DFT. The frequency response obtained by this method will show the erratic behavior between the originally defined samples, and look just awful.

To understand this, imagine that we force the frequency response to be what we want by defining it at an infinite number of points between 0 and 0.5. That is, we create a continuous curve. The inverse DTFT is then used to find the impulse response, which will be *infinite* in length. In other words, the "good" frequency response corresponds to something that cannot be represented in a computer, an infinitely long impulse response. When we represent the frequency spectrum with $N/2 + 1$ samples, only N points are provided in the time domain, making it unable to correctly contain the signal. The result is that the infinitely long impulse response wraps up (aliases) into the N points. When this aliasing occurs, the frequency response changes from "good" to "bad." Fortunately, windowing the N point impulse response greatly reduces this aliasing, providing a smooth curve between the frequency domain samples.

Designing a digital filter to produce a given frequency response is quite simple. The hard part is finding what frequency response to use. Let's look at some strategies used in DSP to design custom filters.

Deconvolution

Unwanted convolution is an inherent problem in transferring analog information. For instance, all of the following can be modeled as a convolution: image blurring in a shaky camera, echoes in long distance telephone calls, the finite bandwidth of analog sensors and electronics, etc. Deconvolution is the process of filtering a signal to compensate for an undesired convolution. The goal of deconvolution is to recreate the signal as it existed *before* the convolution took place. This usually requires the characteristics of the convolution (i.e., the impulse or frequency response) to be known. This can be distinguished from **blind deconvolution**, where the characteristics of the parasitic convolution are *not* known. Blind deconvolution is a much more difficult problem that has no general solution, and the approach must be tailored to the particular application.

Deconvolution is nearly impossible to understand in the *time domain*, but quite straightforward in the *frequency domain*. Each sinusoid that composes the original signal can be changed in amplitude and/or phase as it passes through the undesired convolution. To extract the original signal, the deconvolution filter must *undo* these amplitude and phase changes. For

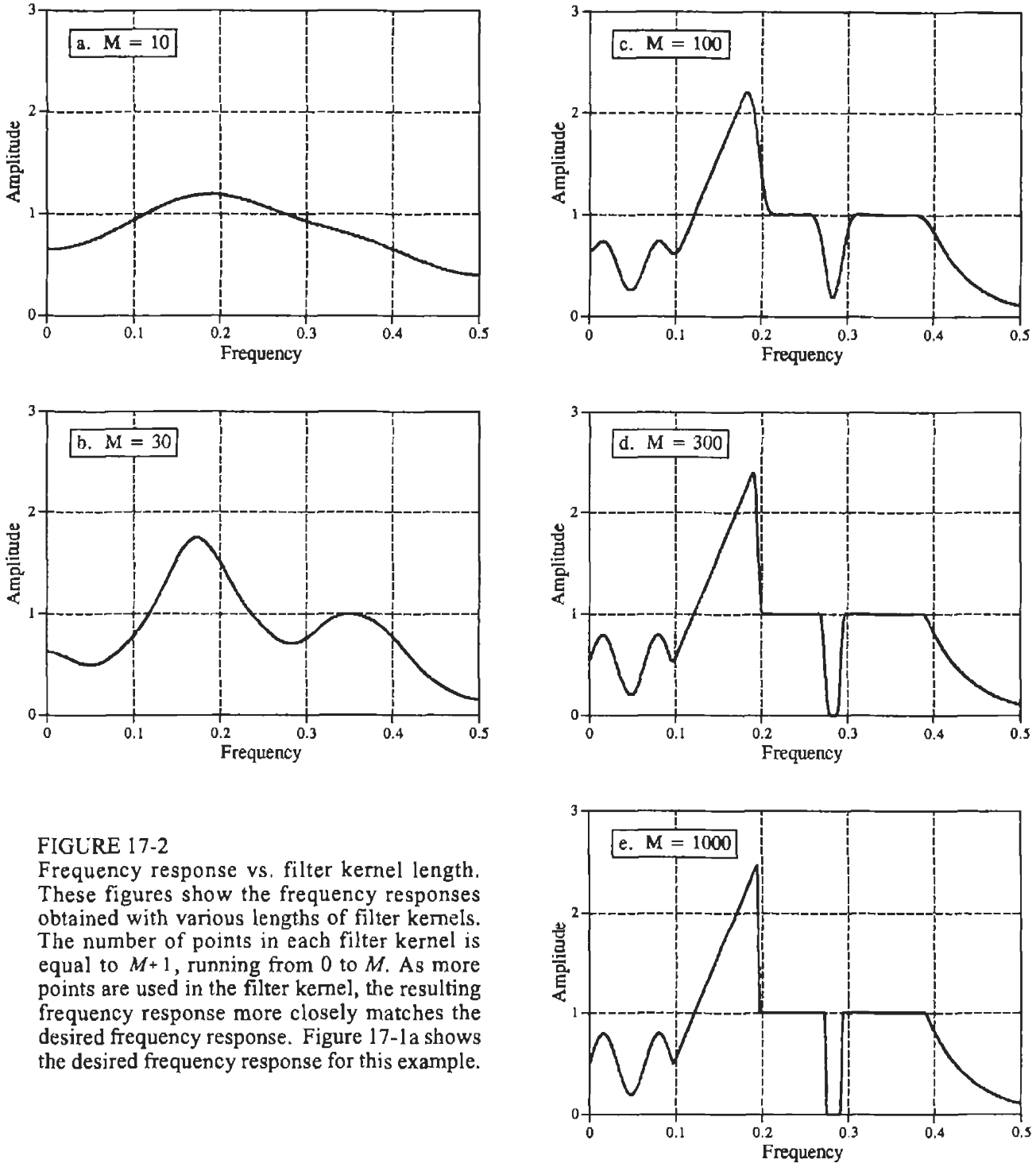

FIGURE 17-2
Frequency response vs. filter kernel length. These figures show the frequency responses obtained with various lengths of filter kernels. The number of points in each filter kernel is equal to $M+1$, running from 0 to M. As more points are used in the filter kernel, the resulting frequency response more closely matches the desired frequency response. Figure 17-1a shows the desired frequency response for this example.

example, if the convolution changes a sinusoid's amplitude by 0.5 with a 30 degree phase shift, the deconvolution filter must amplify the sinusoid by 2.0 with a -30 degree phase change.

The example we will use to illustrate deconvolution is a *gamma ray detector.* As illustrated in Fig. 17-3, this device is composed of two parts, a *scintillator* and a *light detector*. A scintillator is a special type of transparent material, such as sodium iodide or bismuth germanate. These compounds change the energy in each gamma ray into a brief burst of visible light. This light

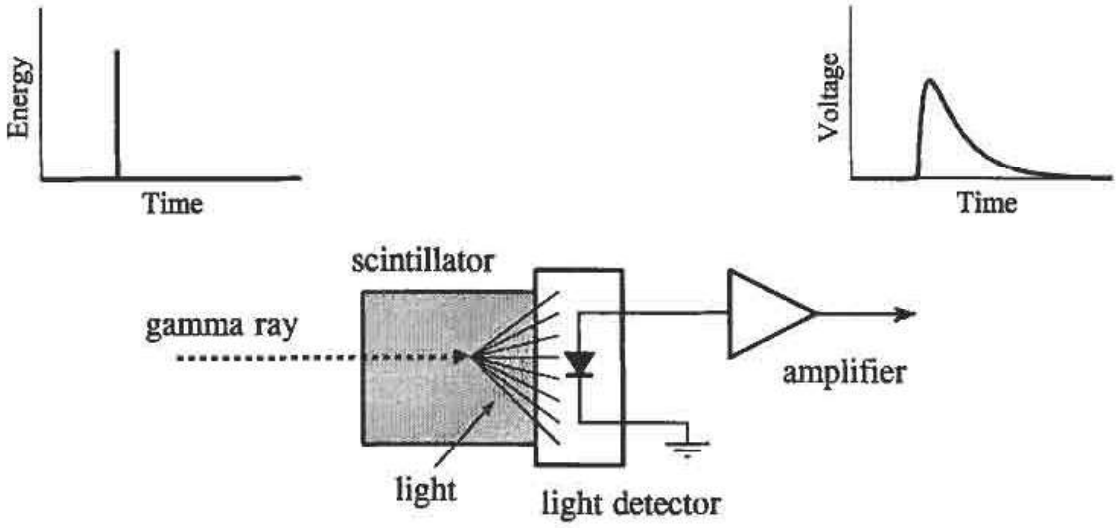

FIGURE 17-3
Example of an unavoidable convolution. A gamma ray detector can be formed by mounting a *scintillator* on a *light detector*. When a gamma ray strikes the scintillator, its energy is converted into a pulse of light. This pulse of light is then converted into an electronic signal by the light detector. The gamma ray is an *impulse*, while the output of the detector (i.e., the *impulse response*) resembles a one-sided exponential.

is then converted into an electronic signal by a light detector, such as a photodiode or photomultiplier tube. Each pulse produced by the detector resembles a *one-sided exponential*, with some rounding of the corners. This shape is determined by the characteristics of the scintillator used. When a gamma ray deposits its energy into the scintillator, nearby atoms are excited to a higher energy level. These atoms randomly *deexcite*, each producing a single photon of visible light. The net result is a light pulse whose amplitude decays over a few hundred nanoseconds (for sodium iodide). Since the arrival of each gamma ray is an *impulse*, the output pulse from the detector (i.e., the one-sided exponential) is the *impulse response* of the system.

Figure 17-4a shows pulses generated by the detector in response to randomly arriving gamma rays. The information we would like to extract from this output signal is the *amplitude* of each pulse, which is proportional to the *energy* of the gamma ray that generated it. This is useful information because the energy can tell interesting things about where the gamma ray has been. For example, it may provide medical information on a patient, tell the age of a distant galaxy, detect a bomb in airline luggage, etc.

Everything would be fine if only an occasional gamma ray were detected, but this is usually not the case. As shown in (a), two or more pulses may overlap, shifting the measured amplitude. One answer to this problem is to *deconvolve* the detector's output signal, making the pulses narrower so that less pile-up occurs. Ideally, we would like each pulse to resemble the original impulse. As you may suspect, this isn't possible and we must settle for a pulse that is finite in length, but significantly shorter than the detected pulse. This goal is illustrated in Fig. 17-4b.

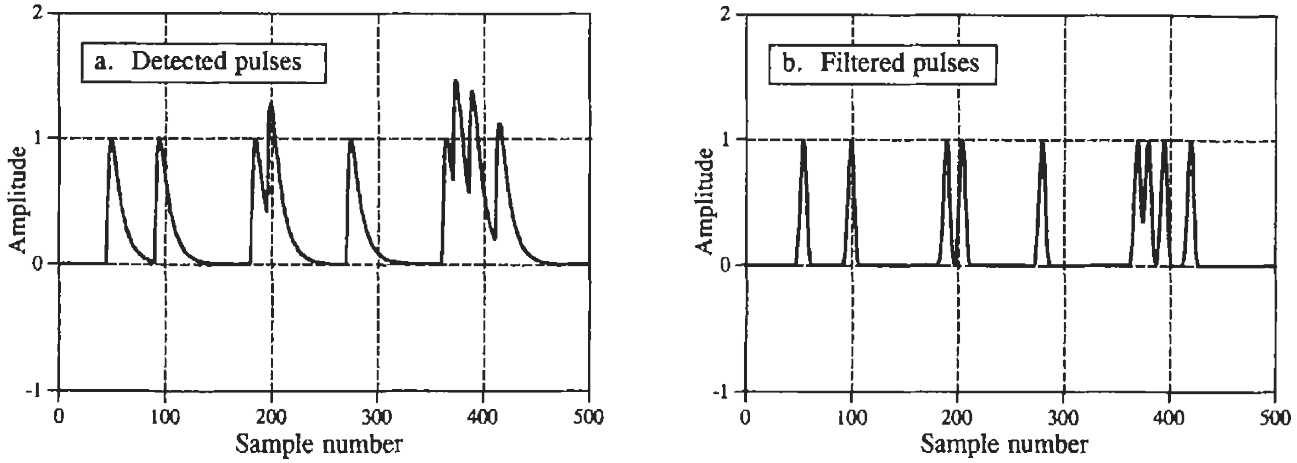

FIGURE 17-4
Example of deconvolution. Figure (a) shows the output signal from a gamma ray detector in response to a series of randomly arriving gamma rays. The deconvolution filter is designed to convert (a) into (b), by reducing the width of the pulses. This minimizes the amplitude shift when pulses land on top of each other.

Even though the detector signal has its information encoded in the *time domain*, much of our analysis must be done in the *frequency domain*, where the problem is easier to understand. Figure 17-5a is the signal produced by the detector (something we know). Figure (c) is the signal we wish to have (also something we know). This desired pulse was arbitrarily selected to be the same shape as a Blackman window, with a length about one-third that of the original pulse. Our goal is to find a filter kernel, (e), that when convolved with the signal in (a), produces the signal in (c). In equation form: if $a * e = c$, and given a and c, find e.

If these signals were combined by addition or multiplication instead of convolution, the solution would be easy: *subtraction* is used to "de-add" and *division* is used to "de-multiply." Convolution is different; there is not a simple inverse operation that can be called "deconvolution." Convolution is too messy to be undone by directly manipulating the time domain signals.

Fortunately, this problem is simpler in the frequency domain. Remember, *convolution* in one domain corresponds with *multiplication* in the other domain. Again referring to the signals in Fig. 17-5: if $b \times f = d$, and given b and d, find f. This is an easy problem to solve: the frequency response of the filter, (f), is the frequency spectrum of the desired pulse, (d), *divided* by the frequency spectrum of the detected pulse, (b). Since the detected pulse is asymmetrical, it will have a *nonzero* phase. This means that a *complex* division must be used (that is, a magnitude & phase divided by another magnitude & phase). In case you have forgotten, Chapter 9 defines how to perform a complex division of one spectrum by another. The required filter kernel, (e), is then found from the frequency response by the custom filter method (IDFT, shift, truncate, & multiply by a window).

There are limits to the improvement that deconvolution can provide. In other words, if you get greedy, things will fall apart. Getting greedy in this

example means trying to make the desired pulse excessively narrow. Let's look at what happens. If the desired pulse is made narrower, its frequency spectrum must contain more high frequency components. Since these high frequency components are at a very low amplitude in the detected pulse, the filter must have a very high gain at these frequencies. For instance, (f) shows that some frequencies must be multiplied by a factor of *three* to achieve the desired pulse in (c). If the desired pulse is made narrower, the gain of the deconvolution filter will be even greater at high frequencies.

The problem is, small errors are very unforgiving in this situation. For instance, if some frequency is amplified by 30, when only 28 is required, the deconvolved signal will probably be a mess. When the deconvolution is pushed to greater levels of performance, the characteristics of the unwanted convolution must be understood with greater *accuracy* and *precision*. There are always unknowns in real world applications, caused by such villains as: electronic noise, temperature drift, variation between devices, etc. These unknowns set a limit on how well deconvolution will work.

Even if the unwanted convolution is *perfectly* understood, there is still a factor that limits the performance of deconvolution: *noise*. For instance, most unwanted convolutions take the form of a low-pass filter, reducing the amplitude of the high frequency components in the signal. Deconvolution corrects this by amplifying these frequencies. However, if the amplitude of these components falls below the inherent noise of the system, the information contained in these frequencies is lost. No amount of signal processing can retrieve it. It's gone forever. Adios! Goodbye! Sayonara! Trying to reclaim this data will only amplify the noise. As an extreme case, the amplitude of some frequencies may be completely reduced to *zero*. This not only obliterates the information, it will try to make the deconvolution filter have *infinite* gain at these frequencies. The solution: design a less aggressive deconvolution filter and/or place limits on how much gain is allowed at any of the frequencies.

How far can you go? How greedy is too greedy? This depends totally on the problem you are attacking. If the signal is well behaved and has low noise, a significant improvement can probably be made (think a factor of 5-10). If the signal changes over time, isn't especially well understood, or is noisy, you won't do nearly as well (think a factor of 1-2). Successful deconvolution involves a great deal of testing. If it works at some level, try going farther; you will know when it falls apart. No amount of theoretical work will allow you to bypass this iterative process.

Deconvolution can also be applied to *frequency domain* encoded signals. A classic example is the restoration of old recordings of the famous opera singer, Enrico Caruso (1873-1921). These recordings were made with very primitive equipment by modern standards. The most significant problem is the *resonances* of the long tubular recording horn used to gather the sound. Whenever the singer happens to hit one of these resonance frequencies, the loudness of the recording abruptly increases. Digital deconvolution has improved the subjective quality of these recordings by

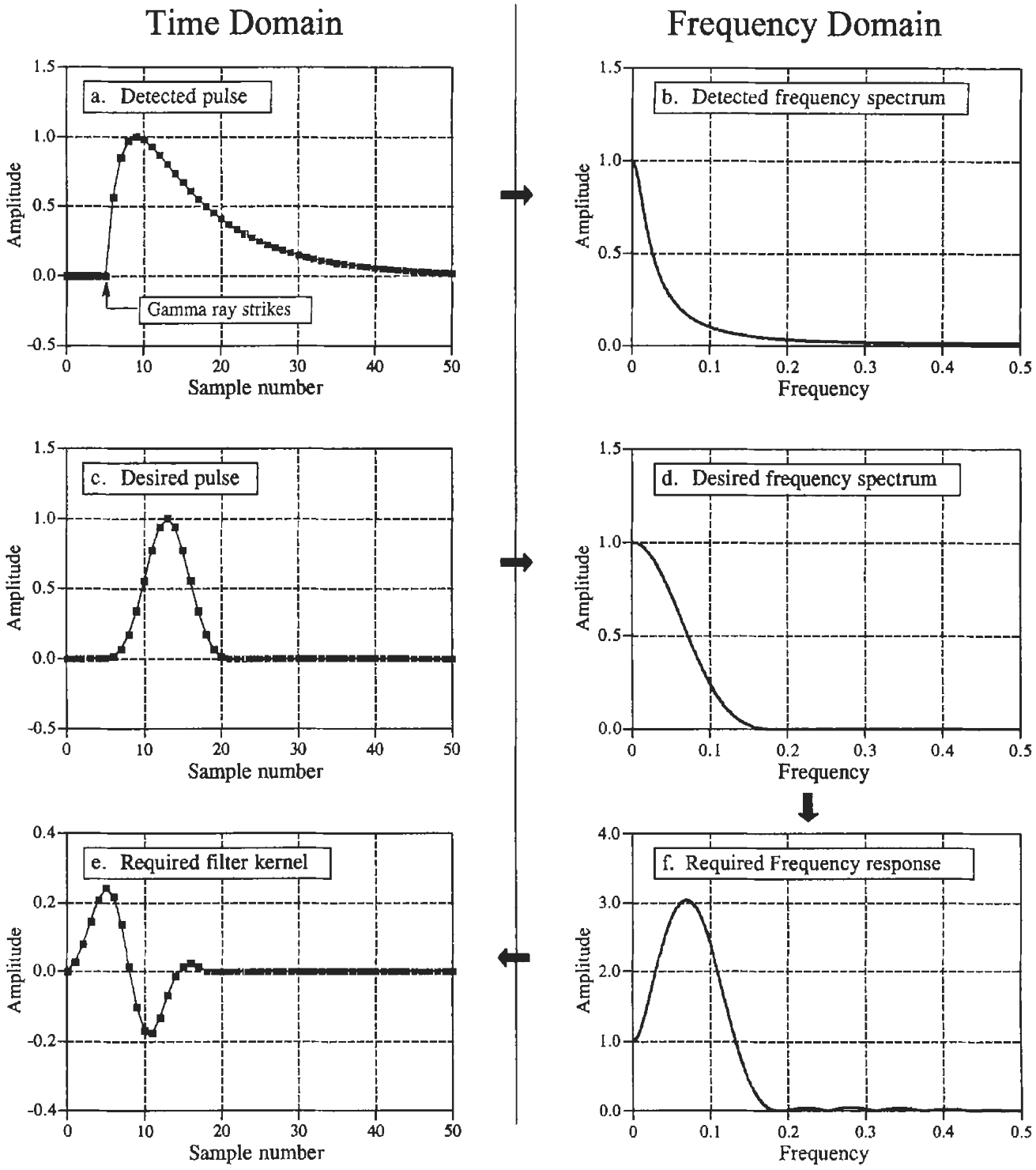

FIGURE 17-5
Example of deconvolution in the time and frequency domains. The impulse response of the example gamma ray detector is shown in (a), while the desired impulse response is shown in (c). The frequency spectra of these two signals are shown in (b) and (d), respectively. The filter that changes (a) into (c) has a frequency response, (f), equal to (d) divided by (b). The filter kernel of this filter, (e), is then found from the frequency response using the custom filter design method (inverse DFT, truncation, windowing). Only the magnitudes of the frequency domain signals are shown in this illustration; however, the phases are nonzero and must also be used.

reducing the loud spots in the music. We will only describe the general method; for a detailed description, see the original paper: T. Stockham, T. Cannon, and R. Ingebretsen, "Blind Deconvolution Through Digital Signal Processing", *Proc. IEEE*, vol. 63, Apr. 1975, pp. 678-692.

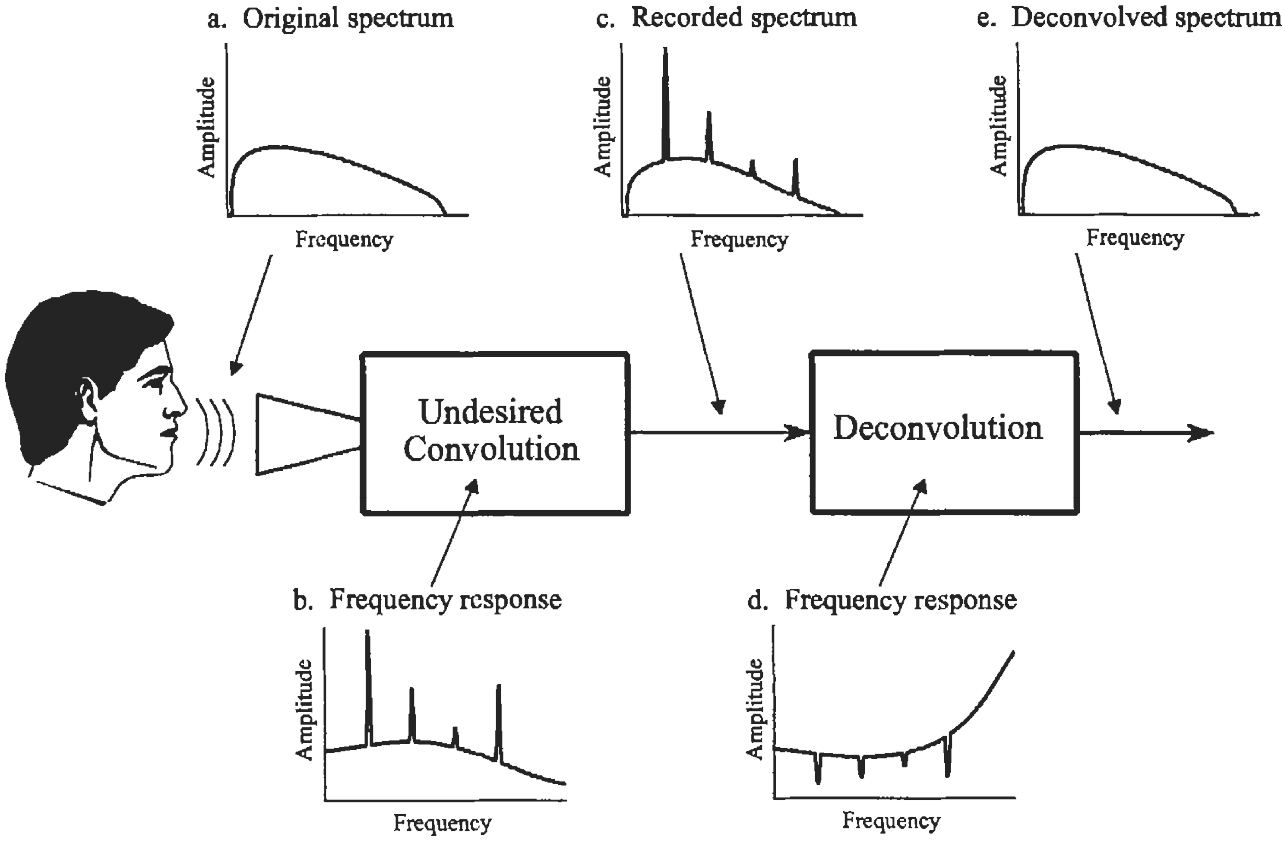

FIGURE 17-6
Deconvolution of old phonograph recordings. The frequency spectrum produced by the original singer is illustrated in (a). Resonance peaks in the primitive equipment, (b), produce distortion in the recorded frequency spectrum, (c). The frequency response of the deconvolution filter, (d), is designed to counteracts the undesired convolution, restoring the original spectrum, (e). These graphs are for illustrative purposes only; they are not actual signals.

Figure 17-6 shows the general approach. The frequency spectrum of the original audio signal is illustrated in (a). Figure (b) shows the frequency response of the recording equipment, a relatively smooth curve except for several sharp resonance peaks. The spectrum of the recorded signal, shown in (c), is equal to the true spectrum, (a), multiplied by the uneven frequency response, (b). The goal of the deconvolution is to *counteract* the undesired convolution. In other words, the frequency response of the deconvolution filter, (d), must be the *inverse* of (b). That is, each peak in (b) is cancelled by a corresponding dip in (d). If this filter were perfectly designed, the resulting signal would have a spectrum, (e), identical to that of the original. Here's the catch: the original recording equipment has long been discarded, and its frequency response, (b), is a mystery. In other words, this is a *blind deconvolution* problem; given only (c), how can we determine (d)?

Blind deconvolution problems are usually attacked by making an estimate or assumption about the unknown parameters. To deal with this example, the *average spectrum* of the original music is assumed to match the *average spectrum* of the same music performed by a present day singer using modern equipment. The *average spectrum* is found by the techniques of Chapter 9:

break the signal into a large number of segments, take the DFT of each segment, convert into polar form, and then average the magnitudes together. In the simplest case, the unknown frequency response is taken as the average spectrum of the old recording, divided by the average spectrum of the modern recording. (The method used by Stockham et al. is based on a more sophisticated technique called *homomorphic* processing, providing a better estimate of the characteristics of the recording system).

Optimal Filters

Figure 17-7a illustrates a common filtering problem: trying to extract a waveform (in this example, an exponential pulse) buried in random noise. As shown in (b), this problem is no easier in the frequency domain. The signal has a spectrum composed mainly of low-frequency components. In comparison, the spectrum of the noise is *white* (the same amplitude at all frequencies). Since the spectra of the signal and noise *overlap*, it is not clear how the two can best be separated. In fact, the real question is how to define what "best" means. We will look at three filters, each of which is "best" (optimal) in a different way. Figure 17-8 shows the filter kernel and frequency response for each of these filters. Figure 17-9 shows the result of using these filters on the example waveform of Fig. 17-7a.

The **moving average filter** is the topic of Chapter 15. As you recall, each output point produced by the moving average filter is the average of a certain number of points from the input signal. This makes the filter kernel a rectangular pulse with an amplitude equal to the reciprocal of the number of points in the average. The moving average filter is optimal in the sense that it provides the fastest step response for a given noise reduction.

The **matched filter** was previously discussed in Chapter 7. As shown in Fig. 17-8a, the filter kernel of the matched filter is the same as the target signal

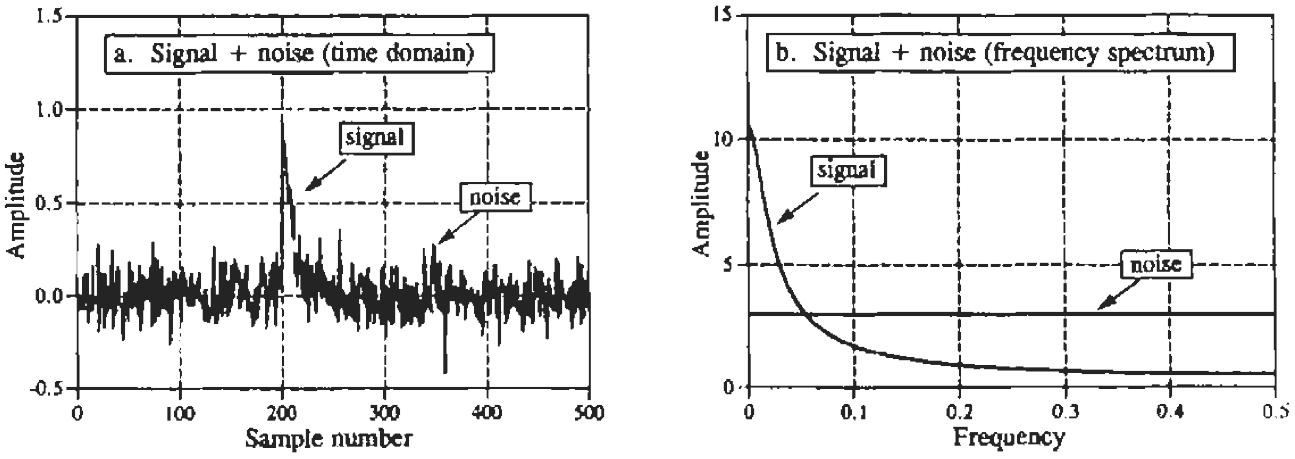

FIGURE 17-7
Example of optimal filtering. In (a), an exponential pulse buried in random noise. The frequency spectra of the pulse and noise are shown in (b). Since the signal and noise overlap in both the time and frequency domains, the best way to separate them isn't obvious.

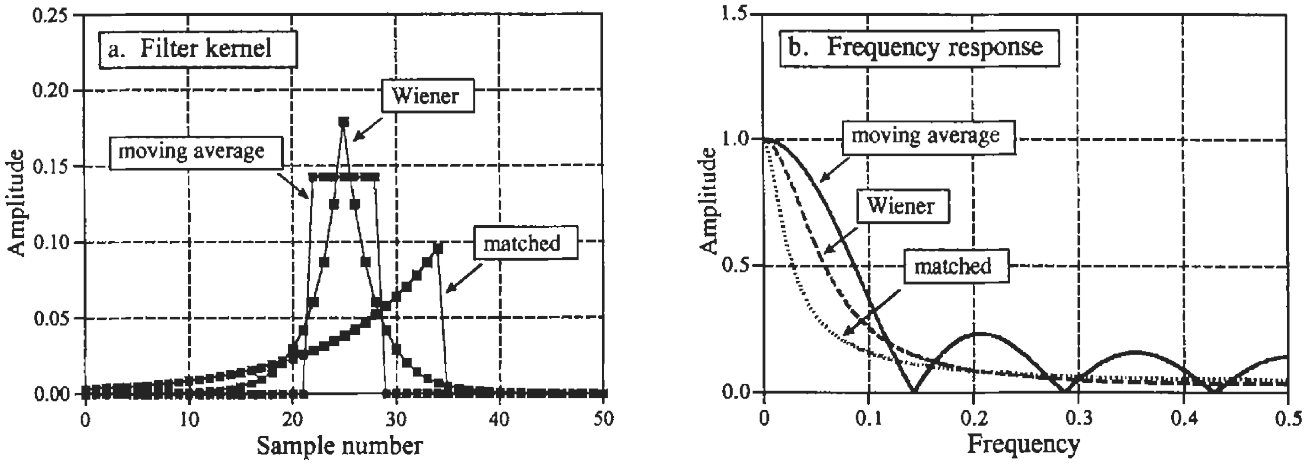

FIGURE 17-8
Example of optimal filters. In (a), three filter kernels are shown, each of which is optimal in some sense. The corresponding frequency responses are shown in (b). The moving average filter is designed to have a rectangular pulse for a filter kernel. In comparison, the filter kernel of the matched filter looks like the signal being detected. The Wiener filter is designed in the frequency domain, based on the relative amounts of signal and noise present at each frequency.

being detected, except it has been flipped left-for-right. The idea behind the matched filter is *correlation*, and this flip is required to perform *correlation* using *convolution*. The amplitude of each point in the output signal is a measure of how well the filter kernel *matches* the corresponding section of the input signal. Recall that the output of a matched filter does not necessarily look like the signal being detected. This doesn't really matter; if a matched filter is used, the shape of the target signal must already be known. The matched filter is optimal in the sense that the top of the peak is farther above the noise than can be achieved with any other linear filter (see Fig. 17-9b).

The **Wiener filter** (named after the optimal estimation theory of Norbert Wiener) separates signals based on their frequency spectra. As shown in Fig. 17-7b, at some frequencies there is mostly signal, while at others there is mostly noise. It seems logical that the "mostly signal" frequencies should be passed through the filter, while the "mostly noise" frequencies should be blocked. The Wiener filter takes this idea a step further; the gain of the filter *at each frequency* is determined by the relative amount of signal and noise *at that frequency*:

EQUATION 17-1
The Wiener filter. The frequency response, represented by $H[f]$, is determined by the frequency spectra of the noise, $N[f]$, and the signal, $S[f]$. Only the magnitudes are important; all of the phases are zero.

$$H[f] = \frac{S[f]^2}{S[f]^2 + N[f]^2}$$

This relation is used to convert the spectra in Fig. 17-7b into the Wiener filter's frequency response in Fig. 17-8b. The Wiener filter is optimal in the sense that it maximizes the ratio of the signal power to the noise power

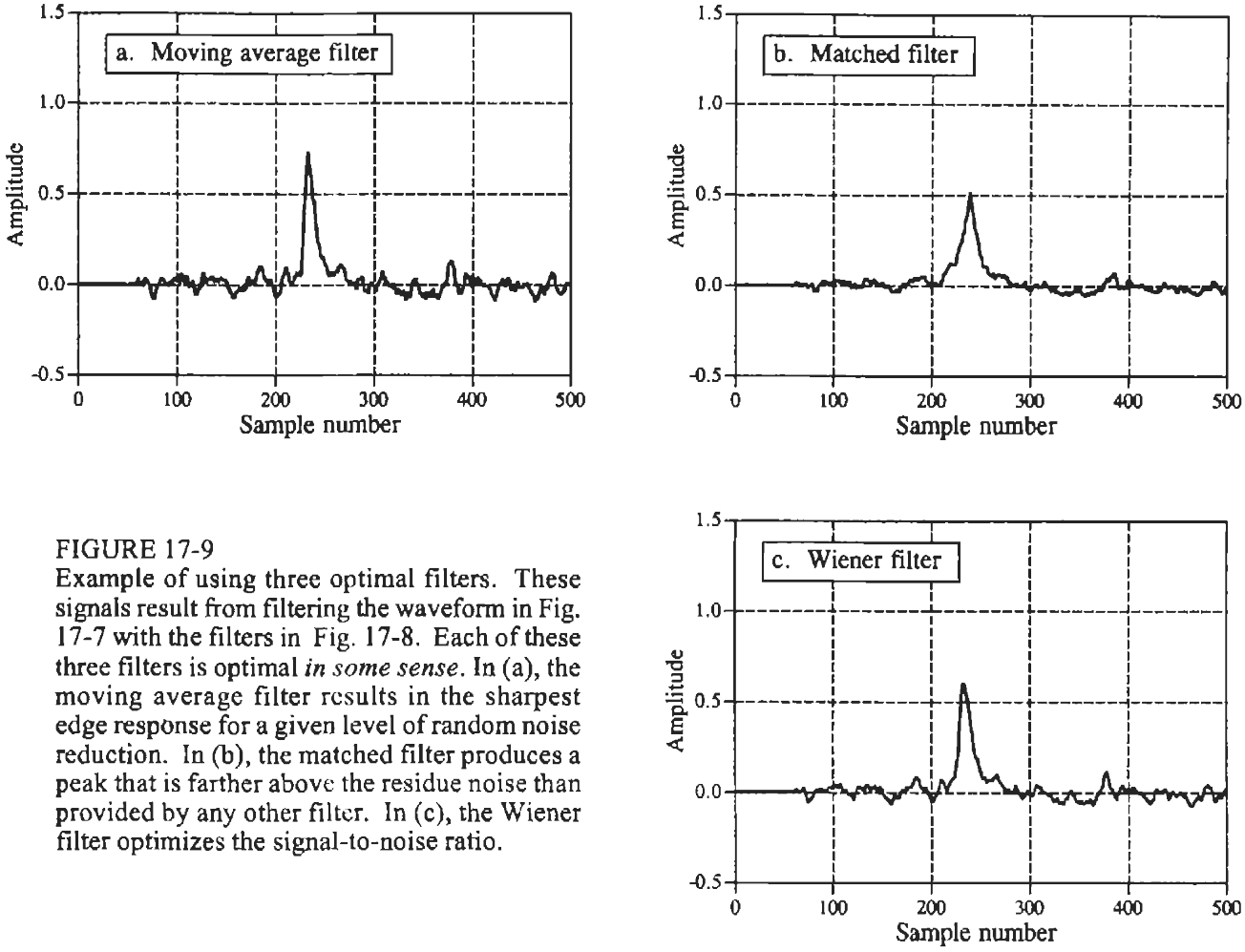

FIGURE 17-9
Example of using three optimal filters. These signals result from filtering the waveform in Fig. 17-7 with the filters in Fig. 17-8. Each of these three filters is optimal *in some sense*. In (a), the moving average filter results in the sharpest edge response for a given level of random noise reduction. In (b), the matched filter produces a peak that is farther above the residue noise than provided by any other filter. In (c), the Wiener filter optimizes the signal-to-noise ratio.

(over the length of the signal, not at each individual point). An appropriate filter kernel is designed from the Wiener frequency response using the custom method.

While the ideas behind these optimal filters are mathematically elegant, they often fail in practicality. This isn't to say they should never be used. The point is, don't hear the word "optimal" and stop thinking. Let's look at several reasons why you might *not* want to use them.

First, the difference between the signals in Fig. 17-9 is very unimpressive. In fact, if you weren't told what parameters were being optimized, you probably couldn't tell by looking at the signals. This is usually the case for problems involving overlapping frequency spectra. The small amount of extra performance obtained from an optimal filter may not be worth the the increased program complexity, the extra design effort, or the longer execution time.

Second: The Wiener and matched filters are completely determined by the characteristics of the problem. Other filters, such as the windowed-sinc and moving average, can be tailored to your liking. Optimal filter advocates would claim that this diddling can only reduce the effectiveness of the filter. This is

very arguable. Remember, each of these filters is optimal in one specific way (i.e., "in some sense"). This is seldom sufficient to claim that the entire problem has been optimized, especially if the resulting signals are interpreted by a human observer. For instance, a biomedical engineer might use a Wiener filter to maximize the signal-to-noise ratio in an electro-cardiogram. However, it is not obvious that this also optimizes a physician's ability to detect irregular heart activity by looking at the signal.

Third: The Wiener and matched filter must be carried out by *convolution*, making them extremely slow to execute. Even with the speed improvements discussed in the next chapter (FFT convolution), the computation time can be excessively long. In comparison, *recursive* filters (such as the moving average or others presented in Chapter 19) are much faster, and may provide an acceptable level of performance.

CHAPTER

18

FFT Convolution

This chapter presents two important DSP techniques, the *overlap-add method*, and *FFT convolution*. The overlap-add method is used to break long signals into smaller segments for easier processing. FFT convolution uses the overlap-add method together with the Fast Fourier Transform, allowing signals to be convolved by multiplying their frequency spectra. For filter kernels longer than about 64 points, FFT convolution is faster than standard convolution, while producing exactly the same result.

The Overlap-Add Method

There are many DSP applications where a long signal must be filtered in *segments*. For instance, high-fidelity digital *audio* requires a data rate of about 5 Mbytes/min, while digital *video* requires about 500 Mbytes/min. With data rates this high, it is common for computers to have insufficient memory to simultaneously hold the entire signal to be processed. There are also systems that process segment-by-segment because they operate in *real time*. For example, telephone signals cannot be delayed by more than a few hundred milliseconds, limiting the amount of data that is available for processing at any one instant. In still other applications, the *processing* may require that the signal be segmented. An example is FFT convolution, the main topic of this chapter.

The overlap-add method is based on the fundamental technique in DSP: (1) decompose the signal into simple components, (2) process each of the components in some useful way, and (3) recombine the processed components into the final signal. Figure 18-1 shows an example of how this is done for the overlap-add method. Figure (a) is the signal to be filtered, while (b) shows the filter kernel to be used, a windowed-sinc low-pass filter. Jumping to the bottom of the figure, (i) shows the filtered signal, a smoothed version of (a). The key to this method is how the *lengths* of these signals are affected by the convolution. When an N sample signal is convolved with an M sample

filter kernel, the output signal is $N+M-1$ samples long. For instance, the input signal, (a), is 300 samples (running from 0 to 299), the filter kernel, (b), is 101 samples (running from 0 to 100), and the output signal, (i), is 400 samples (running from 0 to 399).

In other words, when an N sample signal is filtered, it will be *expanded* by $M-1$ points *to the right*. (This is assuming that the filter kernel runs from index 0 to M. If negative indexes are used in the filter kernel, the expansion will also be to the *left*). In (a), zeros have been added to the signal between sample 300 and 399 to illustrate where this expansion will occur. Don't be confused by the small values at the ends of the output signal, (i). This is simply a result of the windowed-sinc filter kernel having small values near its ends. All 400 samples in (i) are nonzero, even though some of them are too small to be seen in the graph.

Figures (c), (d) and (e) show the decomposition used in the overlap-add method. The signal is broken into segments, with each segment having 100 samples from the original signal. In addition, 100 zeros are added to the right of each segment. In the next step, each segment is individually filtered by convolving it with the filter kernel. This produces the output segments shown in (f), (g), and (h). Since each input segment is 100 samples long, and the filter kernel is 101 samples long, each output segment will be 200 samples long. The important point to understand is that the 100 zeros were added to each input segment to allow for the expansion during the convolution.

Notice that the expansion results in the output segments *overlapping* each other. These overlapping output segments are added to give the output signal, (i). For instance, samples 200 to 299 in (i) are found by adding the corresponding samples in (g) and (h). The overlap-add method produces exactly the same output signal as direct convolution. The disadvantage is a much greater program complexity to keep track of the overlapping samples.

FFT Convolution

FFT convolution uses the principle that *multiplication* in the frequency domain corresponds to *convolution* in the time domain. The input signal is transformed into the frequency domain using the DFT, multiplied by the frequency response of the filter, and then transformed back into the time domain using the inverse DFT. This basic technique was known since the days of Fourier; however, no one really cared. This is because the time required to calculate the DFT was *longer* than the time to directly calculate the convolution. This changed in 1965 with the development of the fast Fourier transform (FFT). By using the FFT algorithm to calculate the DFT, convolution via the frequency domain can be *faster* than directly convolving the time domain signals. The final result is the same; only the number of calculations has been changed by a more efficient algorithm. For this reason, FFT convolution is also called **high-speed convolution**.

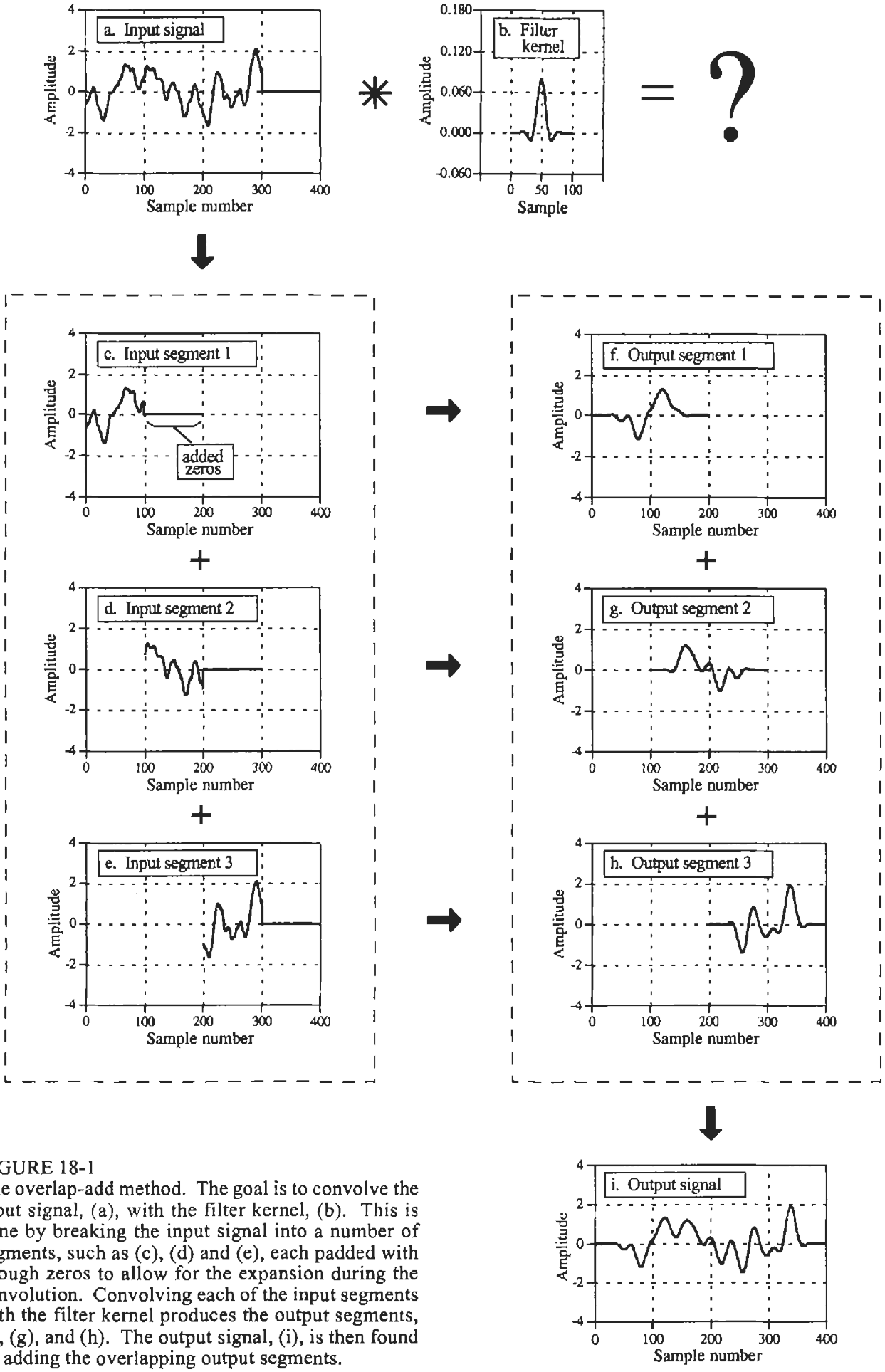

FIGURE 18-1
The overlap-add method. The goal is to convolve the input signal, (a), with the filter kernel, (b). This is done by breaking the input signal into a number of segments, such as (c), (d) and (e), each padded with enough zeros to allow for the expansion during the convolution. Convolving each of the input segments with the filter kernel produces the output segments, (f), (g), and (h). The output signal, (i), is then found by adding the overlapping output segments.

FFT convolution uses the overlap-add method shown in Fig. 18-1; only the way that the input segments are converted into the output segments is changed. Figure 18-2 shows an example of how an input segment is converted into an output segment by FFT convolution. To start, the frequency response of the filter is found by taking the DFT of the filter kernel, using the FFT. For instance, (a) shows an example filter kernel, a windowed-sinc band-pass filter. The FFT converts this into the real and imaginary parts of the frequency response, shown in (b) & (c). These frequency domain signals may not *look* like a band-pass filter because they are in rectangular form. Remember, polar form is usually best for humans to understand the frequency domain, while rectangular form is normally best for mathematical calculations. These real and imaginary parts are stored in the computer for use when each segment is being calculated.

Figure (d) shows the input segment to being processed. The FFT is used to find its frequency spectrum, shown in (e) & (f). The frequency spectrum of the output segment, (h) & (i) is then found by multiplying the filter's frequency response, (b) & (c), by the spectrum of the input segment, (e) & (f). Since these spectra consist of real and imaginary parts, they are multiplied according to Eq. 9-1 in Chapter 9. The inverse FFT is then used to find the output segment, (g), from its frequency spectrum, (h) & (i). It is important to recognize that this output segment is exactly the same as would be obtained by the direct convolution of the input segment, (d), and the filter kernel, (a).

The FFTs must be long enough that *circular convolution* does not take place (also described in Chapter 9). This means that the FFT should be the same length as the output segment, (g). For instance, in the example of Fig. 18-2, the filter kernel contains 129 points and each segment contains 128 points, making output segment 256 points long. This calls for 256-point FFTs to be used. This means that the filter kernel, (a), must be padded with 127 zeros to bring it to a total length of 256 points. Likewise, each of the input segments, (d), must be padded with 128 zeros. As another example, imagine you need to convolve a very long signal with a filter kernel having 600 samples. One alternative would be to use segments of 425 points, and 1024-point FFTs. Another alternative would be to use segments of 1449 points, and 2048-point FFTs.

Table 18-1 shows an example program to carry out FFT convolution. This program filters a 10 million point signal by convolving it with a 400 point filter kernel. This is done by breaking the input signal into 16000 segments, with each segment having 625 points. When each of these segments is convolved with the filter kernel, an output segment of $625 + 400 - 1 = 1024$ points is produced. Thus, 1024-point FFTs are used. After defining and initializing all the arrays (lines 130 to 230), the first step is to calculate and store the frequency response of the filter (lines 250 to 310). Line 260 calls a mythical subroutine that loads the filter kernel into XX[0] through XX[399], and sets XX[400] through XX[1023] to a value of zero. The subroutine in line 270 is the FFT, transforming the 1024 samples held in XX[ ] into the 513 samples held in REX[ ] & IMX[ ], the real and

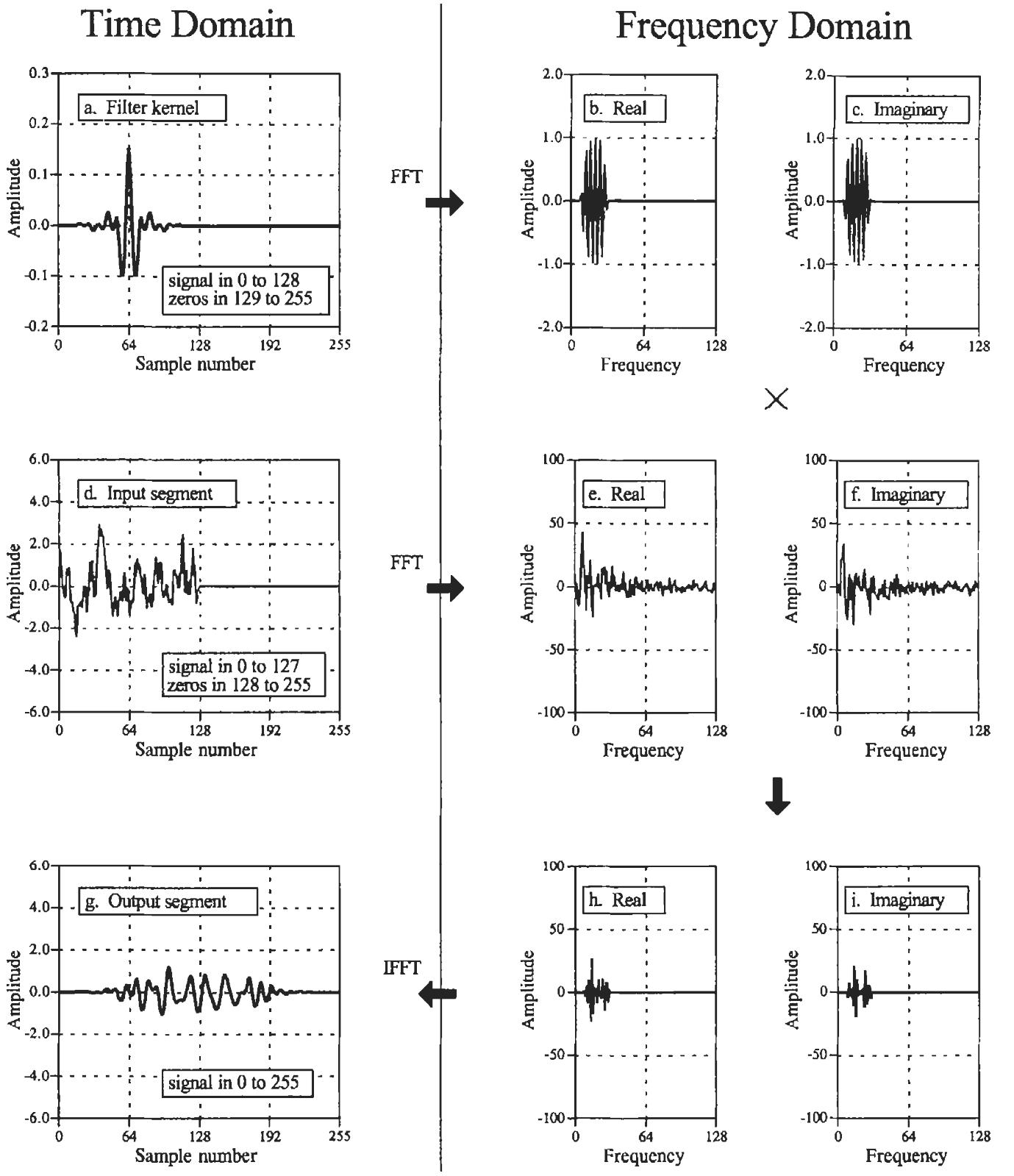

FIGURE 18-2
FFT convolution. The filter kernel, (a), and the signal segment, (d), are converted into their respective spectra, (b) & (c) and (e) & (f), via the FFT. These spectra are multiplied, resulting in the spectrum of the output segment, (h) & (i). The Inverse FFT then finds the output segment, (g).

imaginary parts of the frequency response. These values are transferred into the arrays REFR[ ] & IMFR[ ] (for: REal and IMaginary Frequency Response), to be used later in the program.

The FOR-NEXT loop between lines 340 and 580 controls how the 16000 segments are processed. In line 360, a subroutine loads the next segment to be processed into XX[0] through XX[624], and sets XX[625] through XX[1023] to a value of zero. In line 370, the FFT subroutine is used to find this segment's frequency spectrum, with the real part being placed in the 513 points of REX[ ], and the imaginary part being placed in the 513 points of IMX[ ]. Lines 390 to 430 show the multiplication of the segment's frequency spectrum, held in REX[ ] & IMX[ ], by the filter's frequency response, held in REFR[ ] and IMFR[ ]. The result of the multiplication is stored in REX[ ] & IMX[ ], overwriting the data previously there. Since this is now the frequency spectrum of the output segment, the IFFT can be used to find the output segment. This is done by the mythical IFFT subroutine in line 450, which transforms the 513 points held in REX[ ] & IMX[ ] into the 1024 points held in XX[ ], the output segment.

Lines 470 to 550 handle the overlapping of the segments. Each output segment is divided into two sections. The first 625 points (0 to 624) need to be combined with the overlap from the *previous* output segment, and then written to the output signal. The last 399 points (625 to 1023) need to be saved so that they can overlap with the *next* output segment.

To understand this, look back at Fig 18-1. Samples 100 to 199 in (g) need to be combined with the overlap from the *previous* output segment, (f), and can then be moved to the output signal (i). In comparison, samples 200 to 299 in (g) need to be saved so that they can be combined with the *next* output segment, (h).

Now back to the program. The array OLAP[ ] is used to hold the 399 samples that overlap from one segment to the next. In lines 470 to 490 the 399 values in this array (from the previous output segment) are added to the output segment currently being worked on, held in XX[ ]. The mythical subroutine in line 550 then outputs the 625 samples in XX[0] to XX[624] to the file holding the output signal. The 399 samples of the current output segment that need to be held over to the next output segment are then stored in OLAP[ ] in lines 510 to 530.

After all 0 to 15999 segments have been processed, the array, OLAP[ ], will contain the 399 samples from segment 15999 that should overlap segment 16000. Since segment 16000 doesn't exist (or can be viewed as containing all zeros), the 399 samples are written to the output signal in line 600. This makes the length of the output signal $16000 \times 625 + 399 = 10{,}000{,}399$ points. This matches the length of input signal, plus the length of the filter kernel, minus 1.

Speed Improvements

When is FFT convolution faster than standard convolution? The answer depends on the length of the filter kernel, as shown in Fig. 18-3. The time

```
100 'FFT CONVOLUTION
110 'This program convolves a 10 million point signal with a 400 point filter kernel. The input
120 'signal is broken into 16000 segments, each with 625 points. 1024 point FFTs are used.
130 '
130 '                                   'INITIALIZE THE ARRAYS
140 DIM XX[1023]                        'the time domain signal (for the FFT)
150 DIM REX[512]                        'real part of the frequency domain (for the FFT)
160 DIM IMX[512]                        'imaginary part of the frequency domain (for the FFT)
170 DIM REFR[512]                       'real part of the filter's frequency response
180 DIM IMFR[512]                       'imaginary part of the filter's frequency response
190 DIM OLAP[398]                       'holds the overlapping samples from segment to segment
200 '
210 FOR I% = 0 TO 398                   'zero the array holding the overlapping samples
220  OLAP[I%] = 0
230 NEXT I%
240 '
250 '                                   'FIND & STORE THE FILTER'S FREQUENCY RESPONSE
260 GOSUB XXXX                          'Mythical subroutine to load the filter kernel into XX[ ]
270 GOSUB XXXX                          'Mythical FFT subroutine:  XX[ ] --> REX[ ] & IMX[ ]
280 FOR F% = 0 TO 512                   'Save the frequency response in REFR[ ] & IMFR[ ]
290  REFR[F%] = REX[F%]
300  IMFR[F%] = IMX[F%]
310 NEXT F%
320 '
330 '                                   'PROCESS EACH OF THE 16000 SEGMENTS
340 FOR SEGMENT% = 0 TO 15999
350  '
360  GOSUB XXXX                         'Mythical subroutine to load the next input segment into XX[ ]
370  GOSUB XXXX                         'Mythical FFT subroutine:  XX[ ]  -->  REX[ ] & IMX[ ]
380  '
390  FOR F% = 0 TO 512  'Multiply the frequency spectrum by the frequency response
400   TEMP    = REX[F%]*REFR[F%] - IMX[F%]*IMFR[F%]
410   IMX[F%]  = REX[F%]*IMFR[F%] + IMX[F%]*REFR[F%]
420   REX[F%]  = TEMP
430  NEXT F%
440  '
450  GOSUB XXXX                         'Mythical IFFT subroutine:  REX[ ] & IMX[ ]  -->  XX[ ]
460  '
470  FOR I% = 0 TO 398   'Add the last segment's overlap to this segment
480   XX[I%] = XX[I%] + OLAP[I%]
490  NEXT I%
500  '
510  FOR I% = 625 TO 1023               'Save the samples that will overlap the next segment
520   OLAP[I%-625] = XX[I%]
530  NEXT I%
540  '
550  GOSUB XXXX                         'Mythical subroutine to output the 625 samples stored
560  '                                  'in XX[0] to XX[624]
570  '
580 NEXT SEGMENT%
590 '
600 GOSUB XXXX                          'Mythical subroutine to output all 399 samples in OLAP[ ]
610 END
```

TABLE 18-1

for standard convolution is directly proportional to the number of points in the filter kernel. In comparison, the time required for FFT convolution increases very slowly, only as the *logarithm* of the number of points in the

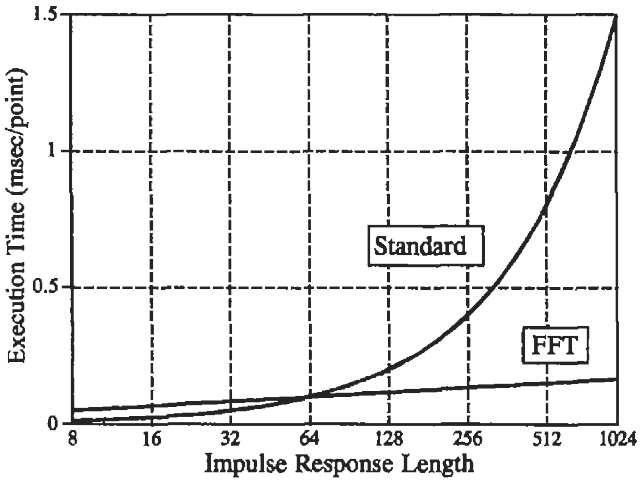

FIGURE 18-3
Execution times for FFT convolution. FFT convolution is faster than the standard method when the filter kernel is longer than about 60 points. These execution times are for a 100 MHz Pentium, using single precision floating point.

filter kernel. The crossover occurs when the filter kernel has about 40 to 80 samples (depending on the particular hardware used).

The important idea to remember: filter kernels shorter than about 60 points can be implemented faster with standard convolution, and the execution time is proportional to the kernel length. Longer filter kernels can be implemented faster with FFT convolution. With FFT convolution, the filter kernel can be made as long as you like, with very little penalty in execution time. For instance, a 16,000-point filter kernel only requires about *twice* as long to execute as one with only 64 points.

The *speed* of the convolution also dictates the *precision* of the calculation (just as described for the FFT in Chapter 12). This is because the round-off error in the output signal depends on the total number of calculations, which is directly proportional to the computation time. If the output signal is calculated *faster*, it will also be calculated more *precisely*. For instance, imagine convolving a signal with a 1000-point filter kernel, with single precision floating point. Using standard convolution, the typical round-off noise can be expected to be about 1 part in 20,000 (from the guidelines in Chapter 4). In comparison, FFT convolution can be expected to be an order of magnitude *faster*, and an order of magnitude more *precise* (i.e., 1 part in 200,000).

Keep FFT convolution tucked away for when you have a large amount of data to process and need an extremely long filter kernel. Think in terms of a *million* sample signal and a *thousand* point filter kernel. Anything less won't justify the extra programming effort. Don't want to write your own FFT convolution routine? Look in software libraries and packages for prewritten code.

CHAPTER

19

Recursive Filters

Recursive filters are an efficient way of achieving a long impulse response, without having to perform a long convolution. They execute very rapidly, but have less performance and flexibility than other digital filters. Recursive filters are also called *Infinite Impulse Response* (IIR) filters, since their impulse responses are composed of decaying exponentials. This distinguishes them from digital filters carried out by convolution, called *Finite Impulse Response* (FIR) filters. This chapter is an introduction to how recursive filters operate, and how simple members of the family can be designed. Chapters 20, 26 and 33 present more sophisticated design methods.

The Recursive Method

To start the discussion of recursive filters, imagine that you need to extract information from some signal, $x[\ ]$. Your need is so great that you hire an old mathematics professor to process the data for you. The professor's task is to filter $x[\ ]$ to produce $y[\ ]$, which hopefully contains the information you are interested in. The professor begins his work of calculating each point in $y[\ ]$ according to some algorithm that is locked tightly in his over-developed brain. Part way through the task, a most unfortunate event occurs. The professor begins to babble about analytic singularities and fractional transforms, and other demons from a mathematician's nightmare. It is clear that the professor has lost his mind. You watch with anxiety as the professor, and your algorithm, are taken away by several men in white coats.

You frantically review the professor's notes to find the algorithm he was using. You find that he had completed the calculation of points $y[0]$ through $y[27]$, and was about to start on point $y[28]$. As shown in Fig. 19-1, we will let the variable, n, represent the point that is currently being calculated. This means that $y[n]$ is sample 28 in the output signal, $y[n-1]$ is sample 27, $y[n-2]$ is sample 26, etc. Likewise, $x[n]$ is point 28 in the input signal,

$x[n-1]$ is point 27, etc. To understand the algorithm being used, we ask ourselves: "What information was available to the professor to calculate $y[n]$, the sample currently being worked on?"

The most obvious source of information is the *input signal*, that is, the values: $x[n]$, $x[n-1]$, $x[n-2]$,···. The professor could have been multiplying each point in the input signal by a coefficient, and adding the products together:

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + \cdots$$

You should recognize that this is nothing more than simple convolution, with the coefficients: a_0, a_1, a_2,···, forming the convolution kernel. If this was all the professor was doing, there wouldn't be much need for this story, or this chapter. However, there is another source of information that the professor had access to: the *previously* calculated values of the output signal, held in: $y[n-1]$, $y[n-2]$, $y[n-3]$,···. Using this additional information, the algorithm would be in the form:

$$\begin{aligned} y[n] = {} & a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + \cdots \\ & + b_1 y[n-1] + b_2 y[n-2] + b_3 y[n-3] + \cdots \end{aligned}$$

EQUATION 19-1
The recursion equation. In this equation, $x[\ ]$ is the input signal, $y[\ ]$ is the output signal, and the a's and b's are coefficients.

In words, each point in the output signal is found by multiplying the values from the input signal by the "a" coefficients, multiplying the previously calculated values from the output signal by the "b" coefficients, and adding the products together. Notice that there isn't a value for b_0, because this corresponds to the sample being calculated. Equation 19-1 is called the **recursion equation**, and filters that use it are called **recursive filters**. The "a" and "b" values that define the filter are called the **recursion coefficients**. In actual practice, no more than about a dozen recursion coefficients can be used or the filter becomes unstable (i.e., the output continually increases or oscillates). Table 19-1 shows an example recursive filter program.

Recursive filters are useful because they *bypass* a longer convolution. For instance, consider what happens when a delta function is passed through a recursive filter. The output is the filter's *impulse response*, and will typically be a sinusoidal oscillation that exponentially decays. Since this impulse response in infinitely long, recursive filters are often called *infinite impulse response* (IIR) filters. In effect, recursive filters *convolve* the input signal with a very long filter kernel, although only a few coefficients are involved.

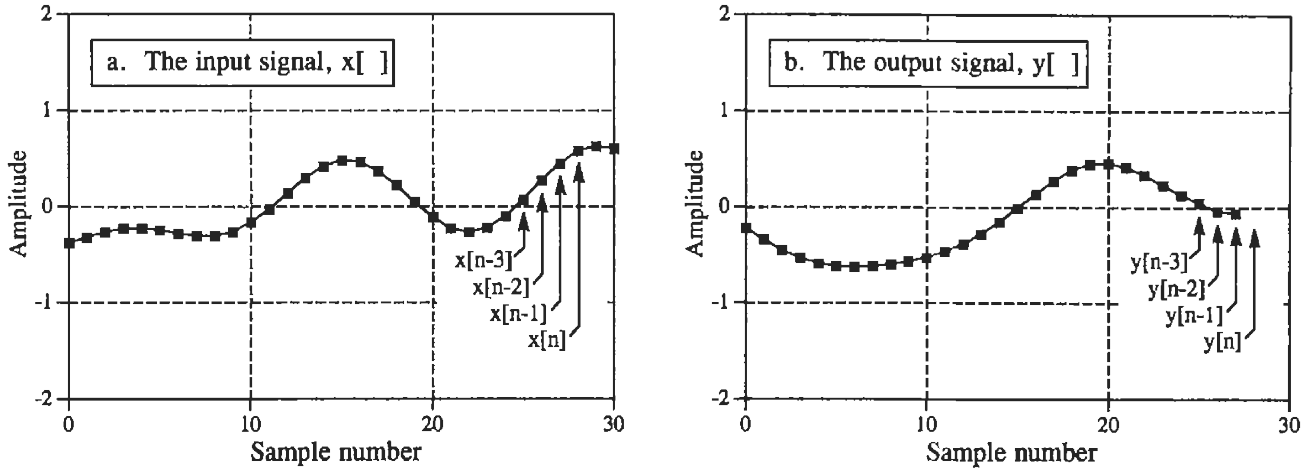

FIGURE 19-1
Recursive filter notation. The output sample being calculated, $y[n]$, is determined by the values from the input signal, $x[n]$, $x[n-1]$, $x[n-2]$, ···, as well as the *previously* calculated values in the output signal, $y[n-1]$, $y[n-2]$, $y[n-3]$, ···. These figures are shown for $n = 28$.

The relationship between the recursion coefficients and the filter's response is given by a mathematical technique called the **z-transform**, the topic of Chapter 33. For example, the z-transform can be used for such tasks as: converting between the recursion coefficients and the frequency response, combining cascaded and parallel stages into a single filter, designing recursive systems that mimic analog filters, etc. Unfortunately, the z-transform is very mathematical, and more complicated than most DSP *users* are willing to deal with. This is the realm of those that specialize in DSP.

There are three ways to find the recursion coefficients without having to understand the z-transform. First, this chapter provides design equations for several types of simple recursive filters. Second, Chapter 20 provides a "cookbook" computer program for designing the more sophisticated *Chebyshev* low-pass and high-pass filters. Third, Chapter 26 describes an iterative method for designing recursive filters with an *arbitrary* frequency response.

```
100 'RECURSIVE FILTER
110 '
120 DIM X[499]                  'holds the input signal
130 DIM Y[499]                  'holds the filtered output signal
140 '
150 GOSUB XXXX                  'mythical subroutine to calculate the recursion
160 '                           'coefficients: A0, A1, A2, B1, B2
170 '
180 GOSUB XXXX                  'mythical subroutine to load X[ ] with the input data
190 '
200 FOR I% = 2 TO 499
210   Y[I%] = A0*X[I%] + A1*X[I%-1] + A2*X[I%-2] + B1*Y[I%-1] + B2*Y[I%-2]
220 NEXT I%
230 '
240 END
```

TABLE 19-1

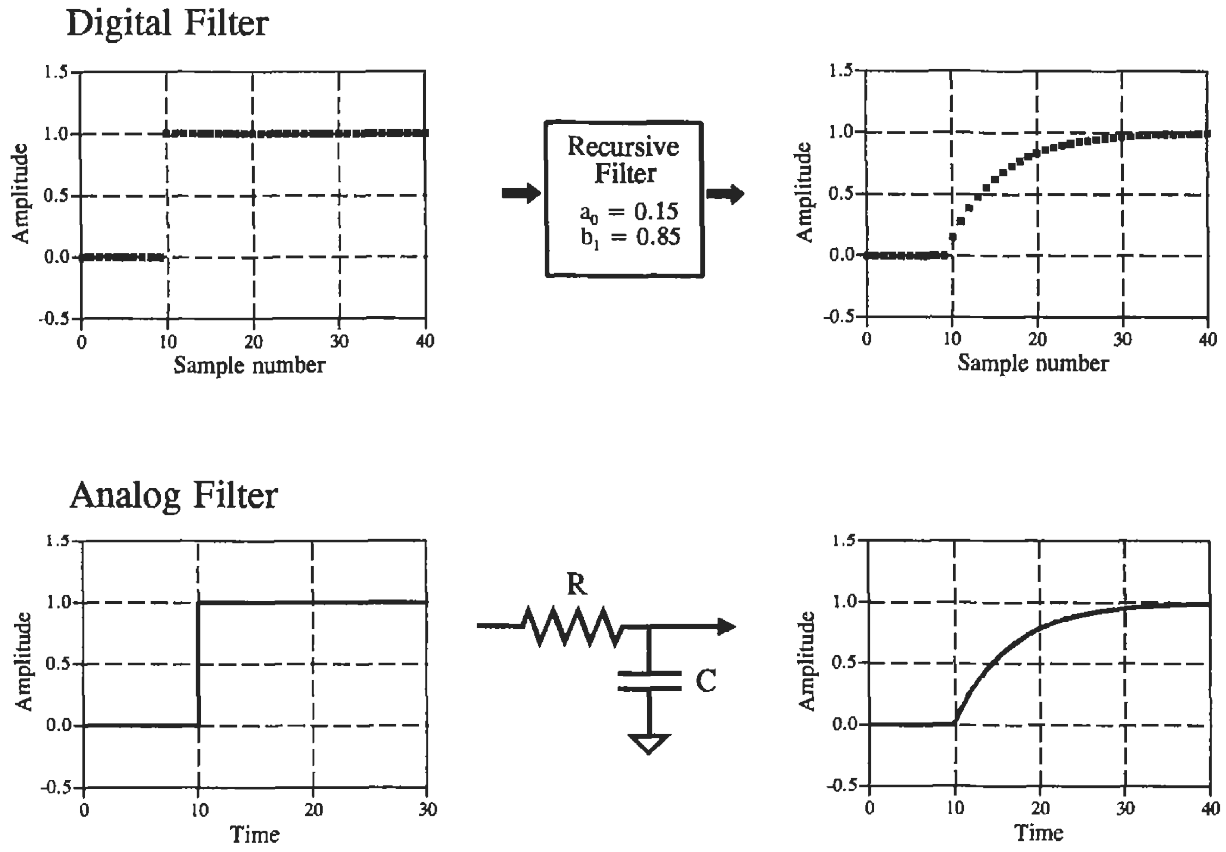

FIGURE 19-2
Single pole low-pass filter. Digital recursive filters can mimic analog filters composed of resistors and capacitors. As shown in this example, a single pole low-pass recursive filter smoothes the edge of a step input, just as an electronic RC filter.

Single Pole Recursive Filters

Figure 19-2 shows an example of what is called a **single pole** low-pass filter. This recursive filter uses just two coefficients, $a_0 = 0.15$ and $b_1 = 0.85$. For this example, the input signal is a step function. As you should expect for a low-pass filter, the output is a smooth rise to the steady state level. This figure also shows something that ties into your knowledge of electronics. This low-pass recursive filter is completely analogous to an electronic low-pass filter composed of a single resistor and capacitor.

The beauty of the recursive method is in its ability to create a wide variety of responses by changing only a few parameters. For example, Fig. 19-3 shows a filter with three coefficients: $a_0 = 0.93$, $a_1 = -0.93$ and $b_1 = 0.86$. As shown by the similar step responses, this digital filter mimics an electronic RC high-pass filter.

These single pole recursive filters are definitely something you want to keep in your DSP toolbox. You can use them to process digital signals just as you would use RC networks to process analog electronic signals. This includes everything you would expect: DC removal, high-frequency noise suppression, wave shaping, smoothing, etc. They are easy to program, fast

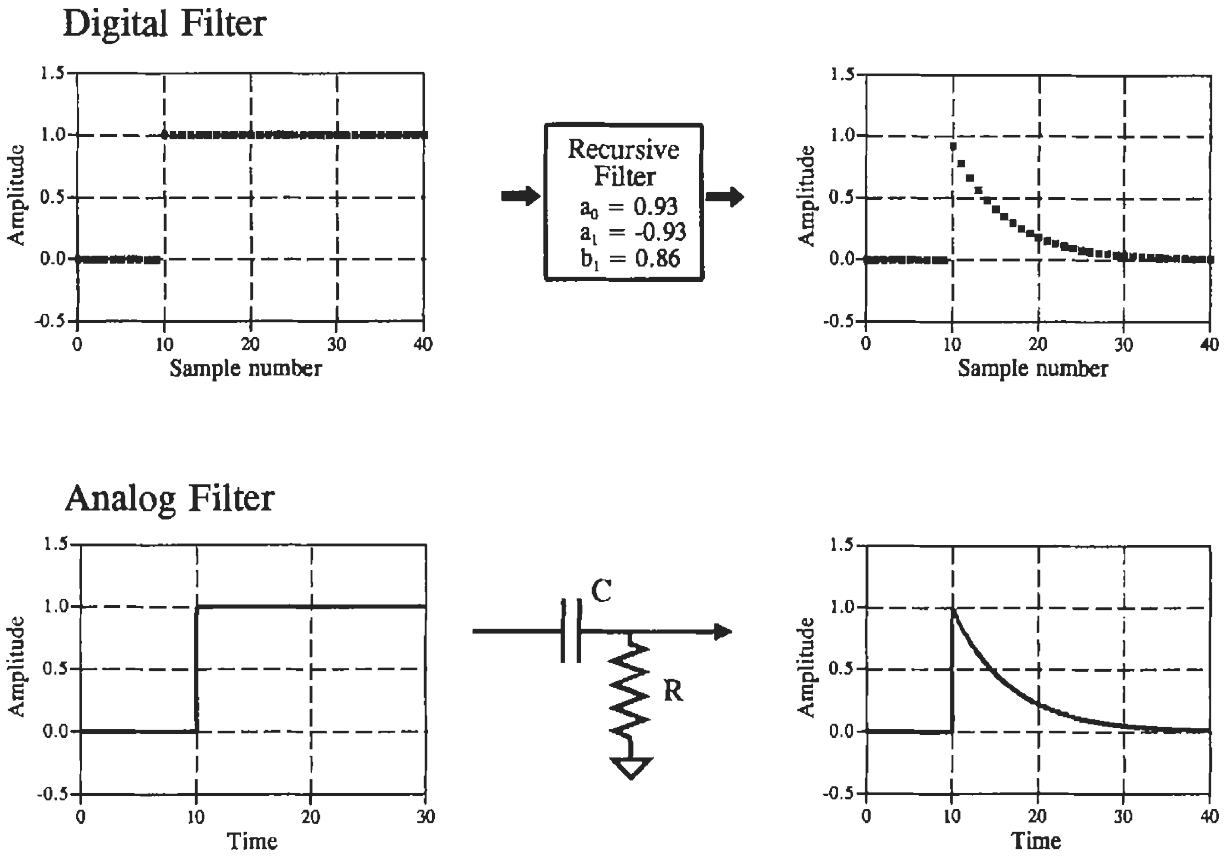

FIGURE 19-3
Single pole high-pass filter. Proper coefficient selection can also make the recursive filter mimic an electronic RC high-pass filter. These single pole recursive filters can be used in DSP just as you would use RC circuits in analog electronics.

to execute, and produce few surprises. The coefficients are found from these simple equations:

EQUATION 19-2
Single pole low-pass filter. The filter's response is controlled by the parameter, x, a value between zero and one.

$$a_0 = 1 - x$$
$$b_1 = x$$

EQUATION 19-3
Single pole high-pass filter.

$$a_0 = (1+x)/2$$
$$a_1 = -(1+x)/2$$
$$b_1 = x$$

The characteristics of these filters are controlled by the parameter, x, a value between zero and one. Physically, x is the amount of *decay* between adjacent samples. For instance, x is 0.86 in Fig. 19-3, meaning that the value of each sample in the output signal is 0.86 the value of the sample before it. The higher the value of x, the slower the decay. Notice that the

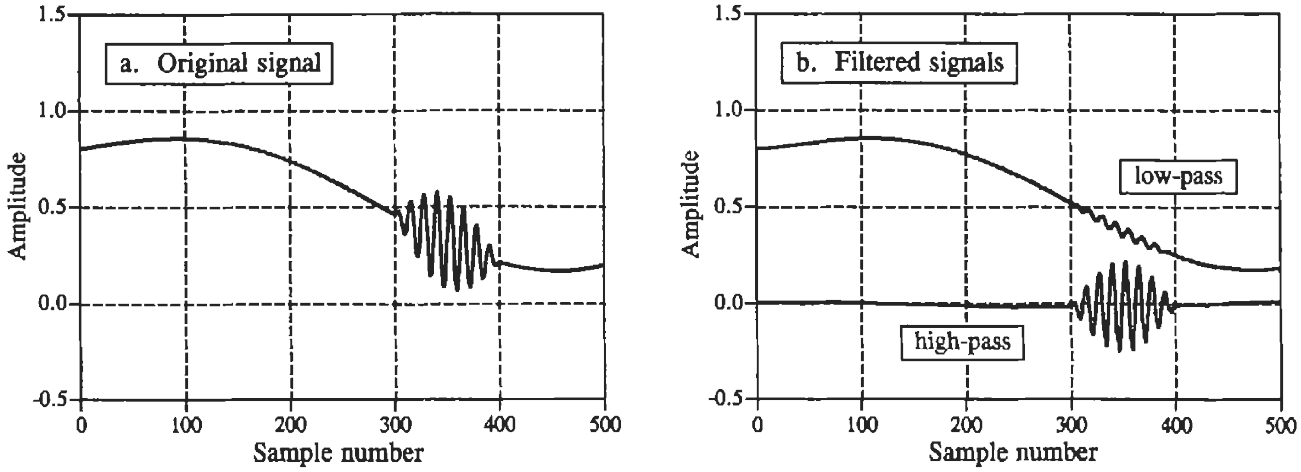

FIGURE 19-4
Example of single pole recursive filters. In (a), a high frequency burst rides on a slowly varying signal. In (b), single pole low-pass and high-pass filters are used to separate the two components. The low-pass filter uses $x = 0.95$, while the high-pass filter is for $x = 0.86$.

filter becomes *unstable* if x is made greater than one. That is, any nonzero value on the input will make the output increase until an overflow occurs.

The value for x can be directly specified, or found from the desired *time constant* of the filter. Just as $R \times C$ is the number of seconds it takes an RC circuit to decay to 36.8% of its final value, d is the number of samples it takes for a recursive filter to decay to this same level:

EQUATION 19-4
Time constant of single pole filters. This equation relates the amount of decay between samples, x, with the filter's time constant, d, the number of samples for the filter to decay to 36.8%.

$$x = e^{-1/d}$$

For instance, a sample-to-sample decay of $x = 0.86$ corresponds to a time constant of $d = 6.63$ samples (as shown in Fig 19-3). There is also a fixed relationship between x and the -3dB *cutoff frequency*, f_C, of the digital filter:

EQUATION 19-5
Cutoff frequency of single pole filters. The amount of decay between samples, x, is related to the cutoff frequency of the filter, f_C, a value between 0 and 0.5.

$$x = e^{-2\pi f_C}$$

This provides three ways to find the "a" and "b" coefficients, starting with the time constant, the cutoff frequency, or just directly picking x.

Figure 19-4 shows an example of using single pole recursive filters. In (a), the original signal is a smooth curve, except a burst of a high frequency sine wave. Figure (b) shows the signal after passing through low-pass and high-pass filters. The signals have been separated fairly well, but not perfectly, just as if simple RC circuits were used on an analog signal.

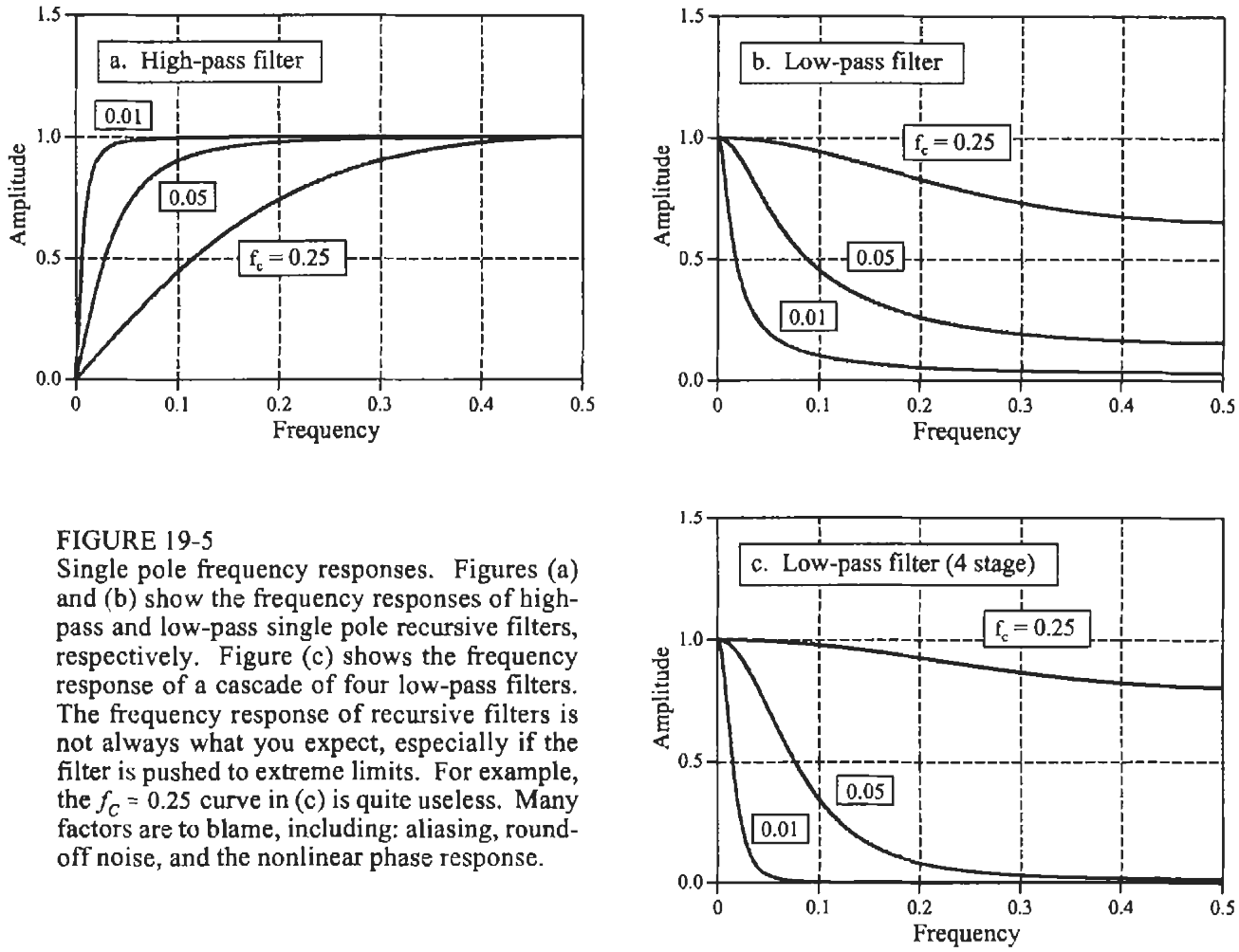

FIGURE 19-5
Single pole frequency responses. Figures (a) and (b) show the frequency responses of high-pass and low-pass single pole recursive filters, respectively. Figure (c) shows the frequency response of a cascade of four low-pass filters. The frequency response of recursive filters is not always what you expect, especially if the filter is pushed to extreme limits. For example, the f_C = 0.25 curve in (c) is quite useless. Many factors are to blame, including: aliasing, round-off noise, and the nonlinear phase response.

Figure 19-5 shows the frequency responses of various single pole recursive filters. These curves are obtained by passing a delta function through the filter to find the filter's impulse response. The FFT is then used to convert the impulse response into the frequency response. In principle, the impulse response is infinitely long; however, it decays below the single precision round-off noise after about 15 to 20 time constants. For example, when the time constant of the filter is d = 6.63 samples, the impulse response can be contained in about 128 samples.

The key feature in Fig. 19-5 is that single pole recursive filters have little ability to separate one band of frequencies from another. In other words, they perform well in the time domain, and poorly in the frequency domain. The frequency response can be improved slightly by cascading several stages. This can be accomplished in two ways. First, the signal can be passed through the filter several times. Second, the z-transform can be used to find the recursion coefficients that combine the cascade into a single stage. Both ways work and are commonly used. Figure (c) shows the frequency response of a cascade of four low-pass filters. Although the stopband attenuation is significantly improved, the roll-off is still terrible. If you need better performance in the frequency domain, look at the Chebyshev filters of the next chapter.

The four stage low-pass filter is comparable to the Blackman and Gaussian filters (relatives of the moving average, Chapter 15), but with a much faster execution speed. The design equations for a four stage low-pass filter are:

EQUATION 19-6
Four stage low-pass filter. These equations provide the "a" and "b" coefficients for a cascade of four single pole low-pass filters. The relationship between x and the cutoff frequency of this filter is given by Eq. 19-5, with the 2π replaced by 14.445.

$$\begin{aligned} a_0 &= (1-x)^4 \\ b_1 &= 4x \\ b_2 &= -6x^2 \\ b_3 &= 4x^3 \\ b_4 &= -x^4 \end{aligned}$$

Narrow-band Filters

A common need in electronics and DSP is to isolate a narrow band of frequencies from a wider bandwidth signal. For example, you may want to eliminate 60 hertz interference in an instrumentation system, or isolate the signaling tones in a telephone network. Two types of frequency responses are available: the *band-pass* and the *band-reject* (also called a **notch filter**). Figure 19-6 shows the frequency response of these filters, with the recursion coefficients provided by the following equations:

EQUATION 19-7
Band-pass filter. An example frequency response is shown in Fig. 19-6a. To use these equations, first select the center frequency, f, and the bandwidth, BW. Both of these are expressed as a fraction of the sampling rate, and therefore in the range of 0 to 0.5. Next, calculate R, and then K, and then the recursion coefficients.

$$\begin{aligned} a_0 &= 1-K \\ a_1 &= 2(K-R)\cos(2\pi f) \\ a_2 &= R^2-K \\ b_1 &= 2R\cos(2\pi f) \\ b_2 &= -R^2 \end{aligned}$$

EQUATION 19-8
Band-reject filter. This filter is commonly called a notch filter. Example frequency responses are shown in Fig. 19-6b.

$$\begin{aligned} a_0 &= K \\ a_1 &= -2K\cos(2\pi f) \\ a_2 &= K \\ b_1 &= 2R\cos(2\pi f) \\ b_2 &= -R^2 \end{aligned}$$

where:

$$K = \frac{1 - 2R\cos(2\pi f) + R^2}{2 - 2\cos(2\pi f)}$$

$$R = 1 - 3BW$$

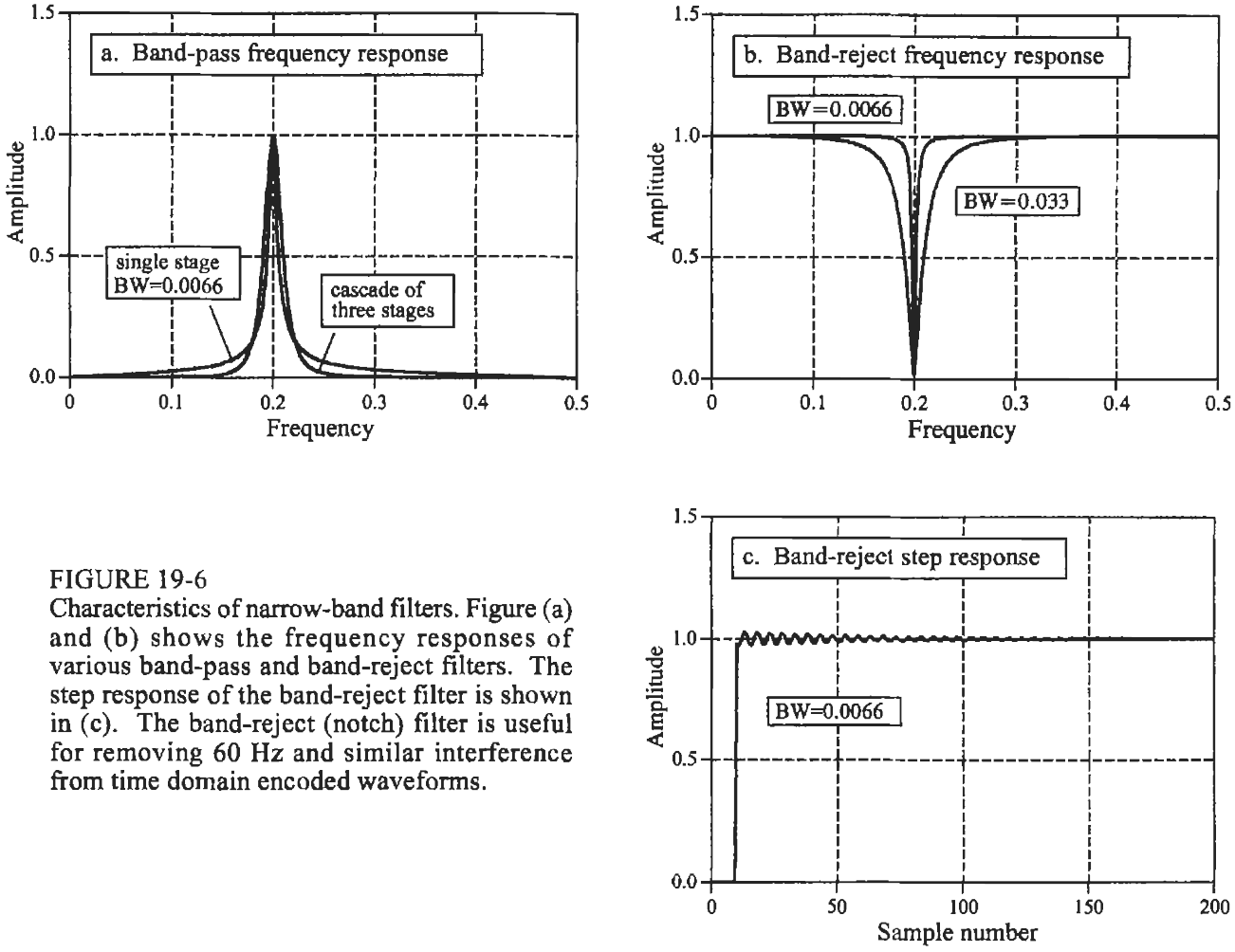

FIGURE 19-6
Characteristics of narrow-band filters. Figure (a) and (b) shows the frequency responses of various band-pass and band-reject filters. The step response of the band-reject filter is shown in (c). The band-reject (notch) filter is useful for removing 60 Hz and similar interference from time domain encoded waveforms.

Two parameters must be selected before using these equations: f, the center frequency, and BW, the bandwidth (measured at an amplitude of 0.707). Both of these are expressed as a fraction of the sampling frequency, and therefore must be between 0 and 0.5. From these two specified values, calculate the intermediate variables: R and K, and then the recursion coefficients.

As shown in (a), the band-pass filter has relatively large *tails* extending from the main peak. This can be improved by cascading several stages. Since the design equations are quite long, it is simpler to implement this cascade by filtering the signal several times, rather than trying to find the coefficients needed for a single filter.

Figure (b) shows examples of the band-reject filter. The narrowest bandwidth that can be obtain with single precision is about 0.0003 of the sampling frequency. When pushed beyond this limit, the attenuation of the notch will degrade. Figure (c) shows the step response of the band-reject filter. There is noticeable overshoot and ringing, but its amplitude is quite small. This allows the filter to remove narrowband interference (60 Hz and the like) with only a minor distortion to the time domain waveform.

Phase Response

There are three types of *phase response* that a filter can have: **zero phase**, **linear phase**, and **nonlinear phase**. An example of each of these is shown in Figure 19-7. As shown in (a), the *zero phase* filter is characterized by an impulse response that is symmetrical around sample zero. The actual shape doesn't matter, only that the negative numbered samples are a mirror image of the positive numbered samples. When the Fourier transform is taken of this symmetrical waveform, the phase will be entirely zero, as shown in (b).

The disadvantage of the zero phase filter is that it requires the use of negative indexes, which can be inconvenient to work with. The linear phase filter is a way around this. The impulse response in (d) is identical to that shown in (a), except it has been shifted to use only positive numbered samples. The impulse response is still symmetrical between the left and right; however, the location of symmetry has been shifted from zero. This shift results in the phase, (e), being a *straight line*, accounting for the name: *linear phase*. The slope of this straight line is directly proportional to the amount of the shift. Since the shift in the impulse response does nothing but produce an identical shift in the output signal, the linear phase filter is equivalent to the zero phase filter for most purposes.

Figure (g) shows an impulse response that is *not* symmetrical between the left and right. Correspondingly, the phase, (h), is *not* a straight line. In other words, it has a *nonlinear phase*. Don't confuse the terms: *nonlinear and linear phase* with the concept of *system linearity* discussed in Chapter 5. Although both use the word *linear*, they are not related.

Why does anyone care if the phase is linear or not? Figures (c), (f), and (i) show the answer. These are the **pulse responses** of each of the three filters. The pulse response is nothing more than a positive-going step response followed by a negative-going step response. The pulse response is used here because it displays what happens to both the rising and falling edges in a signal. Here is the important part: zero and linear phase filters have left and right edges that look the *same*, while nonlinear phase filters have left and right edges that look *different*. Many applications cannot tolerate the left and right edges looking different. One example is the display of an oscilloscope, where this difference could be misinterpreted as a feature of the signal being measured. Another example is in video processing. Can you imagine turning on your TV to find the left ear of your favorite actor looking different from his right ear?

It is easy to make an FIR (finite impulse response) filter have a linear phase. This is because the impulse response (filter kernel) is directly *specified* in the design process. Making the filter kernel have left-right symmetry is all that is required. This is not the case with IIR (recursive) filters, since the recursion coefficients are what is specified, not the impulse response. The impulse response of a recursive filter is *not* symmetrical between the left and right, and therefore has a *nonlinear* phase.

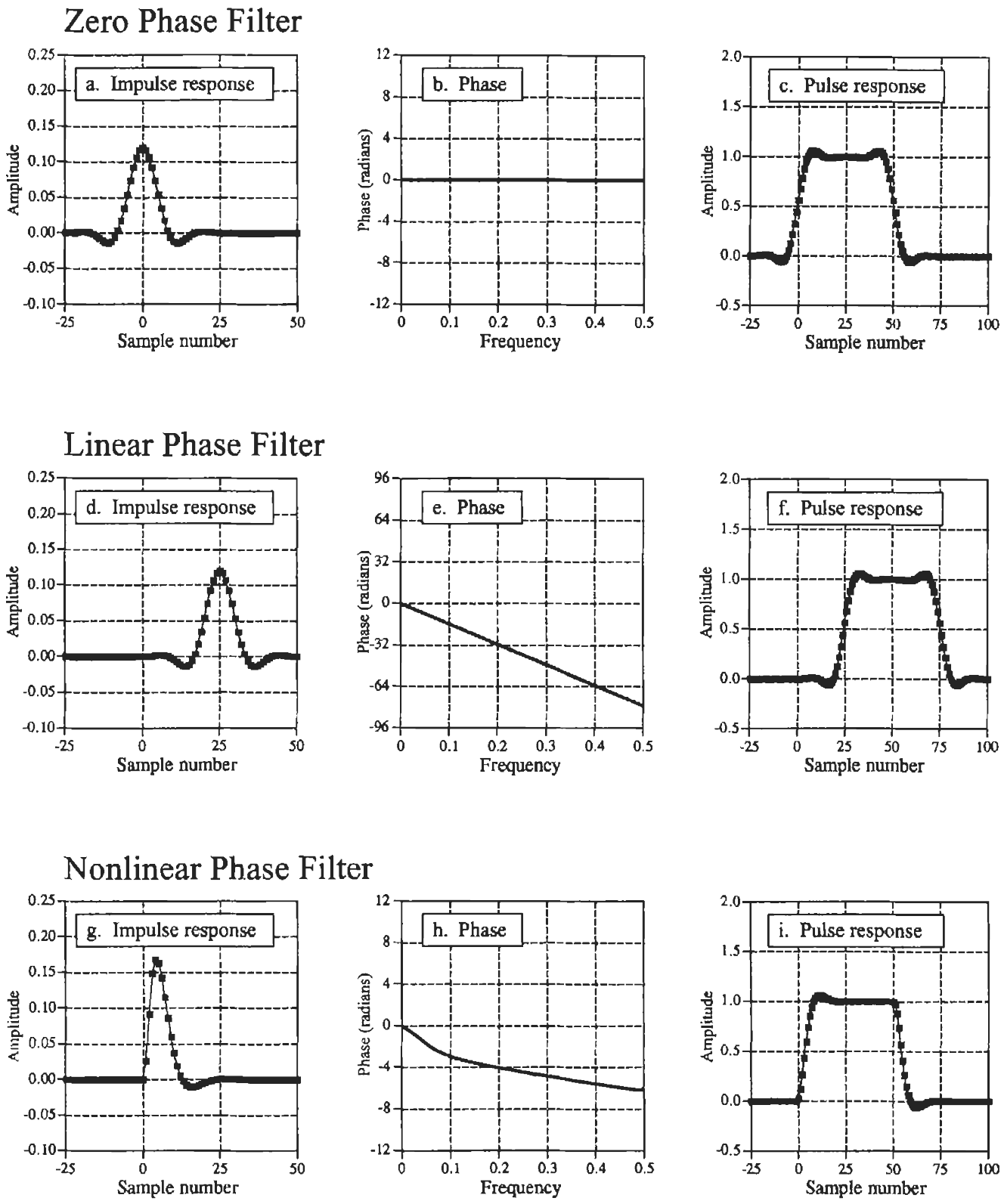

FIGURE 19-7
Zero, linear, and nonlinear phase filters. A *zero phase* filter has an impulse response that has left-right symmetry around sample number zero, as in (a). This results in a frequency response that has a phase composed entirely of zeros, as in (b). Zero phase impulse responses are desirable because their step responses are symmetrical between the top and bottom, making the left and right edges of pulses look the same, as is shown in (c). *Linear phase* filters have left-right symmetry, but not around sample zero, as illustrated in (d). This results in a phase that is linear, that is, a straight line, as shown in (e). The linear phase pulse response, shown in (f), has all the advantages of the zero phase pulse response. In comparison, the impulse responses of *nonlinear phase* filters are not symmetrical between the left and right, as in (g), and the phases are not a straight line, as in (h). The worst part is that the left and right edges of the pulse response are not the same, as shown in (i).

Analog electronic circuits have this same problem with the phase response. Imagine a circuit composed of resistors and capacitors sitting on your desk. If the input has always been zero, the output will also have always been zero. When an impulse is applied to the input, the capacitors quickly charge to some value and then begin to exponentially decay through the resistors. The impulse response (i.e., the output signal) is a combination of these various decaying exponentials. The impulse response *cannot* be symmetrical, because the output was zero before the impulse, and the exponential decay never quite reaches a value of zero again. Analog filter designers attack this problem with the **Bessel filter**, presented in Chapter 3. The Bessel filter is designed to have as linear phase as possible; however, it is far below the performance of digital filters. The ability to provide an *exact* linear phase is a clear advantage of digital filters.

Fortunately, there is a simple way to modify recursive filters to obtain a *zero phase*. Figure 19-8 shows an example of how this works. The input signal to be filtered is shown in (a). Figure (b) shows the signal after it has been filtered by a single pole low-pass filter. Since this is a nonlinear phase filter, the left and right edges do not look the same; they are inverted versions of each other. As previously described, this recursive filter is implemented by starting at sample 0 and working toward sample 150, calculating each sample along the way.

Now, suppose that instead of moving from sample 0 toward sample 150, we start at sample 150 and move toward sample 0. In other words, each sample in the output signal is calculated from input and output samples to the *right* of the sample being worked on. This means that the recursion equation, Eq. 19-1, is changed to:

$$y[n] = a_0 x[n] + a_1 x[n+1] + a_2 x[n+2] + a_3 x[n+3] + \cdots + b_1 y[n+1] + b_2 y[n+2] + b_3 y[n+3] + \cdots$$

EQUATION 19-9
The *reverse* recursion equation. This is the same as Eq. 19-1, except the signal is filtered from left-to-right, instead of right-to-left.

Figure (c) shows the result of this **reverse filtering**. This is analogous to passing an analog signal through an electronic RC circuit while running time *backwards*. !esrevinu eht pu-wercs nac lasrever emit -noituaC

Filtering in the reverse direction does not produce any benefit in itself; the filtered signal still has left and right edges that do not look alike. The magic happens when forward and reverse filtering are *combined*. Figure (d) results from filtering the signal in the forward direction and then filtering again in the reverse direction. Voila! This produces a *zero phase* recursive filter. In fact, *any* recursive filter can be converted to zero phase with this **bidirectional filtering** technique. The only penalty for this improved performance is a factor of two in execution time and program complexity.

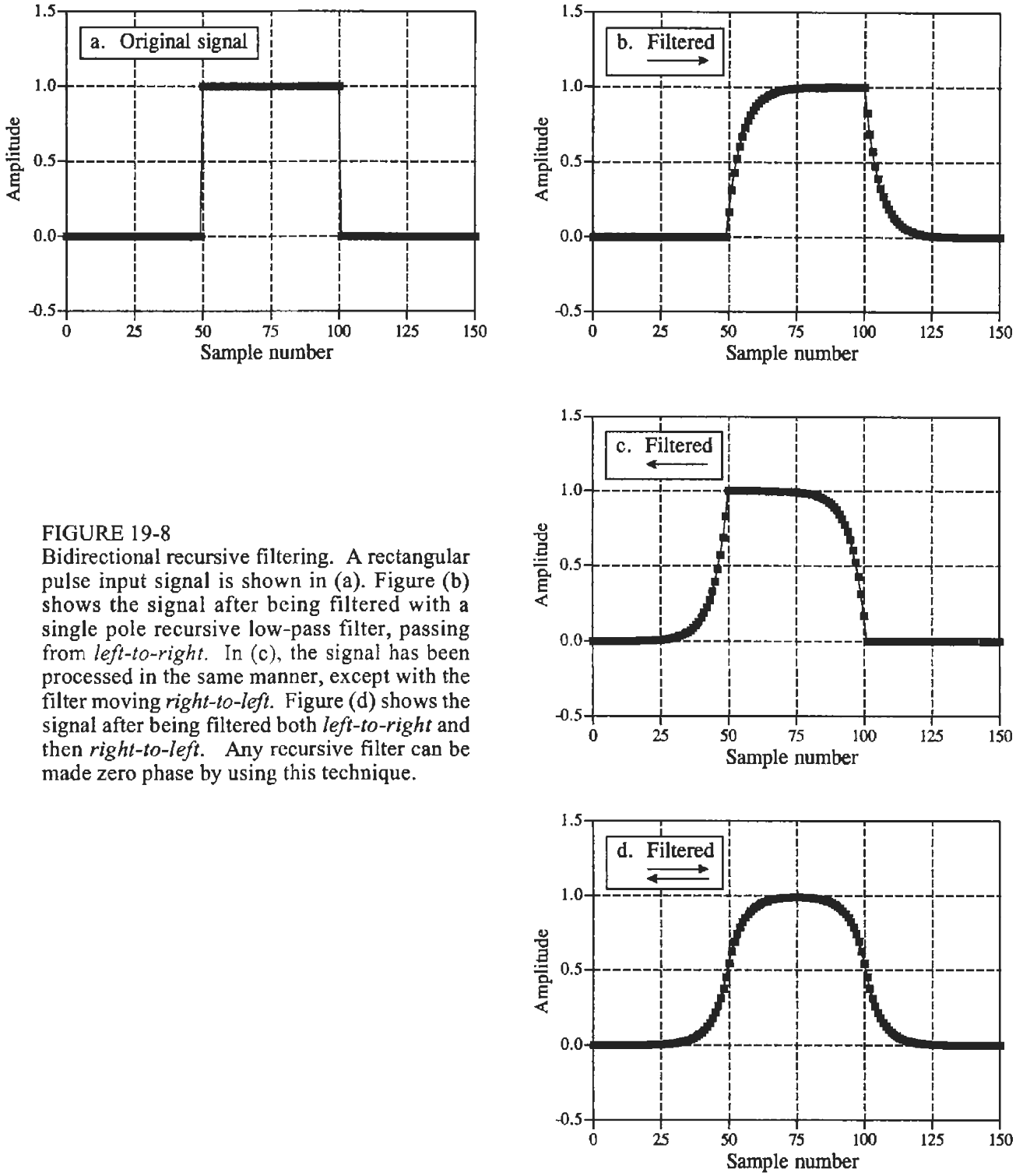

FIGURE 19-8
Bidirectional recursive filtering. A rectangular pulse input signal is shown in (a). Figure (b) shows the signal after being filtered with a single pole recursive low-pass filter, passing from *left-to-right*. In (c), the signal has been processed in the same manner, except with the filter moving *right-to-left*. Figure (d) shows the signal after being filtered both *left-to-right* and then *right-to-left*. Any recursive filter can be made zero phase by using this technique.

How do you find the impulse and frequency responses of the overall filter? The magnitude of the frequency response is the same for each direction, while the phases are opposite in sign. When the two directions are combined, the magnitude becomes *squared*, while the phase cancels to *zero*. In the time domain, this corresponds to convolving the original impulse response with a left-for-right flipped version of itself. For instance, the impulse response of a

single pole low-pass filter is a one-sided exponential. The impulse response of the corresponding bidirectional filter is a one-sided exponential that decays to the right, convolved with a one-sided exponential that decays to the left. Going through the mathematics, this turns out to be a double-sided exponential that decays both to the left and right, with the same decay constant as the original filter.

Some applications only have a portion of the signal in the computer at a particular time, such as systems that alternately input and output data on a continuing basis. Bidirectional filtering can be used in these cases by combining it with the overlap-add method described in the last chapter. When you come to the question of how long the impulse response is, don't say "infinite." If you do, you will need to pad each signal segment with an *infinite* number of zeros. Remember, the impulse response can be truncated when it has decayed below the round-off noise level, i.e., about 15 to 20 time constants. Each segment will need to be padded with zeros on both the left and right to allow for the expansion during the bidirectional filtering.

Using Integers

Single-precision floating point is ideal to implement these simple recursive filters. The use of integers is possible, but it is much more difficult. There are two main problems. First, the round-off error from the limited number of bits can degrade the response of the filter, or even make it unstable. Second, the fractional values of the recursion coefficients must be handled with integer math. One way to attack this problem is to express each coefficient as a fraction. For example, 0.15 becomes 19/128. Instead of multiplying by 0.15, you first multiply by 19 and then divide by 128. Another way is to replace the multiplications with look-up tables. For example, a 12-bit ADC produces samples with a value between 0 and 4095. Instead of multiplying each sample by 0.15, you pass the samples through a look-up table that is 4096 entries long. The value obtained from the look-up table is equal to 0.15 times the value entering the look-up table. This method is very fast, but it does require extra memory; a separate look-up table is needed for each coefficient. Before you try either of these integer methods, make sure the recursive algorithm for the moving average filter will not suit your needs. It *loves* integers.

CHAPTER

20

Chebyshev Filters

Chebyshev filters are used to separate one band of frequencies from another. Although they cannot match the performance of the windowed-sinc filter, they are more than adequate for many applications. The primary attribute of Chebyshev filters is their speed, typically more than an order of magnitude faster than the windowed-sinc. This is because they are carried out by *recursion* rather than *convolution*. The design of these filters is based on a mathematical technique called the *z-transform*, discussed in Chapter 33. This chapter presents the information needed to *use* Chebyshev filters without wading through a mire of advanced mathematics.

The Chebyshev and Butterworth Responses

The Chebyshev response is a mathematical strategy for achieving a faster *roll-off* by allowing *ripple* in the frequency response. Analog and digital filters that use this approach are called *Chebyshev filters*. For instance, analog Chebyshev filters were used in Chapter 3 for analog-to-digital and digital-to-analog conversion. These filters are named from their use of the *Chebyshev polynomials*, developed by the Russian mathematician Pafnuti Chebyshev (1821-1894). This name has been translated from Russian and appears in the literature with different spellings, such as: Chebychev, Tschebyscheff, Tchebysheff and Tchebichef.

Figure 20-1 shows the frequency response of low-pass Chebyshev filters with passband ripples of: 0%, 0.5% and 20%. As the ripple increases (bad), the roll-off becomes sharper (good). The Chebyshev response is an optimal trade-off between these two parameters. When the ripple is set to 0%, the filter is called a **maximally flat** or **Butterworth filter** (after S. Butterworth, a British engineer who described this response in 1930). A ripple of 0.5% is often a good choice for digital filters. This matches the typical precision and accuracy of the analog electronics that the signal has passed through.

The Chebyshev filters discussed in this chapter are called **type 1** filters, meaning that the ripple is only allowed in the *passband*. In comparison,

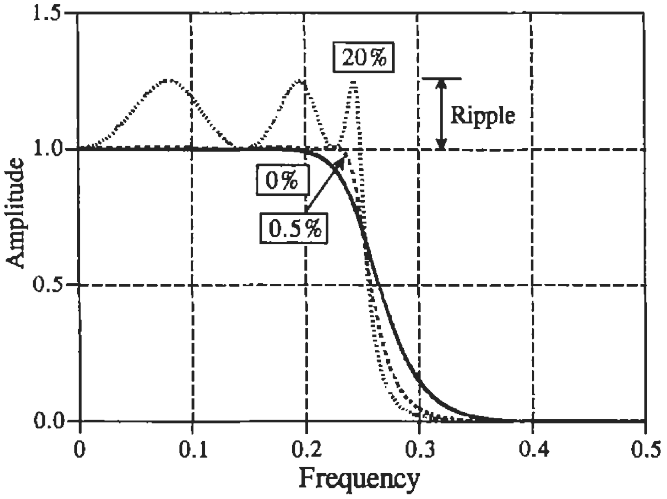

FIGURE 20-1
The Chebyshev response. Chebyshev filters achieve a faster roll-off by allowing ripple in the passband. When the ripple is set to 0%, it is called a *maximally flat* or *Butterworth* filter. Consider using a ripple of 0.5% in your designs; this passband unflatness is so small that it cannot be seen in this graph, but the roll-off is much faster than the Butterworth.

type 2 Chebyshev filters have ripple only in the *stopband*. Type 2 filters are seldom used, and we won't discuss them. There is, however, an important design called the **elliptic filter**, which has ripple in *both* the passband and the stopband. Elliptic filters provide the fastest roll-off for a given number of poles, but are much harder to design. We won't discuss the elliptic filter here, but be aware that it is frequently the first choice of professional filter designers, both in analog electronics and DSP. If you need this level of performance, buy a software package for designing digital filters.

Designing the Filter

You must select four parameters to design a Chebyshev filter: (1) a high-pass or low-pass response, (2) the cutoff frequency, (3) the percent ripple in the passband, and (4) the number of poles. Just what is a *pole*? Here are two answers. If you don't like one, maybe the other will help:

Answer 1- The Laplace transform and z-transform are mathematical ways of breaking an impulse response into sinusoids and decaying exponentials. This is done by expressing the system's characteristics as one complex polynomial divided by another complex polynomial. The roots of the numerator are called *zeros*, while the roots of the denominator are called *poles*. Since poles and zeros can be complex numbers, it is common to say they have a "location" in the complex plane. Elaborate systems have more poles and zeros than simple ones. Recursive filters are designed by first selecting the location of the poles and zeros, and then finding the appropriate recursion coefficients (or analog components). For example, Butterworth filters have poles that lie on a *circle* in the complex plane, while in a Chebyshev filter they lie on an *ellipse*. This is the topic of Chapters 32 and 33.

Answer 2- Poles are containers filled with magic powder. The more poles in a filter, the better the filter works.

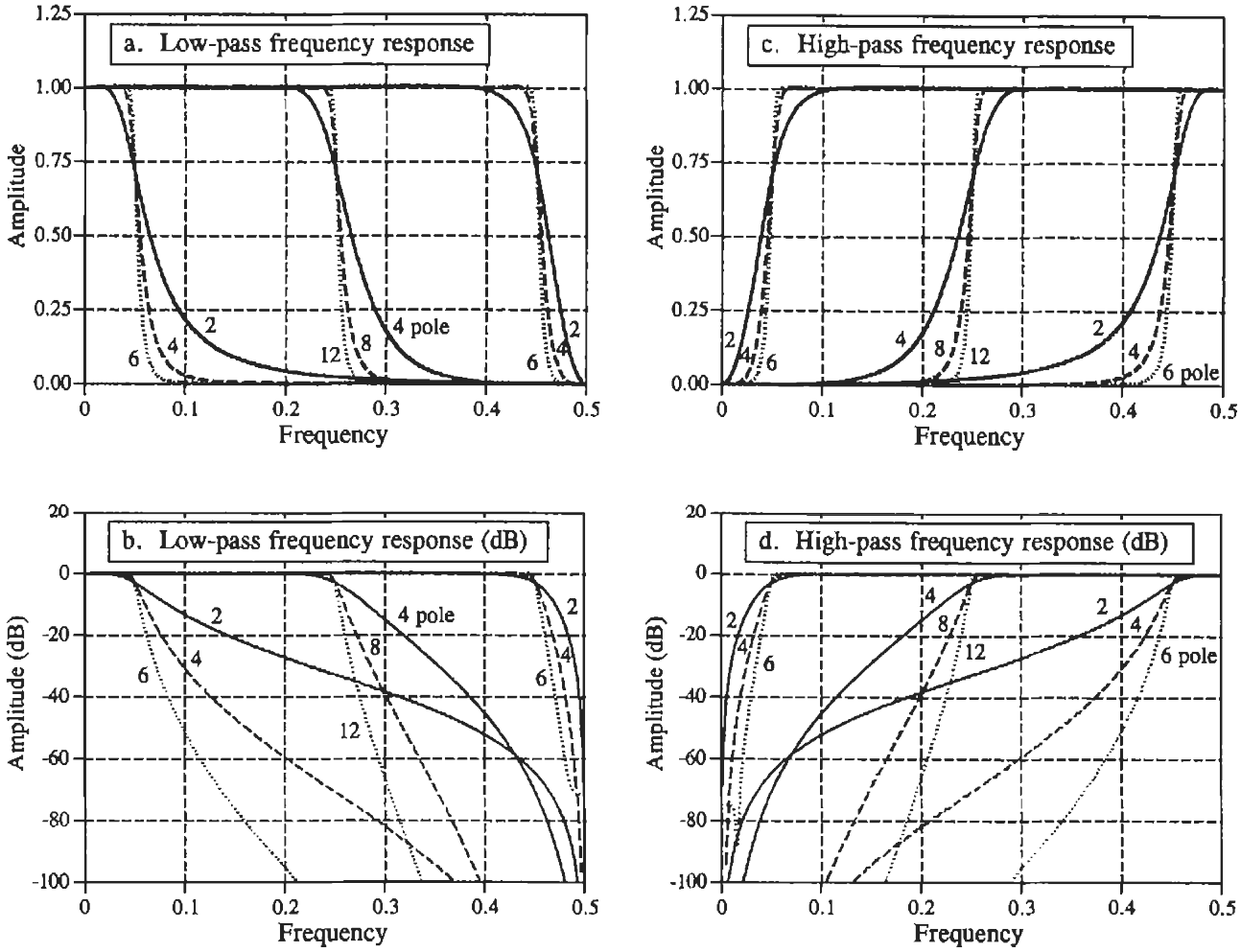

FIGURE 20-2
Chebyshev frequency responses. Figures (a) and (b) show the frequency responses of low-pass Chebyshev filters with 0.5% ripple, while (c) and (d) show the corresponding high-pass filter responses.

Kidding aside, the point is that you can use these filters very effectively without knowing the nasty mathematics behind them. Filter design is a specialty. In actual practice, more engineers, scientists and programmers think in terms of answer 2, than answer 1.

Figure 20-2 shows the frequency response of several Chebyshev filters with 0.5% ripple. For the method used here, the number of poles must be *even*. The cutoff frequency of each filter is measured where the amplitude crosses 0.707 (-3dB). Filters with a cutoff frequency near 0 or 0.5 have a sharper roll-off than filters in the center of the frequency range. For example, a two pole filter at f_C = 0.05 has about the same roll-off as a four pole filter at f_C = 0.25. This is fortunate; fewer poles can be used near 0 and 0.5 because of round-off noise. More about this later.

There are two ways of finding the recursion coefficients without using the z-transform. First, the coward's way: use a table. Tables 20-1 and 20-2 provide the recursion coefficients for low-pass and high-pass filters with 0.5% passband ripple. If you only need a quick and dirty design, copy the appropriate coefficients into your program, and you're done.

f_C	2 Pole	4 Pole	6 Pole
0.01	a0= 8.663387E-04 a1= 1.732678E-03 b1= 1.919129E+00 a2= 8.663387E-04 b2= -9.225943E-01	a0= 4.149425E-07 (!! Unstable !!) a1= 1.659770E-06 b1= 3.893453E+00 a2= 2.489655E-06 b2= -5.688233E+00 a3= 1.659770E-06 b3= 3.695783E+00 a4= 4.149425E-07 b4= -9.010106E-01	a0= 1.391351E-10 (!! Unstable !!) a1= 8.348109E-10 b1= 5.883343E+00 a2= 2.087027E-09 b2= -1.442798E+01 a3= 2.782703E-09 b3= 1.887786E+01 a4= 2.087027E-09 b4= -1.389914E+01 a5= 8.348109E-10 b5= 5.459909E+00 a6= 1.391351E-10 b6= -8.939932E-01
0.025	a0= 5.112374E-03 a1= 1.022475E-02 b1= 1.797154E+00 a2= 5.112374E-03 b2= -8.176033E-01	a0= 1.504626E-05 a1= 6.018503E-05 b1= 3.725385E+00 a2= 9.027754E-05 b2= -5.226004E+00 a3= 6.018503E-05 b3= 3.270902E+00 a4= 1.504626E-05 b4= -7.705239E-01	a0= 3.136210E-08 (!! Unstable !!) a1= 1.881726E-07 b1= 5.691653E+00 a2= 4.704314E-07 b2= -1.353172E+01 a3= 6.272419E-07 b3= 1.719986E+01 a4= 4.704314E-07 b4= -1.232689E+01 a5= 1.881726E-07 b5= 4.722721E+00 a6= 3.136210E-08 b6= -7.556340E-01
0.05	a0= 1.868823E-02 a1= 3.737647E-02 b1= 1.593937E+00 a2= 1.868823E-02 b2= -6.686903E-01	a0= 2.141509E-04 a1= 8.566037E-04 b1= 3.425455E+00 a2= 1.284906E-03 b2= -4.479272E+00 a3= 8.566037E-04 b3= 2.643718E+00 a4= 2.141509E-04 b4= -5.933269E-01	a0= 1.771089E-06 a1= 1.062654E-05 b1= 5.330512E+00 a2= 2.656634E-05 b2= -1.196611E+01 a3= 3.542179E-05 b3= 1.447067E+01 a4= 2.656634E-05 b4= -9.937710E+00 a5= 1.062654E-05 b5= 3.673283E+00 a6= 1.771089E-06 b6= -5.707561E-01
0.075	a0= 3.869430E-02 a1= 7.738860E-02 b1= 1.392667E+00 a2= 3.869430E-02 b2= -5.474446E-01	a0= 9.726342E-04 a1= 3.890537E-03 b1= 3.103944E+00 a2= 5.835806E-03 b2= -3.774453E+00 a3= 3.890537E-03 b3= 2.111238E+00 a4= 9.726342E-04 b4= -4.562908E-01	a0= 1.797538E-05 a1= 1.078523E-04 b1= 4.921746E+00 a2= 2.696307E-04 b2= -1.035734E+01 a3= 3.595076E-04 b3= 1.189764E+01 a4= 2.696307E-04 b4= -7.854533E+00 a5= 1.078523E-04 b5= 2.822109E+00 a6= 1.797538E-05 b6= -4.307710E-01
0.1	a0= 6.372802E-02 a1= 1.274560E-01 b1= 1.194365E+00 a2= 6.372802E-02 b2= -4.492774E-01	a0= 2.780755E-03 a1= 1.112302E-02 b1= 2.764031E+00 a2= 1.668453E-02 b2= -3.122854E+00 a3= 1.112302E-02 b3= 1.664554E+00 a4= 2.780755E-03 b4= -3.502232E-01	a0= 9.086148E-05 a1= 5.451688E-04 b1= 4.470118E+00 a2= 1.362922E-03 b2= -8.755594E+00 a3= 1.817229E-03 b3= 9.543712E+00 a4= 1.362922E-03 b4= -6.079376E+00 a5= 5.451688E-04 b5= 2.140062E+00 a6= 9.086148E-05 b6= -3.247363E-01
0.15	a0= 1.254285E-01 a1= 2.508570E-01 b1= 8.070778E-01 a2= 1.254285E-01 b2= -3.087918E-01	a0= 1.180009E-02 a1= 4.720034E-02 b1= 2.039039E+00 a2= 7.080051E-02 b2= -2.012961E+00 a3= 4.720034E-02 b3= 9.897915E-01 a4= 1.180009E-02 b4= -2.046700E-01	a0= 8.618665E-04 a1= 5.171199E-03 b1= 3.455239E+00 a2= 1.292800E-02 b2= -5.754735E+00 a3= 1.723733E-02 b3= 5.645387E+00 a4= 1.292800E-02 b4= -3.394902E+00 a5= 5.171199E-03 b5= 1.177469E+00 a6= 8.618665E-04 b6= -1.836195E-01
0.2	a0= 1.997396E-01 a1= 3.994792E-01 b1= 4.291048E-01 a2= 1.997396E-01 b2= -2.280633E-01	a0= 3.224554E-02 a1= 1.289821E-01 b1= 1.265912E+00 a2= 1.934732E-01 b2= -1.203878E+00 a3= 1.289821E-01 b3= 5.405908E-01 a4= 3.224554E-02 b4= -1.185536E-01	a0= 4.187408E-03 a1= 2.512445E-02 b1= 2.315806E+00 a2= 6.281112E-02 b2= -3.293726E+00 a3= 8.374816E-02 b3= 2.904826E+00 a4= 6.281112E-02 b4= -1.694128E+00 a5= 2.512445E-02 b5= 6.021426E-01 a6= 4.187408E-03 b6= -1.029147E-01
0.25	a0= 2.858110E-01 a1= 5.716221E-01 b1= 5.423258E-02 a2= 2.858110E-01 b2= -1.974768E-01	a0= 7.015301E-02 a1= 2.806120E-01 b1= 4.541481E-01 a2= 4.209180E-01 b2= -7.417536E-01 a3= 2.806120E-01 b3= 2.361222E-01 a4= 7.015301E-02 b4= -7.096476E-02	a0= 1.434449E-02 a1= 8.606697E-02 b1= 1.076052E+00 a2= 2.151674E-01 b2= -1.662847E+00 a3= 2.868899E-01 b3= 1.191063E+00 a4= 2.151674E-01 b4= -7.403087E-01 a5= 8.606697E-02 b5= 2.752158E-01 a6= 1.434449E-02 b6= -5.722251E-02
0.3	a0= 3.849163E-01 a1= 7.698326E-01 b1= -3.249116E-01 a2= 3.849163E-01 b2= -2.147536E-01	a0= 1.335566E-01 a1= 5.342263E-01 b1= -3.904486E-01 a2= 8.013394E-01 b2= -6.784138E-01 a3= 5.342263E-01 b3= -1.412021E-02 a4= 1.335566E-01 b4= -5.392238E-02	a0= 3.997487E-02 a1= 2.398492E-01 b1= -2.441152E-01 a2= 5.996231E-01 b2= -1.130306E+00 a3= 7.994975E-01 b3= 1.063167E-01 a4= 5.996231E-01 b4= -3.463299E-01 a5= 2.398492E-01 b5= 8.882992E-02 a6= 3.997487E-02 b6= -3.278741E-02
0.35	a0= 5.001024E-01 a1= 1.000205E+00 b1= -7.158993E-01 a2= 5.001024E-01 b2= -2.845103E-01	a0= 2.340973E-01 a1= 9.363892E-01 b1= -1.263672E+00 a2= 1.404584E+00 b2= -1.080487E+00 a3= 9.363892E-01 b3= -3.276296E-01 a4= 2.340973E-01 b4= -7.376791E-02	a0= 9.792321E-02 a1= 5.875393E-01 b1= -1.627573E+00 a2= 1.468848E+00 b2= -1.955020E+00 a3= 1.958464E+00 b3= -1.075051E+00 a4= 1.468848E+00 b4= -5.106501E-01 a5= 5.875393E-01 b5= -7.239843E-02 a6= 9.792321E-02 b6= -2.639193E-02
0.40	a0= 6.362308E-01 a1= 1.272462E+00 b1= -1.125379E+00 a2= 6.362308E-01 b2= -4.195441E-01	a0= 3.896966E-01 a1= 1.558787E+00 b1= -2.161179E+00 a2= 2.338186E+00 b2= -2.033992E+00 a3= 1.558787E+00 b3= -8.789098E-01 a4= 3.896966E-01 b4= -1.610655E-01	a0= 2.211834E-01 a1= 1.327100E+00 b1= -3.058672E+00 a2= 3.317751E+00 b2= -4.390465E+00 a3= 4.423668E+00 b3= -3.523254E+00 a4= 3.317751E+00 b4= -1.684185E+00 a5= 1.327100E+00 b5= -4.414881E-01 a6= 2.211834E-01 b6= -5.767513E-02
0.45	a0= 8.001101E-01 a1= 1.600220E+00 b1= -1.556269E+00 a2= 8.001101E-01 b2= -6.441713E-01	a0= 6.291693E-01 a1= 2.516677E+00 b1= -3.077062E+00 a2= 3.775016E+00 b2= -3.641323E+00 a3= 2.516677E+00 b3= -1.949229E+00 a4= 6.291693E-01 b4= -3.990945E-01	a0= 4.760635E-01 a1= 2.856381E+00 b1= -4.522403E+00 a2= 7.140952E+00 b2= -8.676844E+00 a3= 9.521270E+00 b3= -9.007512E+00 a4= 7.140952E+00 b4= -5.328429E+00 a5= 2.856381E+00 b5= -1.702543E+00 a6= 4.760635E-01 b6= -2.303303E-01

TABLE 20-1
Low-pass Chebyshev filters (0.5% ripple)

f_C	2 Pole	4 Pole	6 Pole
0.01	a0= 9.567529E-01 a1= -1.913506E+00 b1= 1.911437E+00 a2= 9.567529E-01 b2= -9.155749E-01	a0= 9.121579E-01 (!! Unstable !!) a1= -3.648632E+00 b1= 3.815952E+00 a2= 5.472947E+00 b2= -5.465026E+00 a3= -3.648632E+00 b3= 3.481295E+00 a4= 9.121579E-01 b4= -8.322529E-01	a0= 8.630195E-01 (!! Unstable !!) a1= -5.178118E+00 b1= 5.705102E+00 a2= 1.294529E+01 b2= -1.356935E+01 a3= -1.726039E+01 b3= 1.722231E+01 a4= 1.294529E+01 b4= -1.230214E+01 a5= -5.178118E+00 b5= 4.689218E+00 a6= 8.630195E-01 b6= -7.451429E-01
0.025	a0= 8.950355E-01 a1= -1.790071E+00 b1= 1.777932E+00 a2= 8.950355E-01 b2= -8.022106E-01	a0= 7.941874E-01 a1= -3.176750E+00 b1= 3.538919E+00 a2= 4.765125E+00 b2= -4.722213E+00 a3= -3.176750E+00 b3= 2.814036E+00 a4= 7.941874E-01 b4= -6.318300E-01	a0= 6.912863E-01 (!! Unstable !!) a1= -4.147718E+00 b1= 5.261399E+00 a2= 1.036929E+01 b2= -1.157800E+01 a3= -1.382573E+01 b3= 1.363599E+01 a4= 1.036929E+01 b4= -9.063840E+00 a5= -4.147718E+00 b5= 3.223738E+00 a6= 6.912863E-01 b6= -4.793541E-01
0.05	a0= 8.001102E-01 a1= -1.600220E+00 b1= 1.556269E+00 a2= 8.001102E-01 b2= -6.441715E-01	a0= 6.291694E-01 a1= -2.516678E+00 b1= 3.077062E+00 a2= 3.775016E+00 b2= -3.641324E+00 a3= -2.516678E+00 b3= 1.949230E+00 a4= 6.291694E-01 b4= -3.990947E-01	a0= 4.760636E-01 a1= -2.856382E+00 b1= 4.522403E+00 a2= 7.140954E+00 b2= -8.676846E+00 a3= -9.521272E+00 b3= 9.007515E+00 a4= 7.140954E+00 b4= -5.328431E+00 a5= -2.856382E+00 b5= 1.702544E+00 a6= 4.760636E-01 b6= -2.303304E-01
0.075	a0= 7.142028E-01 a1= -1.428406E+00 b1= 1.338264E+00 a2= 7.142028E-01 b2= -5.185469E-01	a0= 4.965350E-01 a1= -1.986140E+00 b1= 2.617304E+00 a2= 2.979210E+00 b2= -2.749252E+00 a3= -1.986140E+00 b3= 1.325548E+00 a4= 4.965350E-01 b4= -2.524546E-01	a0= 3.259100E-01 a1= -1.955460E+00 b1= 3.787397E+00 a2= 4.888651E+00 b2= -6.288362E+00 a3= -6.518201E+00 b3= 5.747801E+00 a4= 4.888651E+00 b4= -3.041570E+00 a5= -1.955460E+00 b5= 8.808669E-01 a6= 3.259100E-01 b6= -1.122464E-01
0.1	a0= 6.362307E-01 a1= -1.272461E+00 b1= 1.125379E+00 a2= 6.362307E-01 b2= -4.195440E-01	a0= 3.896966E-01 a1= -1.558786E+00 b1= 2.161179E+00 a2= 2.338179E+00 b2= -2.033991E+00 a3= -1.558786E+00 b3= 8.789094E-01 a4= 3.896966E-01 b4= -1.610655E-01	a0= 2.211833E-01 a1= -1.327100E+00 b1= 3.058671E+00 a2= 3.317750E+00 b2= -4.390464E+00 a3= -4.423667E+00 b3= 3.523252E+00 a4= 3.317750E+00 b4= -1.684184E+00 a5= -1.327100E+00 b5= 4.414878E-01 a6= 2.211833E-01 b6= -5.767508E-02
0.15	a0= 5.001024E-01 a1= -1.000205E+00 b1= 7.158993E-01 a2= 5.001024E-01 b2= -2.845103E-01	a0= 2.340973E-01 a1= -9.363892E-01 b1= 1.263672E+00 a2= 1.404584E+00 b2= -1.080487E+00 a3= -9.363892E-01 b3= 3.276296E-01 a4= 2.340973E-01 b4= -7.376791E-02	a0= 9.792321E-02 a1= -5.875393E-01 b1= 1.627573E+00 a2= 1.468848E+00 b2= -1.955020E+00 a3= -1.958464E+00 b3= 1.075051E+00 a4= 1.468848E+00 b4= -5.106501E-01 a5= -5.875393E-01 b5= 7.239843E-02 a6= 9.792321E-02 b6= -2.639193E-02
0.2	a0= 3.849163E-01 a1= -7.698326E-01 b1= 3.249116E-01 a2= 3.849163E-01 b2= -2.147536E-01	a0= 1.335566E-01 a1= -5.342262E-01 b1= 3.904484E-01 a2= 8.013393E-01 b2= -6.784138E-01 a3= -5.342262E-01 b3= 1.412016E-02 a4= 1.335566E-01 b4= -5.392238E-02	a0= 3.997486E-02 a1= -2.398492E-01 b1= 2.441149E-01 a2= 5.996230E-01 b2= -1.130306E+00 a3= -7.994973E-01 b3= -1.063169E-01 a4= 5.996230E-01 b4= -3.463299E-01 a5= -2.398492E-01 b5= -8.882996E-02 a6= 3.997486E-02 b6= -3.278741E-02
0.25	a0= 2.858111E-01 a1= -5.716222E-01 b1= -5.423243E-02 a2= 2.858111E-01 b2= -1.974768E-01	a0= 7.015302E-02 a1= -2.806121E-01 b1= -4.541478E-01 a2= 4.209182E-01 b2= -7.417535E-01 a3= -2.806121E-01 b3= -2.361221E-01 a4= 7.015302E-02 b4= -7.096475E-02	a0= 1.434450E-02 a1= -8.606701E-02 b1= -1.076051E+00 a2= 2.151675E-01 b2= -1.662847E+00 a3= -2.868900E-01 b3= -1.191062E+00 a4= 2.151675E-01 b4= -7.403085E-01 a5= -8.606701E-02 b5= -2.752156E-01 a6= 1.434450E-02 b6= -5.722250E-02
0.3	a0= 1.997396E-01 a1= -3.994792E-01 b1= -4.291049E-01 a2= 1.997396E-01 b2= -2.280633E-01	a0= 3.224553E-02 a1= -1.289821E-01 b1= -1.265912E+00 a2= 1.934732E-01 b2= -1.203878E+00 a3= -1.289821E-01 b3= -5.405908E-01 a4= 3.224553E-02 b4= -1.185538E-01	a0= 4.187407E-03 a1= -2.512444E-02 b1= -2.315806E+00 a2= 6.281111E-02 b2= -3.293726E+00 a3= -8.374815E-02 b3= -2.904827E+00 a4= 6.281111E-02 b4= -1.694129E+00 a5= -2.512444E-02 b5= -6.021426E-01 a6= 4.187407E-03 b6= -1.029147E-01
0.35	a0= 1.254285E-01 a1= -2.508570E-01 b1= -8.070777E-01 a2= 1.254285E-01 b2= -3.087918E-01	a0= 1.180009E-02 a1= -4.720035E-02 b1= -2.039039E+00 a2= 7.080051E-02 b2= -2.012961E+00 a3= -4.720035E-02 b3= -9.897915E-01 a4= 1.180009E-02 b4= -2.046700E-01	a0= 8.618665E-04 a1= -5.171200E-03 b1= -3.455239E+00 a2= 1.292800E-02 b2= -5.754734E+00 a3= -1.723733E-02 b3= -5.645387E+00 a4= 1.292800E-02 b4= -3.394902E+00 a5= -5.171200E-03 b5= -1.177469E+00 a6= 8.618665E-04 b6= -1.836195E-01
0.40	a0= 6.372801E-02 a1= -1.274560E-01 b1= -1.194365E+00 a2= 6.372801E-02 b2= -4.492774E-01	a0= 2.780754E-03 a1= -1.112302E-02 b1= -2.764031E+00 a2= 1.668453E-02 b2= -3.122854E+00 a3= -1.112302E-02 b3= -1.664554E+00 a4= 2.780754E-03 b4= -3.502233E-01	a0= 9.066141E-05 a1= -5.451685E-04 b1= -4.470118E+00 a2= 1.362921E-03 b2= -8.755595E+00 a3= -1.817228E-03 b3= -9.543712E+00 a4= 1.362921E-03 b4= -6.079377E+00 a5= -5.451685E-04 b5= -2.140062E+00 a6= 9.066141E-05 b6= -3.247363E-01
0.45	a0= 1.868823E-02 a1= -3.737647E-02 b1= -1.593937E+00 a2= 1.868823E-02 b2= -6.686903E-01	a0= 2.141509E-04 a1= -8.566037E-04 b1= -3.425455E+00 a2= 1.284906E-03 b2= -4.479272E+00 a3= -8.566037E-04 b3= -2.643718E+00 a4= 2.141509E-04 b4= -5.933269E-01	a0= 1.771089E-06 a1= -1.062654E-05 b1= -5.330612E+00 a2= 2.656634E-05 b2= -1.196611E+01 a3= -3.542179E-05 b3= -1.447067E+01 a4= 2.656634E-05 b4= -9.937710E+00 a5= -1.062654E-05 b5= -3.673283E+00 a6= 1.771089E-06 b6= -5.707561E-01

TABLE 20-2
High-pass Chebyshev filters (0.5% ripple)

There are two problems with using tables to design digital filters. First, tables have a limited choice of parameters. For instance, Table 20-1 only provides 12 different cutoff frequencies, a maximum of 6 poles per filter, and *no* choice of passband ripple. Without the ability to select parameters from a continuous range of values, the filter design cannot be *optimized*. Second, the coefficients must be manually transferred from the table into the program. This is very time consuming and will discourage you from trying alternative values.

Instead of using tabulated values, consider including a subroutine in your program that *calculates* the coefficients. Such a program is shown in Table 20-4. The good news is that the program is relatively simple in structure. After the four filter parameters are entered, the program spits out the "a" and "b" coefficients in the arrays A[ ] and B[ ]. The bad news is that the program calls the subroutine in Table 20-5. At first glance this subroutine is really ugly. Don't despair; it isn't as bad as it seems! There is one simple branch in line 1120. Everything else in the subroutine is straightforward number crunching. Six variables enter the routine, five variables leave the routine, and fifteen temporary variables (plus indexes) are used within. Table 20-5 provides two sets of test data for debugging this subroutine. Chapter 31 discusses the operation of this program in detail.

Step Response Overshoot

Butterworth and Chebyshev filters have an overshoot of 5 to 30% in their step responses, becoming larger as the number of poles is increased. Figure 20-3a shows the step response for two example Chebyshev filters. Figure (b) shows something that is unique to digital filters and has no counterpart in analog electronics: the amount of overshoot in the step response depends to a small degree on the cutoff frequency of the filter. The excessive overshoot and ringing in the step response results from the Chebyshev filter being optimized for the *frequency domain* at the expense of the *time domain*.

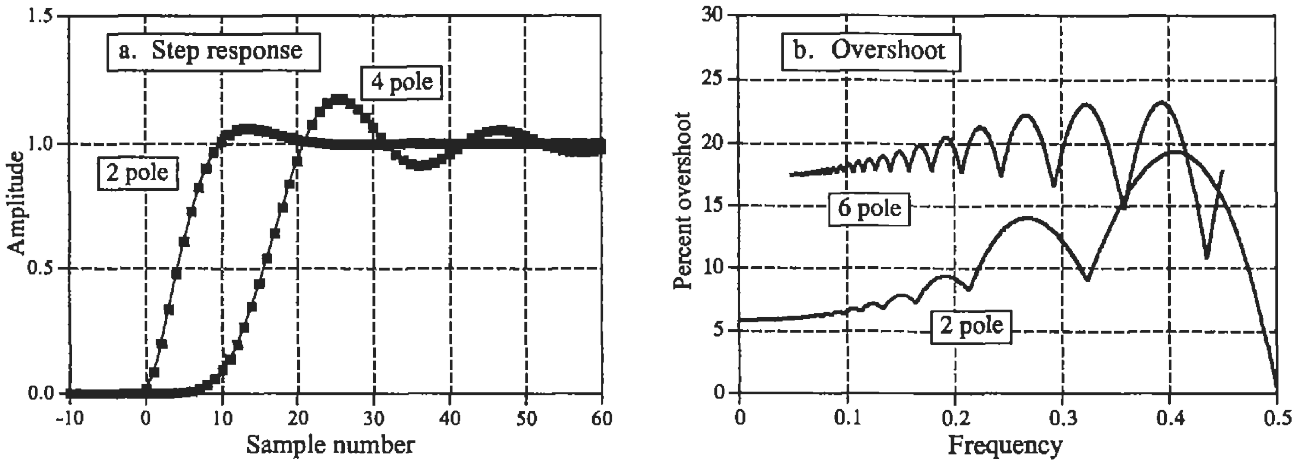

FIGURE 20-3
Chebyshev step response. The overshoot in the Chebyshev filter's step response is 5% to 30%, depending on the number of poles, as shown in (a), and the cutoff frequency, as shown in (b). Figure (a) is for a cutoff frequency of 0.05, and may be scaled to other cutoff frequencies.

Stability

The main limitation of digital filters carried out by convolution is *execution time*. It is possible to achieve nearly any filter response, provided you are willing to wait for the result. Recursive filters are just the opposite. They run like lightning; however, they are limited in performance. For example, consider a 6 pole, 0.5% ripple, low-pass filter with a 0.01 cutoff frequency. The recursion coefficients for this filter can be obtained from Table 20-1:

```
a0=  1.391351E-10
a1=  8.348109E-10  b1=  5.883343E+00
a2=  2.087027E-09  b2= -1.442798E+01
a3=  2.782703E-09  b3=  1.887786E+01
a4=  2.087027E-09  b4= -1.389914E+01
a5=  8.348109E-10  b5=  5.459909E+00
a6=  1.391351E-10  b6= -8.939932E-01
```

Look carefully at these coefficients. The "b" coefficients have an absolute value of about *ten*. Using single precision, the round-off noise on each of these numbers is about one ten-millionth of the value, i.e., 10^{-6}. Now look at the "a" coefficients, with a value of about 10^{-9}. Something is obviously wrong here. The contribution from the input signal (via the "a" coefficients) will be 1000 times smaller than the *noise* from the previously calculated output signal (via the "b" coefficients). This filter won't work! In short, round-off noise limits the number of poles that can be used in a filter. The actual number will depend slightly on the ripple and if it is a high or low-pass filter. The approximate numbers for single precision are:

TABLE 20-3
The maximum number of poles for single precision.

Cutoff frequency	0.02	0.05	0.10	0.25	0.40	0.45	0.48
Maximum poles	4	6	10	20	10	6	4

The filter's performance will start to degrade as this limit is approached; the step response will show more overshoot, the stopband attenuation will be poor, and the frequency response will have excessive ripple. If the filter is pushed too far, or there is an error in the coefficients, the output will probably oscillate until an overflow occurs.

There are two ways of extending the maximum number of poles that can be used. First, use double precision. This requires using double precision in the coefficient calculation as well (including the value for *pi*).

The second method is to implement the filter in *stages*. For example, a six pole filter starts out as a cascade of three stages of two poles each. The program in Table 20-4 combines these three stages into a single set of recursion coefficients for easier programming. However, the filter is more stable if carried out as the original three separate stages. This requires knowing the "a" and "b" coefficients for each of the stages. These can

```
100 'CHEBYSHEV FILTER-  RECURSION COEFFICIENT CALCULATION
110 '
120                                          'INITIALIZE VARIABLES
130 DIM A[22]                                'holds the "a" coefficients upon program completion
140 DIM B[22]                                'holds the "b" coefficients upon program completion
150 DIM TA[22]                               'internal use for combining stages
160 DIM TB[22]                               'internal use for combining stages
170 '
180 FOR I% = 0 TO 22
190   A[I%] = 0
200   B[I%] = 0
210 NEXT I%
220 '
230 A[2] = 1
240 B[2] = 1
250 PI = 3.14159265
260                                          'ENTER THE FOUR FILTER PARAMETERS
270 INPUT "Enter cutoff frequency  (0 to .5): ", FC
280 INPUT "Enter  0 for LP,  1 for HP filter:  ", LH
290 INPUT "Enter percent ripple    (0 to 29):  ", PR
300 INPUT "Enter number of poles (2,4,...20):          ", NP
310 '
320 FOR P% = 1 TO NP/2                       'LOOP FOR EACH POLE-PAIR
330   '
340   GOSUB 1000                             'The subroutine in TABLE 20-5
350   '
360   FOR I% = 0 TO 22                       'Add coefficients to the cascade
370     TA[I%] = A[I%]
380     TB[I%] = B[I%]
390   NEXT I%
400   '
410   FOR I% = 2 TO 22
420     A[I%] = A0*TA[I%] + A1*TA[I%-1] + A2*TA[I%-2]
430     B[I%] =     TB[I%] - B1*TB[I%-1] - B2*TB[I%-2]
440   NEXT I%
450   '
460 NEXT P%
470 '
480 B[2] = 0                                 'Finish combining coefficients
490 FOR I% = 0 TO 20
500   A[I%] = A[I%+2]
510   B[I%] = -B[I%+2]
520 NEXT I%
530 '
540 SA = 0                                   'NORMALIZE THE GAIN
550 SB = 0
560 FOR I% = 0 TO 20
570   IF LH = 0 THEN SA  = SA    + A[I%]
580   IF LH = 0 THEN SB  = SB    + B[I%]
590   IF LH = 1 THEN SA  = SA    + A[I%] * (-1)^I%
600   IF LH = 1 THEN SB  = SB    + B[I%] * (-1)^I%
610 NEXT I%
620 '
630 GAIN = SA / (1 - SB)
640 '
650 FOR I% = 0 TO 20
660   A[I%] = A[I%] / GAIN
670 NEXT I%
680 '                                        'The final recursion coefficients are in A[ ] and B[ ]
690 END
```

TABLE 20-4

```
1000 'THIS SUBROUTINE IS CALLED FROM TABLE 20-4,  LINE 340
1010 '
1020 '  Variables entering subroutine:        PI, FC, LH, PR, HP, P%
1030 '  Variables exiting subroutine:         A0, A1, A2, B1, B2
1040 '  Variables used internally:            RP, IP, ES, VX, KX, T, W, M, D, K,
1050 '                                        X0, X1, X2, Y1, Y2
1060 '
1070 '                                        'Calculate the pole location on the unit circle
1080 RP = -COS(PI/(NP*2) + (P%-1) * PI/NP)
1090 IP =  SIN(PI/(NP*2) + (P%-1) * PI/NP)
1100 '
1110 '                                        'Warp from a circle to an ellipse
1120 IF PR = 0 THEN GOTO 1210
1130 ES = SQR( (100 / (100-PR))^2 -1 )
1140 VX = (1/NP) * LOG( (1/ES) + SQR( (1/ES^2) + 1) )
1150 KX = (1/NP) * LOG( (1/ES) + SQR( (1/ES^2) - 1) )
1160 KX = (EXP(KX) + EXP(-KX))/2
1170 RP = RP * ( (EXP(VX) - EXP(-VX) ) /2 ) / KX
1180 IP  = IP * ( (EXP(VX) + EXP(-VX) ) /2 ) / KX
1190 '
1200 '                                        's-domain to z-domain conversion
1210 T  = 2 * TAN(1/2)
1220 W  = 2*PI*FC
1230 M  = RP^2 + IP^2
1240 D = 4 - 4*RP*T + M*T^2
1250 X0 = T^2/D
1260 X1 = 2*T^2/D
1270 X2 = T^2/D
1280 Y1 = (8 - 2*M*T^2)/D
1290 Y2 = (-4 - 4*RP*T - M*T^2)/D
1300 '
1310 '                                        'LP TO LP, or LP TO HP transform
1320 IF LH = 1 THEN K = -COS(W/2 + 1/2) / COS(W/2 - 1/2)
1330 IF LH = 0 THEN K =  SIN(1/2 - W/2) / SIN(1/2 + W/2)
1340 D = 1 + Y1*K - Y2*K^2
1350 A0 = (X0 - X1*K + X2*K^2)/D
1360 A1 = (-2*X0*K + X1 + X1*K^2 - 2*X2*K)/D
1370 A2 = (X0*K^2 - X1*K + X2)/D
1380 B1 = (2*K + Y1 + Y1*K^2 - 2*Y2*K)/D
1390 B2 = (-(K^2) - Y1*K + Y2)/D
1400 IF LH = 1 THEN A1 = -A1
1410 IF LH = 1 THEN B1 = -B1
1420 '
1430 RETURN
```

TABLE 20-5

TABLE 20-4 and 20-5

Program to calculate the "a" and "b" coefficients for Chebyshev recursive filters. In lines 270-300, four parameters are entered into the program. The cutoff frequency, FC, is expressed as a fraction of the sampling frequency, and therefore must be in the range: 0 to 0.5. The variable, LH, is set to a value of *one* for a high-pass filter, and *zero* for a low-pass filter. The value entered for PR must be in the range of 0 to 29, corresponding to 0 to 29% ripple in the filter's frequency response. The number of poles in the filter, entered in the variable NP, must be an even integer between 2 and 20. At the completion of the program, the "a" and "b" coefficients are stored in the arrays A[ ] and B[ ] (a_0 = A[0], a_1 = A[1], etc.). TABLE 20-5 is a subroutine called from line 340 of the main program. Six variables are passed to this subroutine, and five variables are returned. Table 20-6 (next page) contains two sets of data to help debug this subroutine. The functions: COS and SIN, use radians, not degrees. The function: LOG is the natural (base *e*) logarithm. Declaring all floating point variables (including the value of π) to be double precision will allow more poles to be used. Tables 20-1 and 20-2 were generated with this program and can be used to test for proper operation. Chapter 33 describes the mathematical operation of this program.

DATA SET 1		DATA SET 2	
Enter the subroutine with these values:			
FC	= 0.1	FC	= 0.1
LH	= 0	LH	= 1
PR	= 0	PR	= 10
NP	= 4	NP	= 4
P%	= 1	P%	= 2
PI	= 3.141592	PI	= 3.141592
These values should be present at line 1200:			
RP	= -0.923879	RP	= -0.136178
IP	= 0.382683	IP	= 0.933223
ES	= not used	ES	= 0.484322
VX	= not used	VX	= 0.368054
KX	= not used	KX	= 1.057802
These values should be present at line 1310:			
T	= 1.092605	T	= 1.092605
W	= 0.628318	W	= 0.628318
M	= 1.000000	M	= 0.889450
D	= 9.231528	D	= 5.656972
X0	= 0.129316	X0	= 0.211029
X1	= 0.258632	X1	= 0.422058
X2	= 0.129316	X2	= 0.211029
Y1	= 0.607963	Y1	= 1.038784
Y2	= -0.125227	Y2	= -0.789584
These values should be return to the main program:			
A0	= 0.061885	A0	= 0.922919
A1	= 0.123770	A1	= -1.845840
A2	= 0.061885	A2	= 0.922919
B1	= 1.048600	B1	= 1.446913
B2	= -0.296140	B2	= -0.836653

TABLE 20-6
Debugging data. This table contains two sets of data for debugging the subroutine listed in Table 20-5.

be obtained from the program in Table 20-4. The subroutine in Table 20-5 is called once for each stage in the cascade. For example, it is called three times for a six pole filter. At the completion of the subroutine, five variables are returned to the main program: *A0*, *A1*, *A2*, *B1*, & *B2*. These are the recursion coefficients for the two pole stage being worked on, and can be used to implement the filter in stages.

CHAPTER

21

Filter Comparison

Decisions, decisions, decisions! With all these filters to choose from, how do you know which to use? This chapter is a head-to-head competition between filters; we'll select champions from each side and let them fight it out. In the first match, *digital* filters are pitted against *analog* filters to see which technology is best. In the second round, the windowed-sinc is matched against the Chebyshev to find the king of the *frequency domain* filters. In the final battle, the moving average fights the single pole filter for the *time domain* championship. Enough talk; let the competition begin!

Match #1: Analog vs. Digital Filters

Most digital signals originate in analog electronics. If the signal needs to be filtered, is it better to use an analog filter before digitization, or a digital filter after? We will answer this question by letting two of the best contenders deliver their blows.

The goal will be to provide a low-pass filter at 1 kHz. Fighting for the analog side is a six pole Chebyshev filter with 0.5 dB (6%) ripple. As described in Chapter 3, this can be constructed with 3 op amps, 12 resistors, and 6 capacitors. In the digital corner, the windowed-sinc is warming up and ready to fight. The analog signal is digitized at a 10 kHz sampling rate, making the cutoff frequency 0.1 on the digital frequency scale. The length of the windowed-sinc will be chosen to be 129 points, providing the same 90% to 10% roll-off as the analog filter. Fair is fair. Figure 21-1 shows the frequency and step responses for these two filters.

Let's compare the two filters blow-by-blow. As shown in (a) and (b), the analog filter has a 6% ripple in the passband, while the digital filter is perfectly flat (within 0.02%). The analog designer might argue that the ripple can be *selected* in the design; however, this misses the point. The flatness achievable with analog filters is limited by the accuracy of their resistors and

capacitors. Even if a Butterworth response is designed (i.e., 0% ripple), filters of this complexity will have a residue ripple of, perhaps, 1%. On the other hand, the flatness of digital filters is primarily limited by round-off error, making them *hundreds* of times flatter than their analog counterparts. Score one point for the digital filter.

Next, look at the frequency response on a log scale, as shown in (c) and (d). Again, the digital filter is clearly the victor in both *roll-off* and *stopband attenuation*. Even if the analog performance is improved by adding additional stages, it still can't compare to the digital filter. For instance, imagine that you need to improve these two parameters by a factor of 100. This can be done with simple modifications to the windowed-sinc, but is virtually impossible for the analog circuit. Score two more for the digital filter.

The step response of the two filters is shown in (e) and (f). The digital filter's step response is symmetrical between the lower and upper portions of the step, i.e., it has a linear phase. The analog filter's step response is *not* symmetrical, i.e., it has a nonlinear phase. One more point for the digital filter. Lastly, the analog filter overshoots about 20% on one side of the step. The digital filter overshoots about 10%, but on both sides of the step. Since both are bad, no points are awarded.

In spite of this beating, there are still many applications where analog filters should, or must, be used. This is not related to the actual performance of the filter (i.e., what goes in and what comes out), but to the general advantages that analog circuits have over digital techniques. The first advantage is *speed*: digital is slow; analog is fast. For example, a personal computer can only filter data at about 10,000 samples per second, using FFT convolution. Even simple op amps can operate at 100 kHz to 1 MHz, 10 to 100 times as fast as the digital system!

The second inherent advantage of analog over digital is *dynamic range*. This comes in two flavors. **Amplitude dynamic range** is the ratio between the largest signal that can be passed through a system, and the inherent noise of the system. For instance, a 12-bit ADC has a saturation level of 4095, and an rms quantization noise of 0.29 digital numbers, for a dynamic range of about 14000. In comparison, a standard op amp has a saturation voltage of about 20 volts and an internal noise of about 2 microvolts, for a dynamic range of about *ten million*. Just as before, a simple op amp devastates the digital system.

The other flavor is **frequency dynamic range**. For example, it is easy to design an op amp circuit to simultaneously handle frequencies between 0.01 Hz and 100 kHz (seven decades). When this is tried with a digital system, the computer becomes swamped with data. For instance, sampling at 200 kHz, it takes 20 million points to capture one complete cycle at 0.01 Hz. You may have noticed that the frequency response of digital filters is almost always plotted on a *linear* frequency scale, while analog filters are usually displayed with a *logarithmic* frequency. This is because digital filters need

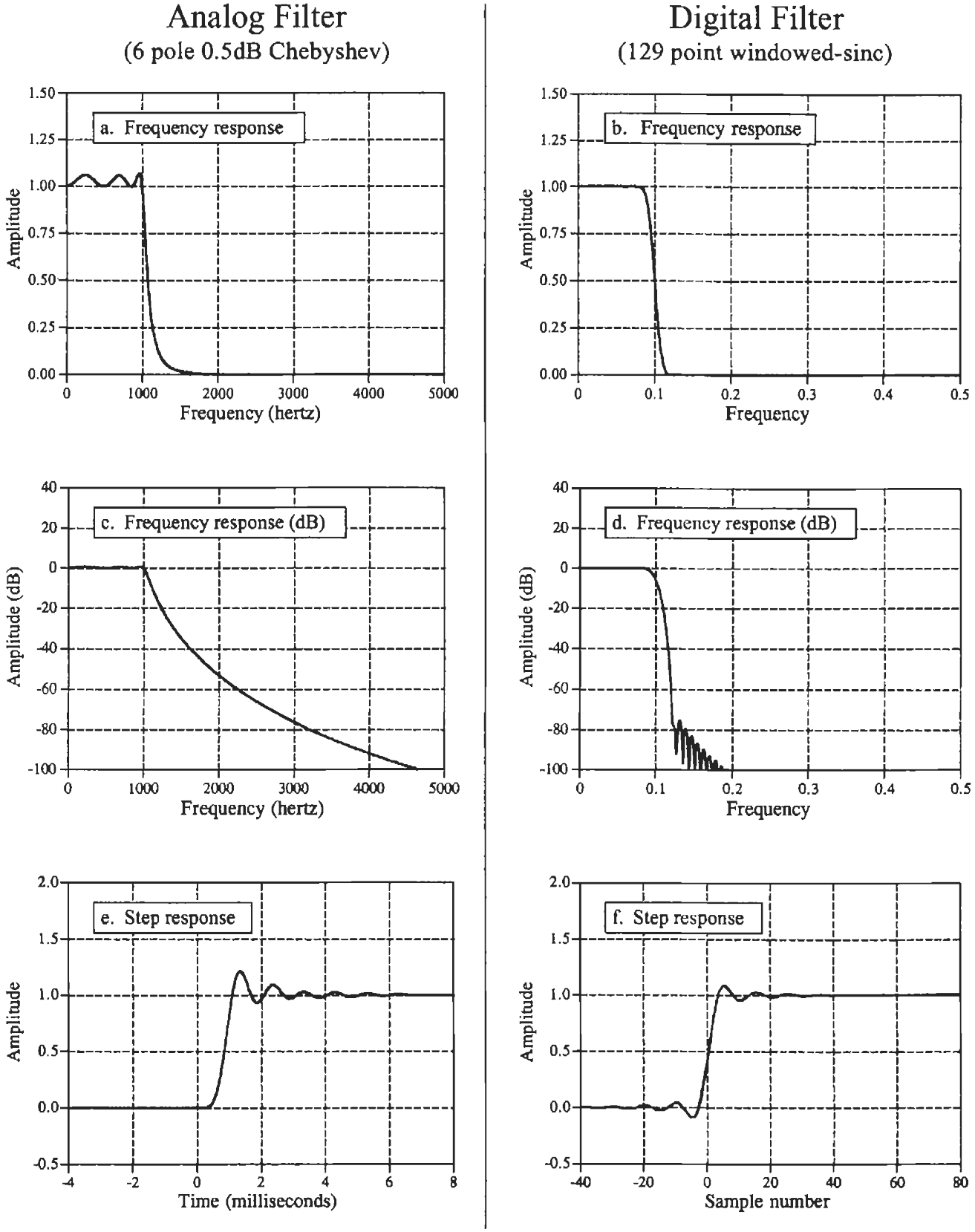

FIGURE 21-1
Comparison of analog and digital filters. Digital filters have better performance in many areas, such as: passband ripple, (a) vs. (b), roll-off and stopband attenuation, (c) vs. (d), and step response symmetry, (e) vs. (f). The digital filter in this example has a cutoff frequency of 0.1 of the 10 kHz sampling rate. This provides a fair comparison to the 1 kHz cutoff frequency of the analog filter.

a linear scale to show their exceptional filter performance, while analog filters need the logarithmic scale to show their huge dynamic range.

Match #2: Windowed-Sinc vs. Chebyshev

Both the windowed-sinc and the Chebyshev filters are designed to separate one band of frequencies from another. The windowed-sinc is an FIR filter implemented by *convolution*, while the Chebyshev is an IIR filter carried out by *recursion*. Which is the best digital filter in the frequency domain? We'll let them fight it out.

The recursive filter contender will be a 0.5% ripple, 6 pole Chebyshev low-pass filter. A fair comparison is complicated by the fact that the Chebyshev's frequency response changes with the cutoff frequency. We will use a cutoff frequency of 0.2, and select the windowed-sinc's filter kernel to be 51 points. This makes both filters have the same 90% to 10% roll-off, as shown in Fig. 21-2(a).

Now the pushing and shoving begins. The recursive filter has a 0.5% ripple in the passband, while the windowed-sinc is flat. However, we could easily set the recursive filter ripple to 0% if needed. No points. Figure 21-2b shows that the windowed-sinc has a much better stopband attenuation than the Chebyshev. One point for the windowed-sinc.

Figure 21-3 shows the step response of the two filters. Both are bad, as you should expect for frequency domain filters. The recursive filter has a nonlinear phase, but this can be corrected with bidirectional filtering. Since both filters are so ugly in this parameter, we will call this a draw.

So far, there isn't much difference between these two filters; either will work when moderate performance is needed. The heavy hitting comes over two critical issues: *maximum performance* and *speed*. The windowed-sinc is a powerhouse, while the Chebyshev is quick and agile. Suppose you have a really tough frequency separation problem, say, needing to isolate a 100

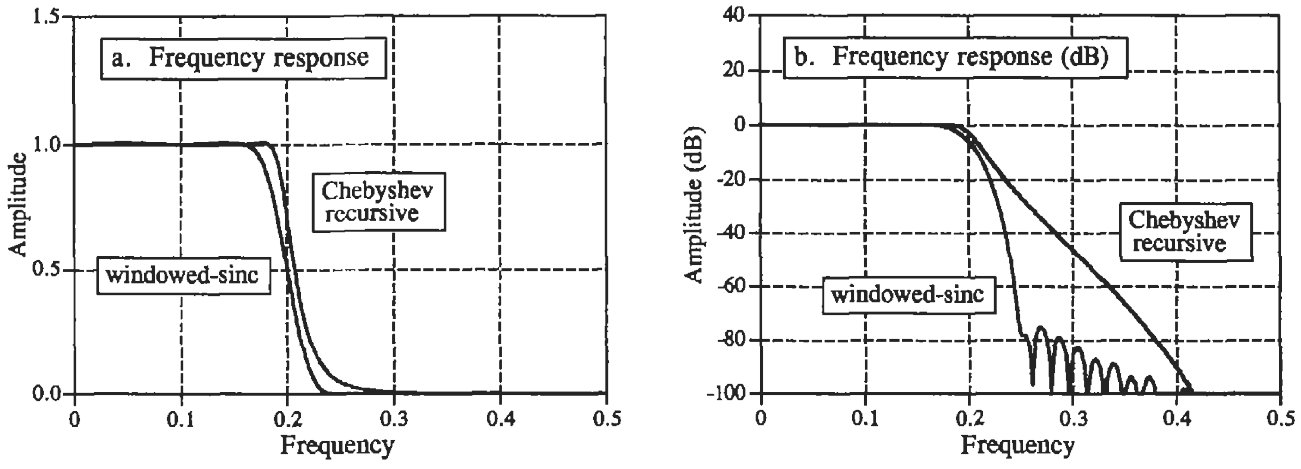

FIGURE 21-2
Windowed-sinc and Chebyshev frequency responses. Frequency responses are shown for a 51 point windowed-sinc filter and a 6 pole, 0.5% ripple Chebyshev recursive filter. The windowed-sinc has better stopband attenuation, but either will work in moderate performance applications. The cutoff frequency of both filters is 0.2, measured at an amplitude of 0.5 for the windowed-sinc, and 0.707 for the recursive.

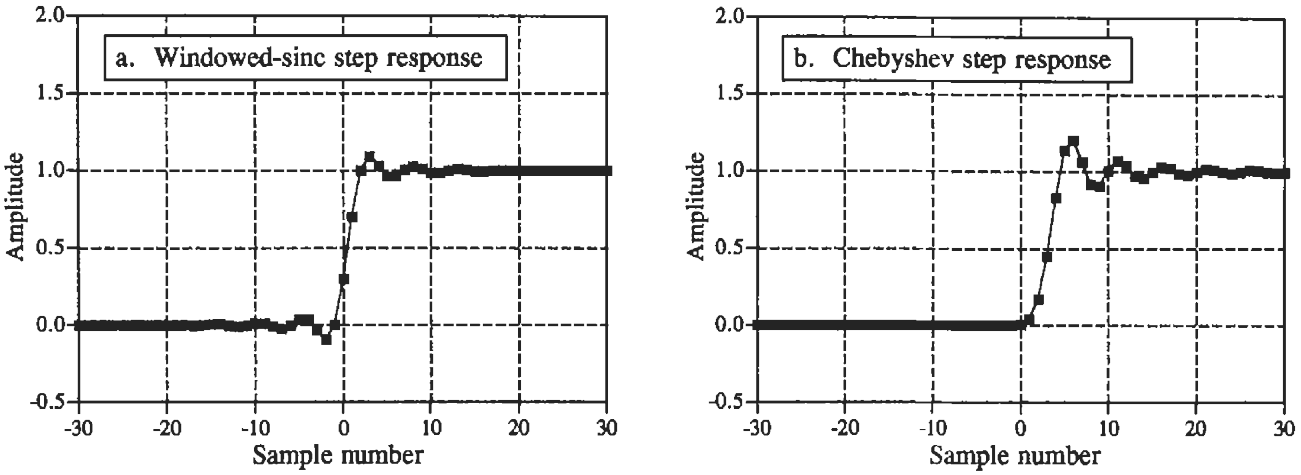

FIGURE 21-3
Windowed--sinc and Chebyshev step responses. The step responses are shown for a 51 point windowed-sinc filter and a 6 pole, 0.5% ripple Chebyshev recursive filter. Each of these filters has a cutoff frequency of 0.2. The windowed-sinc has a slightly better step response because it has less overshoot and a zero phase.

millivolt signal at 61 hertz that is riding on a 120 volt power line at 60 hertz. Figure 21-4 shows how these two filters compare when you need maximum performance. The recursive filter is a 6 pole Chebyshev with 0.5% ripple. This is the maximum number of poles that can be used at a 0.05 cutoff frequency with single precision. The windowed-sinc uses a 1001 point filter kernel, formed by convolving a 501 point windowed-sinc filter kernel with itself. As shown in Chapter 16, this provides greater stopband attenuation.

How do these two filters compare when maximum performance is needed? The windowed-sinc crushes the Chebyshev! Even if the recursive filter were improved (more poles, multistage implementation, double precision, etc.), it is still no match for the FIR performance. This is especially impressive when you consider that the windowed-sinc has only begun to fight. There are strong limits on the maximum performance that recursive filters can provide. The windowed-sinc, in contrast, can be pushed to incredible levels. This is, of course, provided you are willing to wait for the result. Which brings up the second critical issue: *speed.*

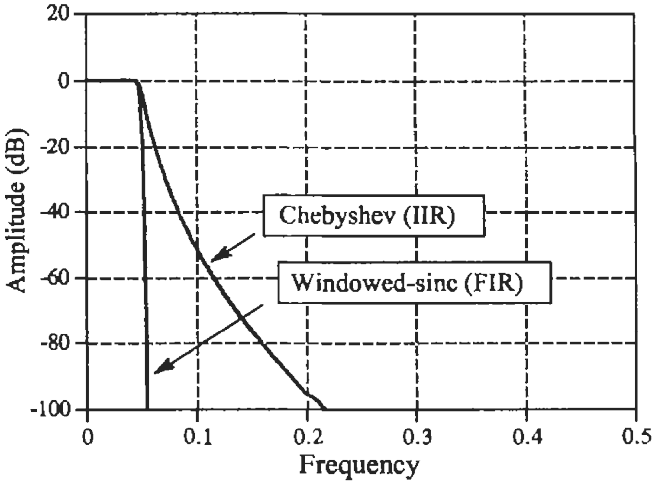

FIGURE 21-4
Maximum performance of FIR and IIR filters. The frequency response of the windowed-sinc can be virtually any shape needed, while the Chebyshev recursive filter is very limited. This graph compares the frequency response of a six pole Chebyshev recursive filter with a 1001 point windowed-sinc filter.

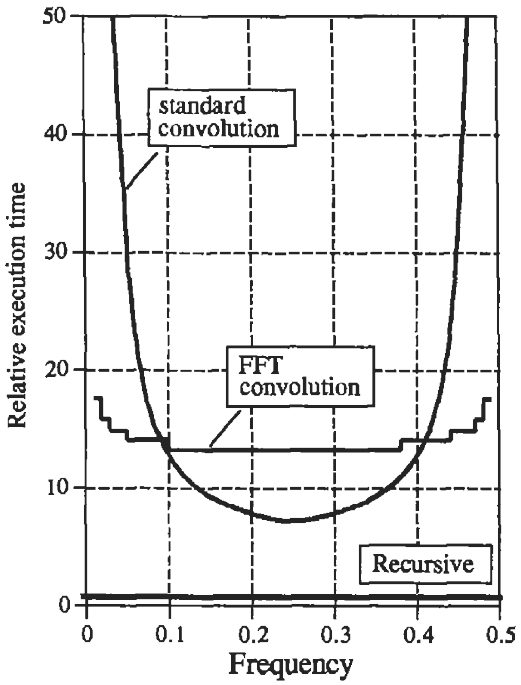

FIGURE 21-5
Comparing FIR and IIR execution speeds. These curves shows the relative execution times for a windowed-sinc filter compared with an equivalent six pole Chebyshev recursive filter. Curves are shown for implementing the FIR filter by both the standard and the FFT convolution algorithms. The windowed-sinc execution time rises at low and high frequencies because the filter kernel must be made longer to keep up with the greater performance of the recursive filter at these frequencies. In general, IIR filters are an order of magnitude faster than FIR filters of comparable performance.

Comparing these filters for speed is like racing a Ferrari against a go-cart. Figure 21-5 shows how much longer the windowed-sinc takes to execute, compared to a six pole recursive filter. Since the recursive filter has a faster roll-off at low and high frequencies, the length of the windowed-sinc kernel must be made *longer* to match the performance (i.e., to keep the comparison fair). This accounts for the increased execution time for the windowed-sinc near frequencies 0 and 0.5. The important point is that FIR filters can be expected to be about an order of magnitude slower than comparable IIR filters (go-cart: 15 mph, Ferrari: 150 mph).

Match #3: Moving Average vs. Single Pole

Our third competition will be a battle of the time domain filters. The first fighter will be a nine point moving average filter. Its opponent for today's match will be a single pole recursive filter using the bidirectional technique. To achieve a comparable frequency response, the single pole filter will use a sample-to-sample decay of $x = 0.70$. The battle begins in Fig. 21-6 where the frequency response of each filter is shown. Neither one is very impressive, but of course, frequency separation isn't what these filters are used for. No points for either side.

Figure 21-7 shows the step responses of the filters. In (a), the moving average step response is a straight line, the most rapid way of moving from one level to another. In (b), the recursive filter's step response is smoother, which may be better for some applications. One point for each side.

These filters are quite equally matched in terms of performance and often the choice between the two is made on personal preference. However, there are

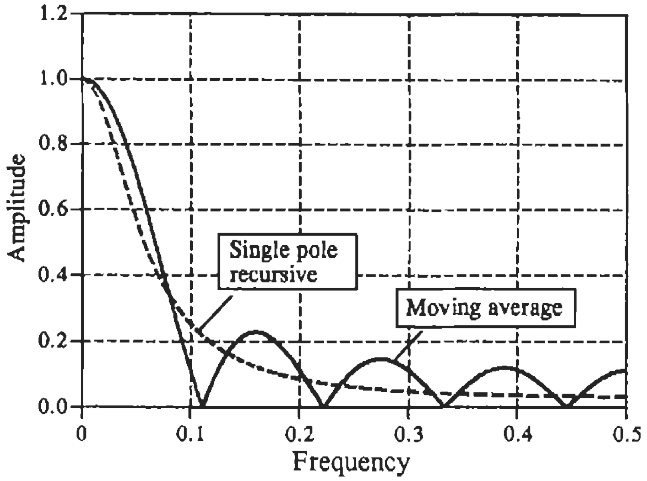

FIGURE 21-6
Moving average and single pole frequency responses. Both of these filters have a poor frequency response, as you should expect for time domain filters.

two cases where one filter has a slight edge over the other. These are based on the trade-off between *development* time and *execution* time. In the first instance, you want to reduce development time and are willing to accept a slower filter. For example, you might have a one time need to filter a few thousand points. Since the entire program runs in only a few seconds, it is pointless to spend time optimizing the algorithm. Floating point will almost certainly be used. The choice is to use the moving average filter carried out by convolution, or a single pole recursive filter. The winner here is the recursive filter. It will be slightly easier to program and modify, and will execute much faster.

The second case is just the opposite; your filter must operate as fast as possible and you are willing to spend the extra development time to get it. For instance, this filter might be a part of a commercial product, with the potential to be run *millions* of times. You will probably use integers for the highest possible speed. Your choice of filters will be the moving average

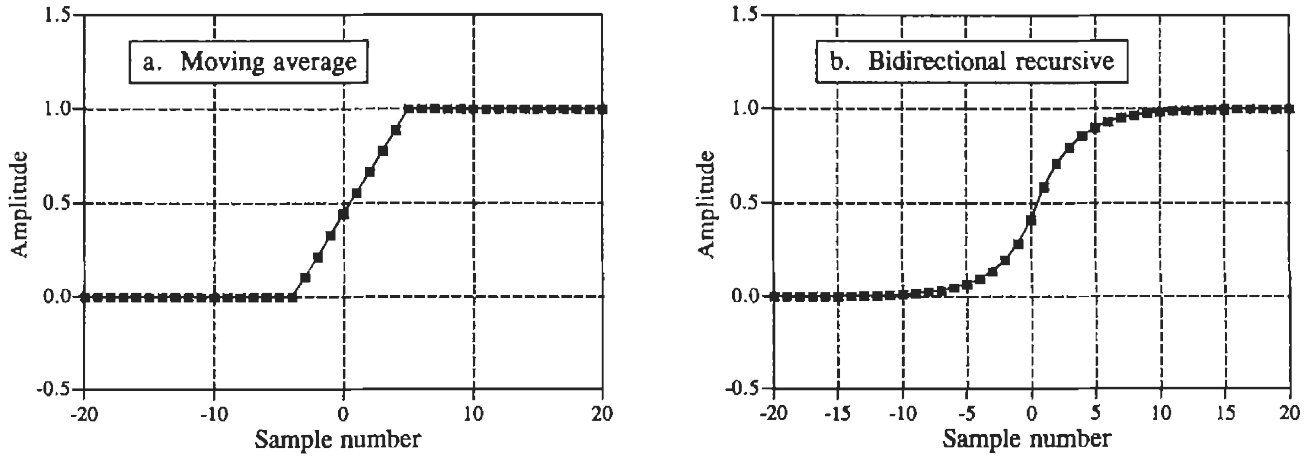

FIGURE 21-7
Step responses of the moving average and the bidirectional single pole filter. The moving average step response occurs over a smaller number of samples, while the single pole filter's step response is smoother.

carried out by *recursion*, or the single pole recursive filter implemented with look-up tables or integer math. The winner is the moving average filter. It will execute faster and not be susceptible to the development and execution problems of integer arithmetic.