[8] Wait, J. V. "Digital Filters," in *Active Filters: Lumped, Distributed, Integrated, Digital, and Parametric*. Ed. by L. P. Huelsman. McGraw-Hill, New York, 1970, pp. 200–277.

[9] Dolph, C. L. "A Current Distribution for Broadside Arrays Which Optimizes the Relationship between Beam Width and Side-Lobe Level," *Proceedings of the IRE*, Vol. 35, June 1946.

[10] Barbiere, D. "A Method for Calculating the Current Distribution of Chebyshev Arrays," *Proceedings of the IRE*, Vol. 40, January 1952.

[11] Cook, C. E., and Bernfeld, M. *Radar Signals*, Academic Press, New York, 1967, pp. 178–180.

[12] Kaiser, J. F. "Digital Filters," in *System Analysis by Digital Computer*. Ed. by F. F. Kuo and J. F. Kaiser, John Wiley and Sons, New York, 1966, pp. 218–277.

[13] Williams, C. S. *Designing Digital Filters*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986, pp. 117.

[14] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*, Vol. 66, No. 1, January 1978.

[15] McClellan, J. H., Parks, T. W., and Rabiner, L. R. "A Computer Program for Designing Optimum FIR Linear Phase Digital Filters," *IEEE Trans. on Audio and Electroacoustics*, Vol. AU-21, No. 6, December 1973, pp. 515.

[16] Rabiner, L. R. and Gold, B. *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975, pp. 136.

[17] Parks, T. W., and McClellan, J. H. "Chebyshev Approximation For Nonrecursive Digital Filters with Linear Phase," *IEEE Trans. on Circuit Theory*, Vol. CT-19, March 1972.

[18] McClellan, J. H., and Parks, T. W. "A Unified Approach to the Design of Optimum FIR Linear Phase Digital Filters," *IEEE Trans. on Circuit Theory*, Vol. CT-20, November 1973.

[19] Rabiner, L. R., McClellan, J. H., and Parks, T. W. "FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximation," *Proc. IEEE*, Vol. 63, No. 4, April 1975.

[20] Oppenheim, A. V., and Schafer, R. W. *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989, pp. 478.

[21] Funderburk, D. M., and Park, S. "Implementation of a C-QUAM AM-Stereo Receiver Using a General Purpose DSP Device," *RF Design*, June 1993.

[22] Harris Semiconductor Inc. "A Digital, 16-Bit, 52 Msps Halfband Filter," *Microwave Journal*, September 1993.

[23] Ballanger, M. G. "Computation Rate and Storage Estimation in Multirate Digital Filtering with Half-Band Filters," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-25, No. 4, August 1977.

Sony v. Jawbone
U.S. Patent No. 7,246,058
Sony Ex. 1008 (Part 2 of 3)

[24] Crochiere, R. E., and Rabiner, L. R. "Decimation and Interpolation of Digital Signals—A Tutorial Review," *Proceedings of the IEEE*, Vol. 69, No. 3, March 1981, pp. 318.

[25] Ballanger, M. G., Daguet, J. L., and Lepagnol, G. P. "Interpolation, Extrapolation, and Reduction of Computational Speed in Digital Filters," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-22, No. 4, August 1974.

[26] Oppenheim, A. V., Willsky, A. S., and Young, I. T. *Signals and Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983, pp. 212.

[27] Stearns, S. *Digital Signal Analysis*, Hayden Book Co., Rochelle Park, New Jersey, 1975, pp. 93.

[28] Oppenheim, A. V., and Schafer, R. W. *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989, pp. 58.

[29] Rabiner, L. R., and Gold, B. *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975, pp. 59.

[30] Oppenheim, A. V., Willsky, A. S., and Young, I. T. *Signals and Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983, pp. 201.

CHAPTER SIX

Infinite Impulse Response Filters

Infinite impulse response (IIR) digital filters are fundamentally different from FIR filters because practical IIR filters always require feedback. Where FIR filter output samples depend only on past input samples, each IIR filter output sample depends on previous input samples *and* previous filter output samples. IIR filters' *memory* of past outputs is both a blessing and a curse. Like all feedback systems, perturbations at the IIR filter input could, depending on the design, cause the filter output to become unstable and oscillate indefinitely. This characteristic of possibly having an infinite duration of nonzero output samples, even if the input becomes all zeros, is the origin of the phrase *infinite impulse response*. It's interesting at this point to know that, relative to FIR filters, IIR filters have more complicated structures (block diagrams), are harder to design and analyze, and do not have linear phase responses. Why in the world, then, would anyone use an IIR filter? Because they are very efficient. IIR filters require far fewer multiplications per filter output sample to achieve a given frequency magnitude response. From a hardware standpoint, this means that IIR filters can be very fast, allowing us to build real-time IIR filters that operate over much higher sample rates than FIR filters.[†]

To illustrate the utility of IIR filters, Figure 6-1 contrasts the frequency magnitude responses of what's called a fourth-order low-pass IIR filter and the 19-tap FIR filter of Figure 5-19(b) from Chapter 5. Where the 19-tap FIR filter in Figure 6-1 requires 19 multiplications per filter output sample, the fourth-order IIR filter requires only 9 multiplications for each filter output sample. Not only does the IIR filter give us reduced

[†] At the end of this chapter, we briefly compare the advantages and disadvantages of IIR filters relative to FIR filters.

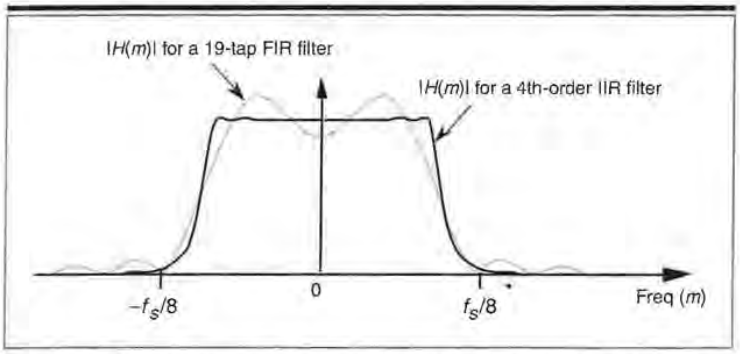


Figure 6-1 Comparison of the frequency magnitude responses of a 19-tap low-pass FIR filter and a 4th-order low-pass IIR filter.

passband ripple and a sharper filter roll-off; it does so with less than half the multiplication workload of the FIR filter.

Recall from Section 5.3 that, to force an FIR filter's frequency response to have very steep transition regions, we had to design an FIR filter with a very long impulse response. The longer the impulse response, the more ideal our filter frequency response will become. From a hardware standpoint, the maximum number of FIR filter taps we can have (the length of the impulse response) depends on how fast our hardware can perform the required number of multiplications and additions to get a filter output value before the next filter input sample arrives. IIR filters, however, can be designed to have impulse responses that are longer than their number of taps! Thus, IIR filters can give us much better filtering for a given number of multiplications per output sample than FIR filters. With this in mind, let's take a deep breath, flex our mathematical muscles, and learn about IIR filters.

6.1 An Introduction to Infinite Impulse Response Filters

IIR filters get their name from the fact that some of the filter's previous output samples are used to calculate the current output sample. Given a finite duration of nonzero input values, the effect is that an IIR filter could have a infinite duration of nonzero output samples. So, if the IIR filter's input suddenly becomes a sequence of all zeros, the filter's output could conceivably remain nonzero forever. This peculiar attribute of IIR filters comes about because of the way they're realized, i.e., the feedback structure of their delay units, multipliers, and adders. Understanding IIR filter

structures is straightforward if we start by recalling the building blocks of an FIR filter. Figure 6-2(a) shows the now familiar structure of a 4-tap FIR digital filter that implements the time-domain FIR equation

$$y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + h(3)x(n-3) + h(4)x(n-4) . \quad (6\text{-}1)$$

Although not specifically called out as such in Chapter 5, Eq. (6-1) is known as a *difference equation*. To appreciate how past filter output samples are used in the structure of IIR filters, let's begin by reorienting our FIR structure in Figure 6-2(a) to that of Figure 6-2(b). Notice how the structures in Figure 6-2 are computationally identical, and both are implementations, or realizations, of Eq. (6-1).

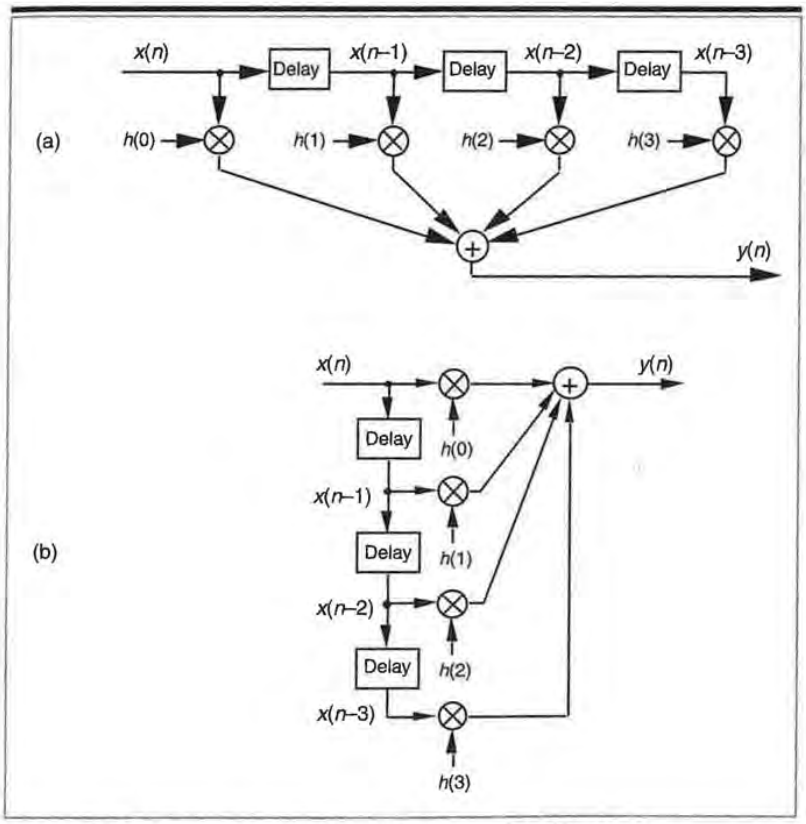


Figure 6-2 FIR digital filter structures: (a) traditional FIR filter structure; (b) rearranged, but equivalent, FIR filter structure.

We can now show how past filter output samples are combined with past input samples by using the IIR filter structure in Figure 6-3. Because IIR filters have two sets of coefficients, we'll use the standard notation of the variables $b(k)$ to denote the feed forward coefficients and the variables $a(k)$ to indicate the feedback coefficients in Figure 6-3. OK, the difference equation describing the IIR filter in Figure 6-3 is

$$y(n) = b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + b(3)x(n-3)$$
$$+ a(1)y(n-1) + a(2)y(n-2) + a(3)y(n-3) \quad . \tag{6-2}$$

Look at Figure 6-3 and Eq. (6-2) carefully. It's important to convince ourselves that Figure 6-3 really is a valid implementation of Eq. (6-2) and that, conversely, difference equation Eq. (6-2) fully describes the IIR filter structure in Figure 6-3. Keep in mind now that the sequence $y(n)$ in Figure 6-3 is not the same $y(n)$ sequence that's shown in Figure 6-2. The $d(n)$ sequence in Figure 6-3 is equal to the $y(n)$ sequence in Figure 6-2.

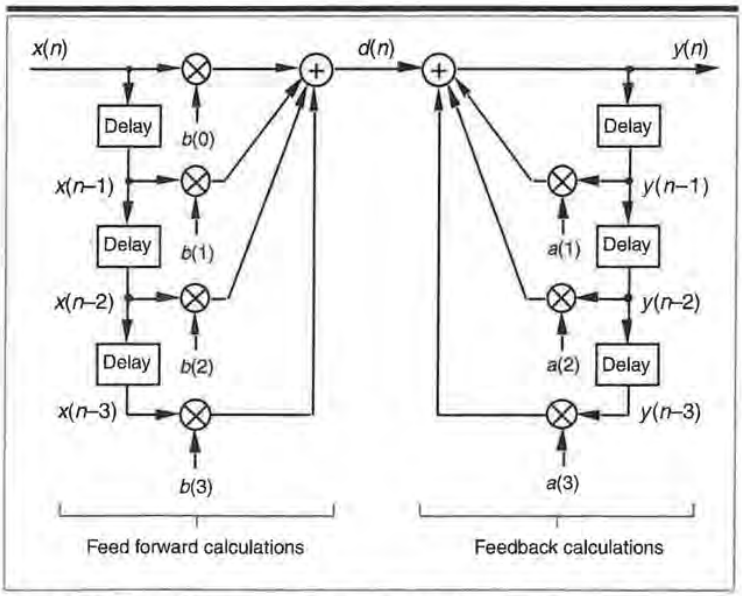


Figure 6-3 IIR digital filter structure showing feed forward and feedback calculations.

By now you're probably wondering, "Just how do we determine those $a(k)$ and $b(k)$ IIR filter coefficients if we actually want to design an IIR filter?" Well, fasten your seat belt because this is where we get serious about understanding IIR filters. Recall from the last chapter concerning the window method of low-pass FIR filter design that we defined the frequency response of our desired FIR filter, took the inverse Fourier transform of that frequency response, and then shifted that transform result to get the filter's time-domain impulse response. Happily, due to the nature of transversal FIR filters, the desired $h(k)$ filter coefficients turned out to be exactly equal to the impulse response sequence. Following that same procedure with IIR filters, we could define the desired frequency response of our IIR filter and then take the inverse Fourier transform of that response to yield the filter's time-domain impulse response. The bad news is that there's no direct method for computing the IIR filter's $a(k)$ and $b(k)$ coefficients from the impulse response! Unfortunately, the FIR filter design techniques that we've learned so far simply cannot be used to design IIR filters. Fortunately for us, this wrinkle can be ironed out by using one of several available methods of designing IIR filters.

Standard IIR filter design techniques fall into three basic classes: the impulse invariance, bilinear transform, and optimization methods. These design methods use a discrete sequence, mathematical transformation process known as the z-transform whose origin is the Laplace transform traditionally used in the analyzing of continuous systems. With that in mind, let's start this IIR filter analysis and design discussion by briefly reacquainting ourselves with the fundamentals of the Laplace transform.

6.2 The Laplace Transform

The Laplace transform is a mathematical method of solving linear differential equations that has proved very useful in the fields of engineering and physics. This transform technique, as it's used today, originated from the work of the brilliant English physicist Oliver Heaviside.[†] The fundamental process of using the Laplace transform goes something like the following:

[†] Heaviside (1850–1925), who was interested in electrical phenomena, developed an efficient algebraic process of solving differential equations. He initially took a lot of heat from his contemporaries because they thought his work was not sufficiently justified from a mathematical standpoint. However, the discovered correlation of Heaviside's methods with the rigorous mathematical treatment of the French mathematician Marquis Pierre Simon de Laplace's (1749–1827) operational calculus verified the validity of Heaviside's techniques.

Step 1: A time-domain differential equation is written that describes the input/output relationship of a physical system (and we want to find the output function that satisfies that equation with a given input).

Step 2: The differential equation is Laplace transformed, converting it to an algebraic equation.

Step 3: Standard algebraic techniques are used to determine the desired output function's equation in the Laplace domain.

Step 4: The desired Laplace output equation is, then, inverse Laplace transformed to yield the desired time-domain output function's equation.

This procedure, at first, seems cumbersome because it forces us to *go the long way around*, instead of just solving a differential equation directly. The justification for using the Laplace transform is that although solving differential equations by classical methods is a very powerful analysis technique for all but the most simple systems, it can be tedious and (for some of us) error-prone. The reduced complexity of using algebra outweighs the extra effort needed to perform the required forward and inverse Laplace transformations. This is especially true now that tables of forward and inverse Laplace transforms exist for most of the commonly encountered time functions. Well-known properties of the Laplace transform also allow practitioners to decompose complicated time functions into combinations of simpler functions and, then, use the tables. (Tables of Laplace transforms allow us to translate quickly back and forth between a time function and its Laplace transform—analogous to, say, a German-English dictionary if we were studying the German language.[†]) Let's briefly look at a few of the more important characteristics of the Laplace transform that will prove useful as we make our way toward the discrete z-transform used to design and analyze IIR digital filters.

The Laplace transform of a continuous time-domain function $f(t)$, where $f(t)$ is defined only for positive time ($t > 0$), is expressed mathematically as

$$F(s) = \int_0^\infty f(t)e^{-st}dt \quad . \tag{6-3}$$

[†] Although tables of commonly encountered Laplace transforms are included in almost every system analysis textbook, very comprehensive tables are also available[1–3].

$F(s)$ is called "the Laplace transform of $f(t)$," and the variable s is the complex number

$$s = \sigma + j\omega . \tag{6-4}$$

A more general expression for the Laplace transform, called the bilateral or two-sided transform, uses negative infinity ($-\infty$) as the lower limit of integration. However, for the systems that we'll be interested in, where system conditions for negative time ($t < 0$) are not needed in our analysis, the *one-sided* Eq. (6-3) applies. Those systems, often referred to as causal systems, may have initial conditions at $t = 0$ that must be taken into account (velocity of a mass, charge on a capacitor, temperature of a body, etc.) but we don't need to know what the system was doing prior to $t = 0$.

In equation (6-4), σ is a real number and ω is frequency in radians/second. Because e^{-st} is dimensionless, the exponent term s must have the dimension of 1/time, or frequency. That's why the Laplace variable s is often called a complex frequency.

To put Eq. (6-3) into words, we can say that it requires us to multiply, point for point, the function $f(t)$ by the complex function e^{-st} for a given value of s. (We'll soon see that using the function e^{-st} here is not accidental; e^{-st} is used because it's the general form for the solution of linear differential equations.) After the point-for-point multiplications, we find the area under the curve of the function $f(t)e^{-st}$ by summing all the products. That area, a complex number, represents the value of the Laplace transform for the particular value of $s = \sigma + j\omega$ chosen for the original multiplications. If we were to go through this process for all values of s, we'd have a full description of $F(s)$ for every value of s.

I like to think of the Laplace transform as a continuous function, where the complex value of that function for a particular value of s is a correlation of $f(t)$ and a damped complex e^{-st} sinusoid whose frequency is ω and whose damping factor is σ. What do these complex sinusoids look like? Well, they are rotating phasors described by

$$e^{-st} = e^{-(\sigma+j\omega)t} = e^{-\sigma t}e^{-j\omega t} = \frac{e^{-j\omega t}}{e^{\sigma t}} . \tag{6-5}$$

From our knowledge of complex numbers, we know that $e^{-j\omega t}$ is a unity-magnitude phasor rotating clockwise around the origin of a complex plane at a frequency of ω radians per second. The denominator of Eq. (6-5) is a real number whose value is one at time $t = 0$. As t increases, the denominator $e^{\sigma t}$ gets larger (when σ is positive), and the complex e^{-st} phasor's

magnitude gets smaller as the phasor rotates on the complex plane. The tip of that phasor traces out a curve spiraling in toward the origin of the complex plane. One way to visualize a complex sinusoid is to consider its real and imaginary parts individually. We do this by expressing the complex e^{-st} sinusoid from Eq. (6-5) in rectangular form as

$$e^{-st} = \frac{e^{-j\omega t}}{e^{\sigma t}} = \frac{\cos(\omega t)}{e^{\sigma t}} - j\frac{\sin(\omega t)}{e^{\sigma t}}. \qquad (6\text{-}5')$$

Figure 6-4 shows the real parts (cosine) of several complex sinusoids with different frequencies and different damping factors. In Figure 6-4(a), the complex sinusoid's frequency is the arbitrary ω', and the damping factor is the arbitrary σ'. So the real part of $F(s)$, at $s = \sigma' + j\omega'$, is equal to the correlation of $f(t)$ and the wave in Figure 6-4(a). For different values of s, we'll correlate $f(t)$ with different complex sinusoids as shown in Figure 6-4. (As we'll see, this correlation is very much like the correlation of $f(t)$ with various sine and cosine waves when we were calculating the discrete Fourier transform.) Again, the real part of $F(s)$, for a particular value of s, is the correlation of $f(t)$ with a cosine wave of frequency ω and a damping factor of σ, and the imaginary part of $F(s)$ is the correlation of $f(t)$ with a sinewave of frequency ω and a damping factor of σ.

Now, if we associate each of the different values of the complex s variable with a point on a complex plane, rightfully called the s-plane, we could plot the real part of the $F(s)$ correlation as a surface above (or below) that s-plane and generate a second plot of the imaginary part of the $F(s)$ correlation as a surface above (or below) the s-plane. We can't plot the full complex $F(s)$ surface on paper because that would require four dimensions. That's because s is complex, requiring two dimensions, and $F(s)$ is itself complex and also requires two dimensions. What we can do, however, is graph the magnitude $|F(s)|$ as a function of s because this graph requires only three dimensions. Let's do that as we demonstrate this notion of an $|F(s)|$ surface by illustrating the Laplace transform in a tangible way.

Say, for example, that we have the linear system shown in Figure 6-5. Also, let's assume that we can relate the $x(t)$ input and the $y(t)$ output of the linear time invariant physical system in Figure 6-5 with the following messy homogeneous constant-coefficient differential equation

$$a_2\frac{d^2y(t)}{dt^2} + a_1\frac{dy(t)}{dt} + a_0y(t) = b_1\frac{dx(t)}{dt} + b_0x(t) \quad . \qquad (6\text{-}6)$$

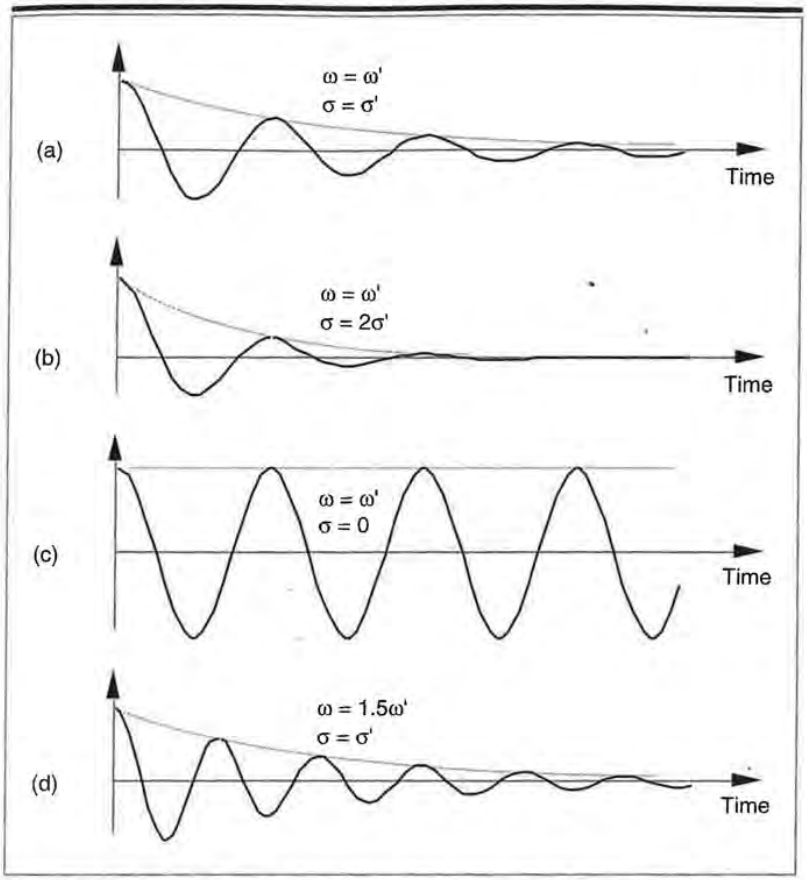


Figure 6-4 Real part (cosine) of various e^{-st} functions, where $s = \sigma + j\omega$, to be correlated with $f(t)$.

We'll use the Laplace transform toward our goal of figuring out how the system will behave when various types of input functions are applied, i.e., what will the $y(t)$ output be for any given $x(t)$ input.

Let's slow down here and see exactly what Figure 6-5 and Eq. (6-6) are telling us. First, if the system is time invariant, then the a_n and b_n coefficients in Eq. (6-6) are constant. They may be positive or negative, zero, real or complex, but they do not change with time. If the system is electrical, the coefficients might be related to capacitance, inductance, and resistance. If the system is mechanical with masses and springs, the coefficients could be related to mass, coefficient of damping, and coefficient of

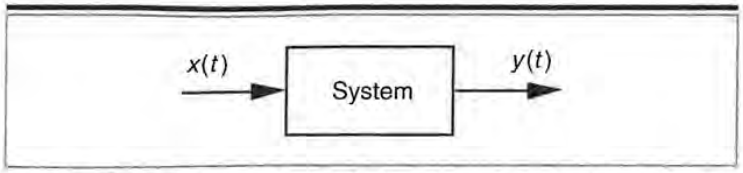


Figure 6-5 System described by Eq. (6-6). The system's input and output are the continuous time functions $x(t)$ and $y(t)$ respectively.

resilience. Then, again, if the system is thermal with masses and insulators, the coefficients would be related to thermal capacity and thermal conductance. To keep this discussion general, though, we don't really care what the coefficients actually represent.

OK, Eq. (6-6) also indicates that, ignoring the coefficients for the moment, the sum of the $y(t)$ output plus derivatives of that output is equal to the sum of the $x(t)$ input plus the derivative of that input. Our problem is to determine exactly what input and output functions satisfy the elaborate relationship in Eq. (6-6). (For the stout hearted, classical methods of solving differential equations could be used here, but the Laplace transform makes the problem much simpler for our purposes.) Thanks to Laplace, the complex exponential time function of e^{st} is the one we'll use. It has the beautiful property that it can be differentiated any number of times without destroying its original form. That is

$$\frac{d(e^{st})}{dt} = se^{st},\ \frac{d^2(e^{st})}{dt^2} = s^2e^{st},\ \frac{d^3(e^{st})}{dt^3} = s^3e^{st},\ldots,\ \frac{d^n(e^{st})}{dt^n} = s^ne^{st}\ . \quad (6\text{-}7)$$

If we let $x(t)$ and $y(t)$ be functions of e^{st}, $x(e^{st})$ and $y(e^{st})$, and use the properties shown in Eq. (6-7), Eq. (6-6) becomes

$$a_2s^2y(e^{st}) + a_1sy(e^{st}) + a_0y(e^{st}) = b_1sx(e^{st}) + b_0x(e^{st}),$$

or

$$(a_2s^2 + a_1s + a_0)y(e^{st}) = (b_1s + b_0)x(e^{st})\ . \quad (6\text{-}8)$$

Although it's simpler than Eq. (6-6), we can further simplify the relationship in the last line in Eq. (6-8) by considering the ratio of $y(e^{st})$ over $x(e^{st})$ as the Laplace transfer function of our system in Figure 6-5. If we call that ratio of polynomials the transfer function $H(s)$,

$$H(s) = \frac{y(e^{st})}{x(e^{st})} = \frac{b_1 s + b_0}{a_2 s^2 + a_1 s + a_0} , \tag{6-9}$$

To indicate that the original $x(t)$ and $y(t)$ have the identical functional form of e^{st}, we can follow the standard Laplace notation of capital letters and show the transfer function as

$$H(s) = \frac{Y(s)}{X(s)} = \frac{b_1 s + b_0}{a_2 s^2 + a_1 s + a_0} , \tag{6-10}$$

where the output $Y(s)$ is given by

$$Y(s) = X(s)\frac{b_1 s + b_0}{a_2 s^2 + a_1 s + a_0} = X(s)H(s) . \tag{6-11}$$

Equation (6-11) leads us to redraw the original system diagram in a form that highlights the definition of the transfer function $H(s)$ as shown in Figure 6-6.

The cautious reader may be wondering, "Is it really valid to use this Laplace analysis technique when it's strictly based on the system's $x(t)$ input being some function of e^{st}, or $x(e^{st})$?" The answer is that the Laplace analysis technique, based on the complex exponential $x(e^{st})$, is valid because all practical $x(t)$ input functions can be represented with complex exponentials. For example,

- a constant, $c = ce^{0t}$,
- sinusoids, $\sin(\omega t) = (e^{j\omega t} - e^{-j\omega t})/2j$ or $\cos(\omega t) = (e^{j\omega t} + e^{-j\omega t})/2$,
- a monotonic exponential, e^{at}, and
- an exponentially varying sinusoid, $e^{-at}\cos(\omega t)$.

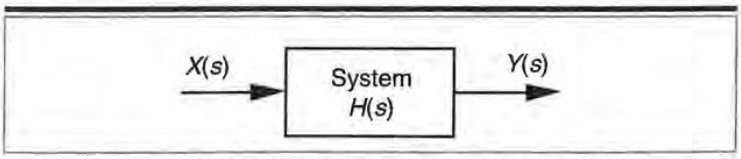


Figure 6-6 Linear system described by Eqs. (6-10) and (6-11). The system's input is the Laplace function $X(s)$, its output is the Laplace function $Y(s)$, and the system transfer function is $H(s)$.

With that said, if we know a system's transfer function $H(s)$, we can take the Laplace transform of any $x(t)$ input to determine $X(s)$, multiply that $X(s)$ by $H(s)$ to get $Y(s)$, and then inverse Laplace transform $Y(s)$ to yield the time-domain expression for the output $y(t)$. In practical situations, however, we usually don't go through all those analytical steps because it's the system's transfer function $H(s)$ in which we're most interested. Being able to express $H(s)$ mathematically or graph the surface $|H(s)|$ as a function of s will tell us the two most important properties we need to know about the system under analysis: Is the system stable, and if so, what is its frequency response?

"But wait a minute," you say. "Equations (6-10) and (6-11) indicate that we have to know the $Y(s)$ output before we can determine $H(s)$!" Not really. All we really need to know is the time-domain differential equation like that in Eq. (6-6). Next we take the Laplace transform of that differential equation and rearrange the terms to get the $H(s)$ ratio in the form of Eq. (6-10). With practice, systems designers can look at a diagram (block, circuit, mechanical, whatever) of their system and promptly write the Laplace expression for $H(s)$. Let's use the concept of the Laplace transfer function $H(s)$ to determine the stability and frequency response of simple continuous systems.

6.2.1 Poles and Zeros on the *s*-plane and Stability

One of the most important characteristics of any system involves the concept of stability. We can think of a system as stable if, given any bounded input, the output will always be bounded. This sounds like an easy condition to achieve because most systems we encounter in our daily lives are indeed stable. Nevertheless, we have all experienced instability in a system containing feedback. Recall the annoying *howl* when a public address system's microphone is placed too close to the loudspeaker. A sensational example of an unstable system occurred in western Washington when the first Tacoma Narrows Bridge began oscillating on the afternoon of November 7th, 1940. Those oscillations, caused by 42 mph winds, grew in amplitude until the bridge destroyed itself. For IIR digital filters with their built-in feedback, instability would result in a filter output that's not at all representative of the filter input; that is, our filter output samples would not be a filtered version of the input; they'd be some strange oscillating or pseudorandom values. A situation we'd like to avoid if we can, right? Let's see how.

We can determine a continuous system's stability by examining several different examples of $H(s)$ transfer functions associated with linear

time-invariant systems. Assume that we have a system whose Laplace transfer function is of the form of Eq. (6-10), the coefficients are all real, and the coefficients b_1 and a_2 are equal to zero. We'll call that Laplace transfer function $H_1(s)$, where

$$H_1(s) = \frac{b_0}{a_1 s + a_0} = \frac{b_0 / a_1}{s + a_0 / a_1} \quad . \tag{6-12}$$

Notice that if $s = -a_0/a_1$, the denominator in Eq. (6-12) equals zero and $H_1(s)$ would have an infinite magnitude. This $s = -a_0/a_1$ point on the s-plane is called a pole, and that pole's location is shown by the "x" in Figure 6-7(a). Notice that the pole is located exactly on the negative portion of the real σ axis. If the system described by H_1 were at rest and we disturbed it with an impulse like $x(t)$ input at time $t = 0$, its continuous time-domain $y(t)$ output would be the damped exponential curve shown in Figure 6-7(b). We can see that $H_1(s)$ is stable because its $y(t)$ output approaches zero as time passes. By the way, the distance of the pole from the $\sigma = 0$ axis, a_0/a_1 for our $H_1(s)$, gives the decay rate of the $y(t)$ impulse response. To illustrate why the term *pole* is appropriate, Figure 6-8(b) depicts the three-dimensional surface of $|H_1(s)|$ above the s-plane. Look at Figure 6-8(b) carefully and see how we've reoriented the s-plane axis. This new axis orientation allows us to see how the $H_1(s)$ system's frequency magnitude response can be determined from its three-dimensional s-plane surface. If we examine the $|H_1(s)|$ surface at $\sigma = 0$, we get the bold curve in Figure 6-8(b). That bold curve, the intersection of the vertical $\sigma = 0$ plane (the $j\omega$ axis plane) and the $|H_1(s)|$ surface, gives us the frequency magnitude response $|H_1(\omega)|$ of the system—and that's one of the things we're after here. The bold $|H_1(\omega)|$

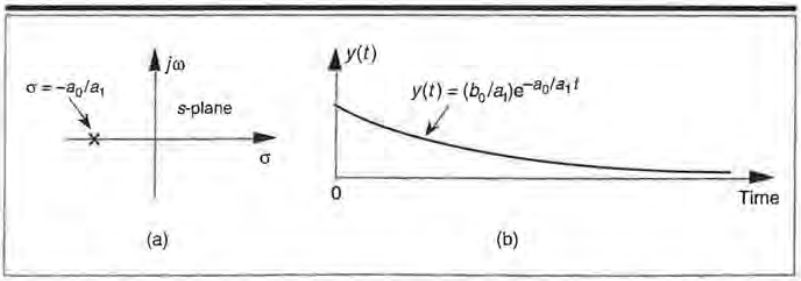


Figure 6-7 Descriptions of $H_1(s)$: (a) pole located at $s = \sigma + j\omega = -a_0/a_1 + j0$ on the s-plane; (b) time-domain $y(t)$ impulse response of the system.

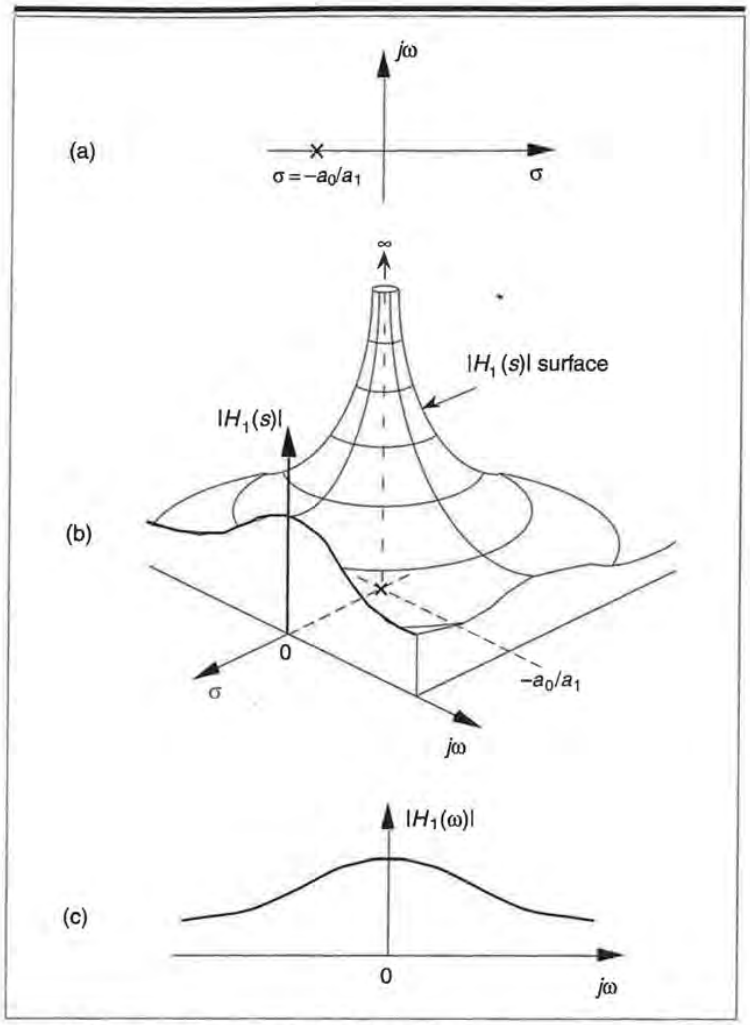


Figure 6-8 Further depictions of $H_1(s)$: (a) pole located at $\sigma = -a_0/a_1$ on the s-plane; (b) $|H_1(s)|$ surface; (c) curve showing the intersection of the $|H_1(s)|$ surface and the vertical $\sigma = 0$ plane. This is the conventional depiction of the $|H_1(\omega)|$ frequency magnitude response.

curve in Figure 6-8(b) is shown in a more conventional way in Figure 6-8(c). Figures 6-8(b) and 6-8(c) highlight the very important property that the Laplace transform is a more general case of the Fourier transform because if $\sigma = 0$, then $s = j\omega$. In this case, the $|H_1(s)|$ curve for

$\sigma = 0$ above the s-plane becomes the $|H_1(\omega)|$ curve above the $j\omega$ axis in Figure 6-8(c).

Another common system transfer function leads to an impulse response that oscillates. Let's think about an alternate system whose Laplace transfer function is of the form of Eq. (6-10), the coefficient b_0 equals zero, and the coefficients lead to complex terms when the denominator polynomial is factored. We'll call this particular second-order transfer function $H_2(s)$, where

$$H_2(s) = \frac{b_1 s}{a_2 s^2 + a_1 s + a_0} = \frac{(b_1 / a_2)s}{s^2 + (a_1 / a_2)s + a_0 / a_2} \quad . \tag{6-13}$$

(By the way, when a transfer function has the Laplace variable s in both the numerator and denominator, the *order* of the overall function is defined by the largest exponential order of s in the denominator polynomial. So our $H_2(s)$ is a second-order transfer function.) To keep the following equations from becoming too messy, let's factor its denominator and rewrite Eq. (6-13) as

$$H_2(s) = \frac{As}{(s+p)(s+p^*)} \quad . \tag{6-14}$$

where $A = b_1/a_2$, $p = p_{real} + jp_{imag}$, and $p^* = p_{real} - jp_{imag}$ (complex conjugate of p). Notice that, if s is equal to $-p$ or $-p^*$, one of the polynomial roots in the denominator of Eq. (6-14) will equal zero, and $H_2(s)$ will have an infinite magnitude. Those two complex poles, shown in Figure 6-9(a), are located off the negative portion of the real σ axis. If the H_2 system were at rest and we disturbed it with an impulselike $x(t)$ input at time $t = 0$, its continuous time-domain $y(t)$ output would be the damped sinusoidal curve shown in Figure 6-9(b). We see that $H_2(s)$ is stable because its oscillating $y(t)$ output, like a plucked guitar string, approaches zero as time increases. Again, the distance of the poles from the $\sigma = 0$ axis ($-p_{real}$) gives the decay rate of the sinusoidal $y(t)$ impulse response. Likewise, the distance of the poles from the $j\omega = 0$ axis ($\pm p_{imag}$) gives the frequency of the sinusoidal $y(t)$ impulse response. Notice something new in Figure 6-9(a). When $s = 0$, the numerator of Eq. (6-14) is zero, making the transfer function $H_2(s)$ equal to zero. Any value of s where $H_2(s) = 0$ is sometimes of interest and is usually plotted on the s-plane as the little circle, called a "zero," shown in Figure 6-9(a). At this point we're not very interested in knowing exactly what p and p^* are in

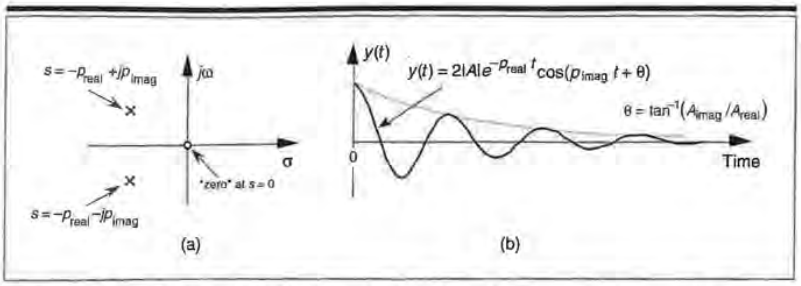


Figure 6-9 Descriptions of $H_2(s)$: (a) poles located at $s = p_{real} \pm jp_{imag}$ on the s-plane; (b) time domain $y(t)$ impulse response of the system.

terms of the coefficients in the denominator of Eq. (6-13). However, an energetic reader could determine the values of p and p^* in terms of a_0, a_1, and a_2 by using the following well-known quadratic factorization formula: Given the second-order polynomial $f(s) = as^2 + bs + c$, then $f(s)$ can be factored as

$$f(s) = as^2 + bs + c = \left(s + \frac{b}{2a} + \sqrt{\frac{b^2 - 4ac}{4a^2}}\right) \cdot \left(s + \frac{b}{2a} - \sqrt{\frac{b^2 - 4ac}{4a^2}}\right) . \quad (6\text{-}15)$$

Figure 6-10(b) illustrates the $|H_2(s)|$ surface above the s-plane. Again, the bold $|H_2(\omega)|$ curve in Figure 6-10(b) is shown in the conventional way in Figure 6-10(c) to indicate the frequency magnitude response of the system described by Eq. (6-13). Although the three-dimensional surfaces in Figures 6-8(b) and 6-10(b) are informative, they're also unwieldy and unnecessary. We can determine a system's stability merely by looking at the locations of the poles on the two-dimensional s-plane.

To further illustrate the concept of system stability, Figure 6-11 shows the s-plane pole locations of several example Laplace transfer functions and their corresponding time-domain impulse responses. We recognize Figures 6-11(a) and 6-11(b), from our previous discussion, as indicative of stable systems. When disturbed from their at-rest condition they respond and, at some later time, return to that initial condition. The single pole location at $s = 0$ in Figure 6-11(c) is indicative of the $1/s$ transfer function of a single element of a linear system. In an electrical system, this $1/s$ transfer function could be a capacitor that was charged with an impulse of current, and there's no discharge path in the circuit.

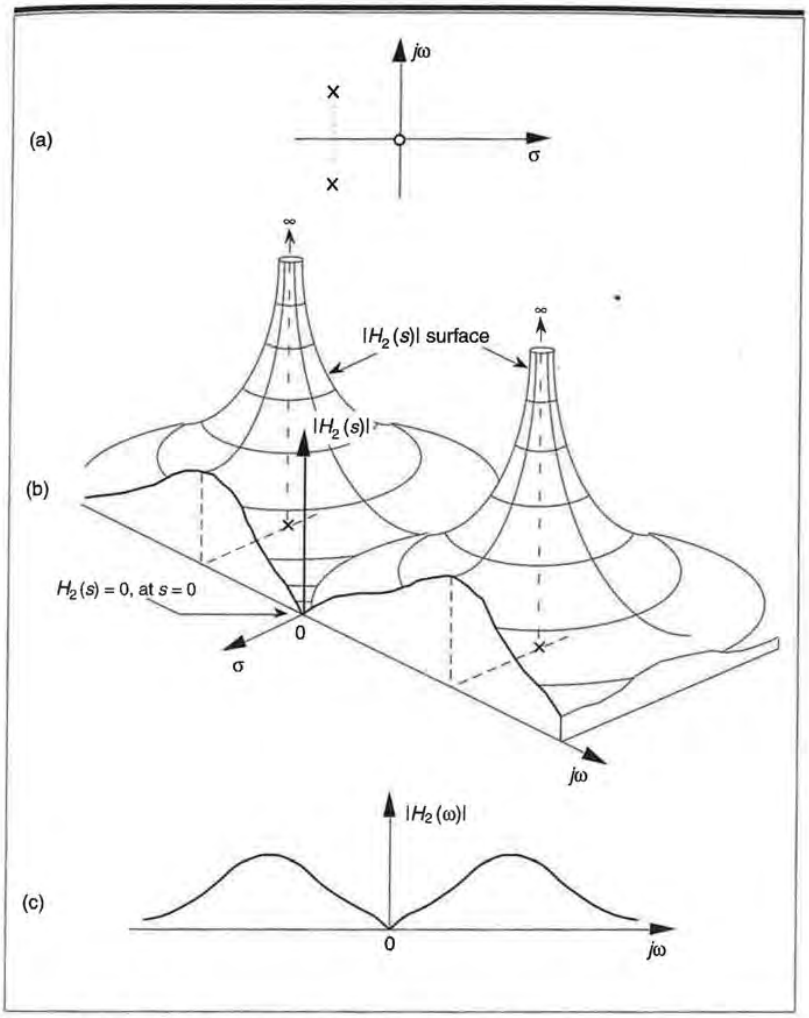


Figure 6-10 Further depictions of $H_2(s)$: (a) poles and zero locations on the s-plane; (b) $|H_2(s)|$ surface; (c) $|H_2(\omega)|$ frequency magnitude response curve.

For a mechanical system, Figure 6-11(c) would describe a kind of spring that's compressed with an impulse of force and, for some reason, remains under compression. Notice, in Figure 6-11(d), that, if an $H(s)$ transfer function has conjugate poles located exactly on the $j\omega$ axis

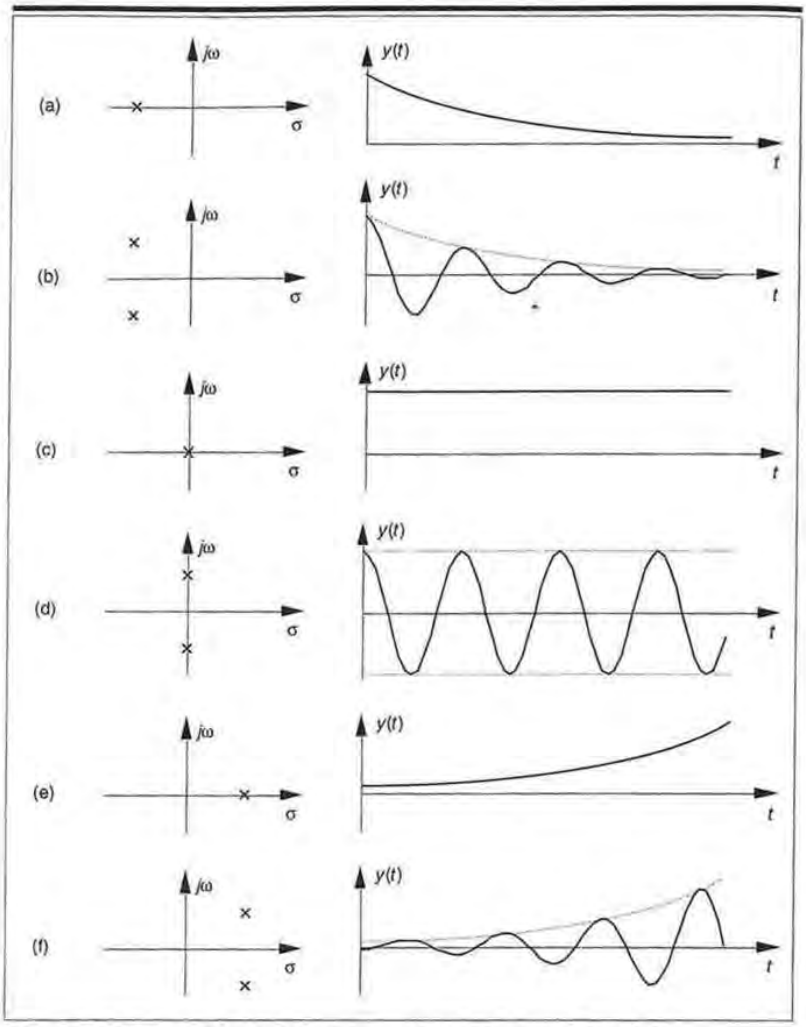


Figure 6-11 Various $H(s)$ pole locations and their time-domain impulse responses: (a) single pole at $\sigma < 0$; (b) conjugate poles at $\sigma < 0$; (c) single pole located at $\sigma = 0$; (d) conjugate poles located at $\sigma = 0$; (e) single pole at $\sigma > 0$; (f) conjugate poles at $\sigma > 0$.

($\sigma = 0$), the system will go into oscillation when disturbed from its initial condition. This situation, called conditional stability, happens to describe the intentional transfer function of electronic oscillators. Instability is indicated in Figures 6-11(e) and 6-11(f). Here, the poles lie to the right of the $j\omega$ axis. When disturbed from their initial at-rest condition

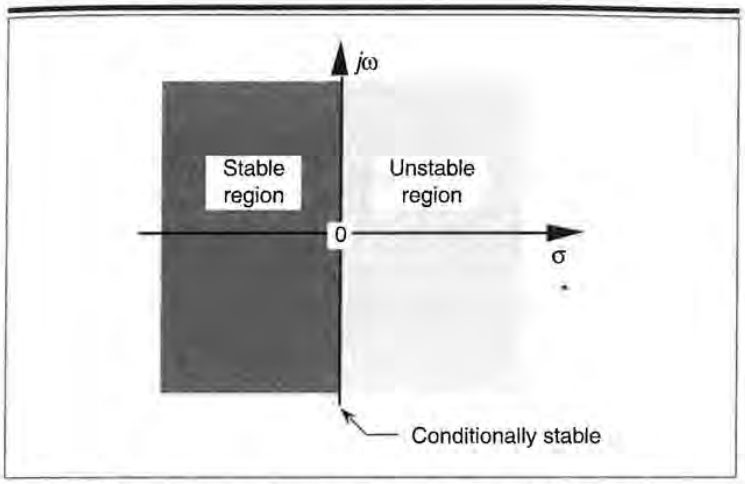


Figure 6-12 The Laplace *s*-plane showing the regions of stability and instability for pole locations for linear continuous systems.

by an impulse input, their outputs grow without bound.† See how the value of σ, the real part of s, for the pole locations is the key here? When $\sigma < 0$, the system is well behaved and stable; when $\sigma = 0$, the system is conditionally stable; and when $\sigma > 0$ the system is unstable. So we can say that, when σ is located on the right half of the s-plane, the system is unstable. We show this characteristic of linear continuous systems in Figure 6-12. Keep in mind that real-world systems often have more than two poles, and a system is only as stable as its least stable pole. For a system to be stable, all of its transfer-function poles must lie on the left half of the s-plane.

To consolidate what we've learned so far: $H(s)$ is determined by writing a linear system's time-domain differential equation and taking the Laplace transform of that equation to obtain a Laplace expression in terms of $X(s)$, $Y(s)$, s, and the system's coefficients. Next we rearrange the Laplace expression terms to get the $H(s)$ ratio in the form of Eq. (6-10). (The really slick part is that we do not have to know what the time-domain $x(t)$ input

† Impulse response testing in a laboratory can be an important part of the system design process. The difficult part is generating a true impulselike input. If the system is electrical, for example, although somewhat difficult to implement, the input $x(t)$ impulse would be a very short duration voltage or current pulse. If, however, the system were mechanical, a whack with a hammer would suffice as an $x(t)$ impulse input. For digital systems, on the other hand, an impulse input is easy to generate; it's a single unity-valued sample preceded and followed by all zero-valued samples.

is to analyze a linear system!) We can get the expression for the continuous frequency response of a system just by substituting $j\omega$ for s in the $H(s)$ equation. To determine system stability, the denominator polynomial of $H(s)$ is factored to find each of its roots. Each root is set equal to zero and solved for s to find the location of the system poles on the s-plane. Any pole located to the right of the $j\omega$ axis on the s-plane will indicate an unstable system.

OK, returning to our original goal of understanding the z-transform, the process of analyzing IIR filter systems requires us to replace the Laplace transform with the z-transform and to replace the s-plane with a z-plane. Let's introduce the z-transform, determine what this new z-plane is, discuss the stability of IIR filters, and design and analyze a few simple IIR filters.

6.3 The *z*-Transform

The z-transform is the discrete-time cousin of the continuous Laplace transform.[†] While the Laplace transform is used to simplify the analysis of continuous differential equations, the z-transform facilitates the analysis of discrete difference equations. Let's define the z-transform and explore its important characteristics to see how it's used in analyzing IIR digital filters.

The z-transform of a discrete sequence $h(n)$, expressed as $H(z)$, is defined as

$$H(z) = \sum_{n=-\infty}^{\infty} h(n)z^{-n} , \tag{6-16}$$

where the variable z is complex. Where Eq. (6-3) allowed us to take the Laplace transform of a continuous signal, the z-transform is performed on a discrete $h(n)$ sequence, converting that sequence into a continuous function $H(z)$ of the continuous complex variable z. Similarly, as the function e^{-st} is the general form for the solution of linear differential equations, z^{-n} is the general form for the solution of linear difference equations. Moreover, as a Laplace function $F(s)$ is a continuous surface

[†] In the early 1960s, James Kaiser, after whom the Kaiser window function is named, consolidated the theory of digital filters using a mathematical description known as the z-transform[4,5]. Until that time, the use of the z-transform had generally been restricted to the field of discrete control systems[6–9].

above the s-plane, the z-transform function $H(z)$ is a continuous surface above a z-plane. To whet your appetite, we'll now state that, if $H(z)$ represents an IIR filter's z-domain transfer function, evaluating the $H(z)$ surface will give us the filter's frequency magnitude response, and $H(z)$'s pole and zero locations will determine the stability of the filter.

We can determine the frequency response of an IIR digital filter by expressing z in polar form as $z = re^{j\omega}$, where r is a magnitude and ω is the angle. In this form, the z-transform equation becomes

$$H(z) = H(re^{j\omega}) = \sum_{n=-\infty}^{\infty} h(n)(re^{j\omega})^{-n} = \sum_{n=-\infty}^{\infty} h(n)r^{-n}(e^{-j\omega n}) \quad . \qquad (6\text{-}17)$$

Equation (6-17) can be interpreted as the Fourier transform of the product of the original sequence $h(n)$ and the exponential sequence r^{-n}. When r equals one, Eq. (6-17) simplifies to the Fourier transform. Thus on the z-plane, the contour of the $H(z)$ surface for those values where $|z| = 1$ is the Fourier transform of $h(n)$. If $h(n)$ represents a filter impulse response sequence, evaluating the $H(z)$ transfer function for $|z| = 1$ yields the frequency response of the filter. So where on the z-plane is $|z| = 1$? It's a circle with a radius of one, centered about the $z = 0$ point. This circle, so important that it's been given the name *unit circle*, is shown in Figure 6-13. Recall that the $j\omega$ frequency axis on the continuous Laplace s-plane was linear and ranged from $-\infty$ to $+\infty$ radians/s. The ω frequency axis on the complex z-plane, however, spans only the range from $-\pi$ to $+\pi$ radians. With this relationship between the $j\omega$ axis on the Laplace s-plane and the unit circle on the z-plane, we can see that the z-plane frequency axis is

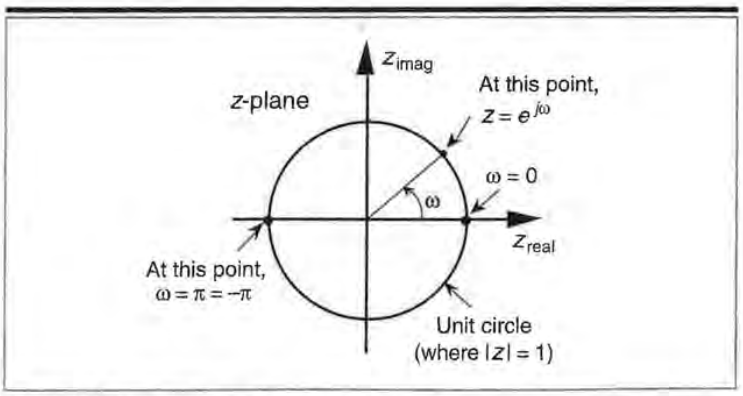


Figure 6-13 Unit circle on the complex z-plane.

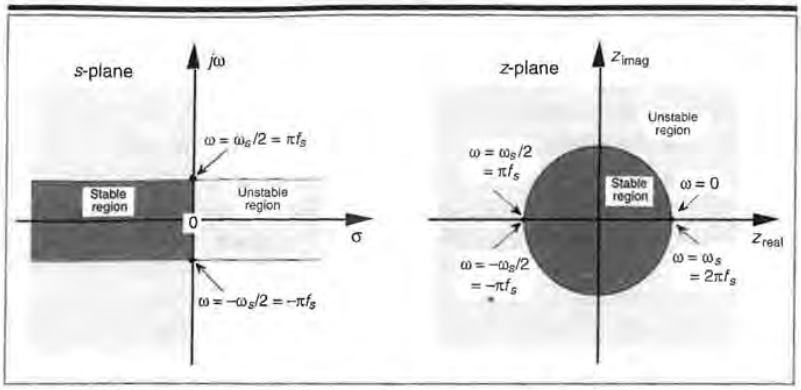


Figure 6-14 Mapping of the Laplace *s*-plane to the *z*-plane. All frequency values are in radians/s.

equivalent to coiling the *s*-plane's jω axis about the unit circle on the *z*-plane as shown in Figure 6-14.

Then, frequency ω on the *z*-plane, is not a distance along a straight line, but rather an angle around a circle. With ω in Figure 6-13 being a general normalized angle in radians ranging from $-\pi$ to $+\pi$, we can relate ω to an equivalent f_s sampling rate by defining a new frequency variable $\omega_s = 2\pi f_s$ in radians/s. The periodicity of discrete frequency representations, with a period of $\omega_s = 2\pi f_s$ radians/s or f_s Hz, is indicated for the *z*-plane in Figure 6-14. Where a walk along the *j*ω frequency axis on the *s*-plane could take us to infinity in either direction, a trip on the ω frequency path on the *z*-plane leads us in circles (on the unit circle). Figure 6-14 shows us that only the $-\pi f_s$ to $+\pi f_s$ radians/s frequency range for ω can be accounted for on the *z*-plane, and this is another example of the universal periodicity of the discrete frequency domain. (Of course the $-\pi f_s$ to $+\pi f_s$ radians/s range corresponds to a cyclic frequency range of $-f_s/2$ to $+f_s/2$.) With the perimeter of the unit circle being $z = e^{j\omega}$, later, we'll show exactly how to substitute $e^{j\omega}$ for *z* in a filter's *H*(*z*) transfer function, giving us the filter's frequency response.

6.3.1 Poles and Zeros on the *z*-plane and Stability

One of the most important characteristics of the *z*-plane is that the region of filter stability is mapped to the inside of the unit circle on the *z*-plane. Given the *H*(*z*) transfer function of a digital filter, we can examine that function's pole locations to determine filter stability. If all poles are located inside the unit circle, the filter will be stable. On the other hand, if

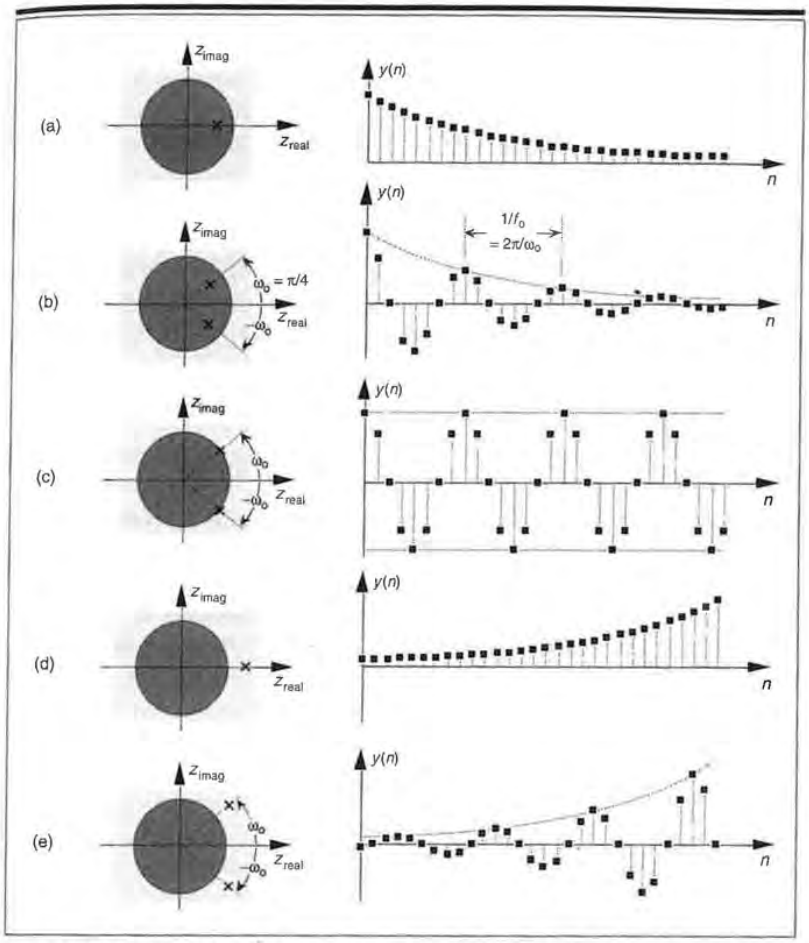


Figure 6-15 Various $H(z)$ pole locations and their discrete time-domain impulse responses: (a) single pole inside the unit circle; (b) conjugate poles located inside the unit circle; (c) conjugate poles located on the unit circle; (d) single pole outside the unit circle; (e) conjugate poles located outside the unit circle.

any pole is located outside the unit circle, the filter will be unstable. Figure 6-15 shows the z-plane pole locations of several example z-domain transfer functions and their corresponding discrete time-domain impulse responses. It's a good idea for the reader to compare the z-plane and discrete-time responses of Figure 6-15 with the s-plane and the continuous time responses of Figure 6-11. The $y(n)$ outputs in Figures 6-15(d) and

(e) show examples of how unstable filter outputs increase in amplitude, as time passes, whenever their $x(n)$ inputs are nonzero. To avoid this situation, any IIR digital filter that we design should have an $H(z)$ transfer function with all of its individual poles inside the unit circle. Like a chain that's only as strong as its weakest link, an IIR filter is only as stable as its least stable pole.

The ω_o oscillation frequency of the impulse responses in Figures 6-15(c) and (e) is, of course, proportional to the angle of the conjugate pole pairs from the z_{real} axis, or ω_o radians/s corresponding to $f_o = \omega_o/2\pi$ Hz. Because the intersection of the $-z_{real}$ axis and the unit circle, point $z = -1$, corresponds to π radians (or πf_s radians/s $= f_s/2$ Hz), the ω_o angle of $\pi/4$ in Figure 6-15 means that $f_o = f_s/8$ and our $y(n)$ will have eight samples per cycle of f_o.

6.3.2 Using the *z*-transform to Analyze IIR Filters

We have one last concept to consider before we can add the z-transform to our collection of digital signal processing tools. We need to determine what the time delay operation in Figure 6-3 is relative to the z-transform. To do this, assume we have a sequence $x(n)$ whose z-transform is $X(z)$ and a sequence $y(n) = x(n-1)$ whose z-transform is $Y(z)$ as shown in Figure 6-16. The z-transform of $y(n)$ is, by definition,

$$Y(z) = \sum_{n=-\infty}^{\infty} y(n)z^{-n} = \sum_{n=-\infty}^{\infty} x(n-1)z^{-n} \quad . \tag{6-18}$$

Now if we let $k = n-1$, then, $Y(z)$ becomes

$$Y(z) = \sum_{k=-\infty}^{\infty} x(k)z^{-(k+1)} = \sum_{k=-\infty}^{\infty} x(k)z^{-k}z^{-1} \quad , \tag{6-19}$$

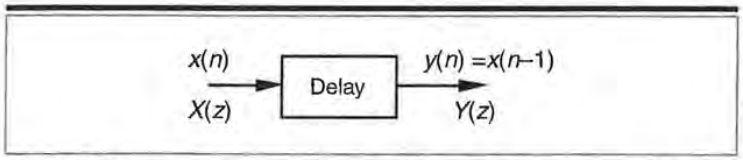


Figure 6-16 Output sequence $y(n)$ equal to a *unit delayed* version of the input $x(n)$ sequence.

which we can write as

$$Y(z) = z^{-1} \sum_{k=-\infty}^{\infty} x(k)z^{(-k)} = z^{-1}[X(z)] \ . \qquad (6\text{-}20)$$

Thus, the effect of a single unit of time delay is to multiply the z-transform by z^{-1}.

Interpreting a unit time delay to be equivalent to the z^{-1} operator leads us to the relationship shown in Figure 6-17, where we can say $X(z)z^0 = X(z)$ is the z-transform of $x(n)$, $X(z)z^{-1}$ is the z-transform of $x(n)$ delayed by one sample, $X(z)z^{-2}$ is the z-transform of $x(n)$ delayed by two samples, and $X(z)z^{-k}$ is the z-transform of $x(n)$ delayed by k samples. So a transfer function of z^{-k} is equivalent to a delay of kt_s seconds from the instant when $t = 0$, where t_s is the period between data samples, or one over the sample rate. Specifically, $t_s = 1/f_s$. Because a delay of one sample is equivalent to the factor z^{-1}, the unit time delay symbol used in Figures 6-2 and 6-3 is usually indicated by the z^{-1} operator.

Let's pause for a moment and consider where we stand so far. Our acquaintance with the Laplace transform with its s-plane, the concept of stability based on $H(s)$ pole locations, the introduction of the z-transform with its z-plane poles, and the concept of the z^{-1} operator signifying a single unit of time delay has led us to our goal: the ability to inspect an IIR filter difference equation or filter structure and immediately write the filter's z-domain transfer function $H(z)$. Accordingly, by evaluating an IIR filter's $H(z)$ transfer function appropriately, we can determine the filter's frequency response and its stability. With those ambitious thoughts in mind, let's develop the z-domain equations we need to analyze IIR filters.

Using the relationships of Figure 6-17, we redraw Figure 6-3 as a general Mth-order IIR filter using the z^{-1} operator as shown in Figure 6-18. (In hardware, those z^{-1} operations are shift registers holding successive filter

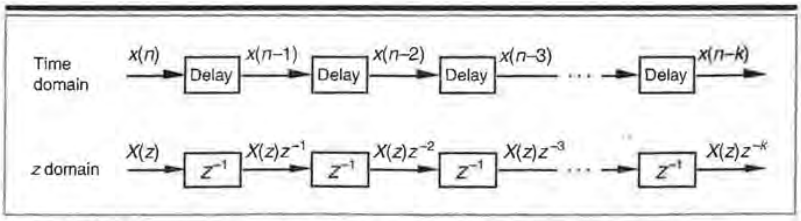


Figure 6-17 Correspondence of the delay operation in the time domain and the z^{-k} operation in the z-domain.

input and output sample values. When implementing an IIR filter in a software routine, the z^{-1} operation merely indicates sequential memory locations where input and output sequences are stored.) The IIR filter structure in Figure 6-18 is often called the *Direct Form I* structure.

The time-domain difference equation describing the general Mth-order IIR filter, having N feed forward stages and M feedback stages, in Figure 6-18 is

$$y(n) = b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + \ldots + b(N)x(n-N) + a(1)y(n-1) + a(2)y(n-2) + \ldots + a(M)y(n-M) \quad . \tag{6-21}$$

Time domain expression for an *M*th-order IIR filter

In the z-domain, that IIR filter's output can be expressed by

$$Y(z) = b(0)X(z) + b(1)X(z)z^{-1} + b(2)X(z)z^{-2} + \ldots + b(N)X(z)z^{-N} + a(1)Y(z)z^{-1} + a(2)Y(z)z^{-2} + \ldots + a(M)Y(z)z^{-M} \quad , \tag{6-22}$$

where $Y(z)$ and $X(z)$ represent the z-transform of $y(n)$ and $x(n)$. Look Eqs. (6-21) and (6-22) over carefully and see how the unit time delays translate

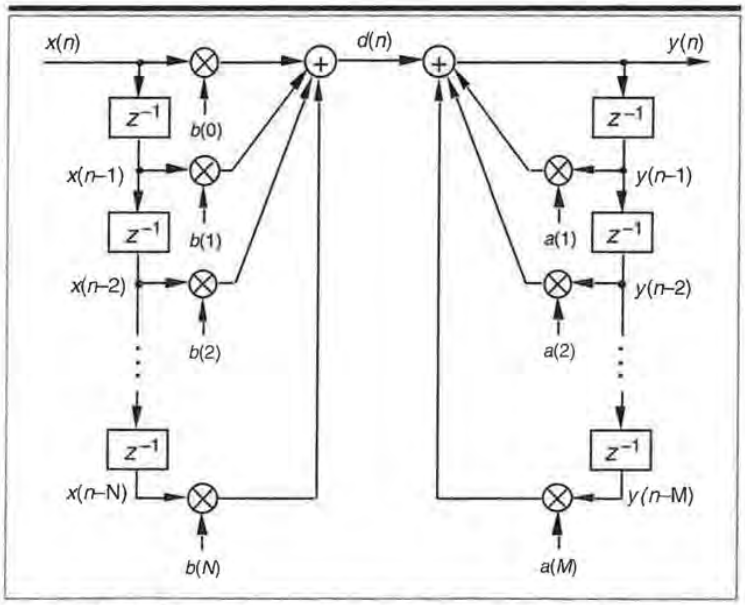


Figure 6-18 General (Direct Form I) structure of an Mth-order IIR filter, having N feed forward stages and M feedback stages, with the z^{-1} operator indicating a unit time delay.

to negative powers of z in the z-domain expression. A more compact form for $Y(z)$ is

z-domain expression for an Mth-order IIR filter: →

$$Y(z) = X(z)\sum_{k=0}^{N} b(k)z^{-k} + Y(z)\sum_{k=1}^{M} a(k)z^{-k} \quad . \quad (6\text{-}23)$$

OK, now we've arrived at the point where we can describe the transfer function of a general IIR filter. Rearranging Eq. (6-23) to collect like terms,

$$Y(z)\left[1 - \sum_{k=1}^{M} a(k)z^{-k}\right] = X(z)\sum_{k=0}^{N} b(k)z^{-k} \quad . \quad (6\text{-}24)$$

Finally, we define the filter's z-domain transfer function as $H(z) = Y(z)/X(z)$, where $H(z)$ is given by

z-domain transfer function of an Mth-order IIR filter: →

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} b(k)z^{-k}}{1 - \sum_{k=1}^{M} a(k)z^{-k}} . \quad (6\text{-}25)$$

(Just like Laplace transfer functions, the order of our z-domain transfer function is defined by the largest exponential order of z in the denominator, in this case M.) Equation (6-25) tells us all we need to know about an IIR filter. We can evaluate the denominator of Eq. (6-25) to determine the positions of the filter's poles on the z-plane indicating the filter's stability.

Remember, now, just as the Laplace transfer function $H(s)$ in Eq. (6-9) was a complex-valued surface on the s-plane, $H(z)$ is a complex-valued surface above, or below, the z-plane. The intersection of that $H(z)$ surface and the perimeter of a cylinder representing the $z = e^{j\omega}$ unit circle is the filter's complex frequency response. This means that substituting $e^{j\omega}$ for z in Eq. (6-25)'s transfer function gives us the expression for the filter's $H_{IIR}(\omega)$ frequency response as

Frequency response of an Mth-order IIR filter filter (exponential form): →

$$H_{IIR}(\omega) = H(z)\Big|_{z=e^{j\omega}} = \frac{\sum_{k=0}^{N} b(k)e^{-jk\omega}}{1 - \sum_{k=1}^{M} a(k)e^{-jk\omega}} \quad . \quad (6\text{-}26)$$

Let's alter the form of Eq. (6-26) to obtain more useful expressions for $H_{IIR}(\omega)$'s frequency magnitude and phase responses. Because a typical IIR filter frequency response $H_{IIR}(\omega)$ is the ratio of two complex functions, we can express $H_{IIR}(\omega)$ in its equivalent rectangular form as

$$H_{IIR}(\omega) = \frac{\sum_{k=0}^{N} b(k) \cdot [\cos(k\omega) - j\sin(k\omega)]}{1 - \sum_{k=1}^{M} a(k) \cdot [\cos(k\omega) - j\sin(k\omega)]}, \quad (6\text{-}27)$$

or

Frequency response of an *M*th-order IIR filter (rectangular form): →

$$H_{IIR}(\omega) = \frac{\sum_{k=0}^{N} b(k) \cdot \cos(k\omega) - j\sum_{k=0}^{N} b(k) \cdot \sin(k\omega)}{1 - \sum_{k=1}^{M} a(k) \cdot \cos(k\omega) + j\sum_{k=1}^{M} a(k) \cdot \sin(k\omega)} \quad (6\text{-}28)$$

It's usually easier and, certainly, more useful, to consider the complex frequency response expression in terms of its magnitude and phase. Let's do this by representing the numerator and denominator in Eq. (6-28) as two complex functions of radian frequency ω. Calling the numerator of Eq. (6-28) $Num(\omega)$, then,

$$Num(\omega) = Num_{real}(\omega) + jNum_{imag}(\omega), \quad (6\text{-}29)$$

where

$$Num_{real}(\omega) = \sum_{k=0}^{N} b(k) \cdot \cos(k\omega),$$

and

$$Num_{imag}(\omega) = -\sum_{k=0}^{N} b(k) \cdot \sin(k\omega). \quad (6\text{-}30)$$

Likewise, the denominator in Eq. (6-28) can be expressed as

$$Den(\omega) = Den_{real}(\omega) + jDen_{imag}(\omega), \quad (6\text{-}31)$$

where

$$Den_{\text{real}}(\omega) = 1 - \sum_{k=1}^{M} a(k) \cdot \cos(k\omega) \ ,$$

and

$$Den_{\text{imag}}(\omega) = \sum_{k=1}^{M} a(k) \cdot \sin(k\omega) \ . \qquad (6\text{-}32)$$

These $Num(\omega)$ and $Den(\omega)$ definitions allow us to represent $H_{\text{IIR}}(\omega)$ in the more simple forms of

$$H_{\text{IIR}}(\omega) = \frac{Num(\omega)}{Den(\omega)} = \frac{Num_{\text{real}}(\omega) + jNum_{\text{imag}}(\omega)}{Den_{\text{real}}(\omega) + jDen_{\text{imag}}(\omega)} \qquad (6\text{-}33)$$

$$= \frac{|Num(\omega)| \angle \varnothing_{Num}(\omega)}{|Den(\omega)| \angle \varnothing_{Den}(\omega)} \ . \qquad (6\text{-}33')$$

Given the form in Eq. (6-33) and the rules for dividing one complex number by another, provided by Eqs. (A-2) and (A-19') in Appendix A, the frequency magnitude response of a general IIR filter is

$$|H_{\text{IIR}}(\omega)| = \frac{|Num(\omega)|}{|Den(\omega)|} = \frac{\sqrt{[Num_{\text{real}}(\omega)]^2 + [Num_{\text{imag}}(\omega)]^2}}{\sqrt{[Den_{\text{real}}(\omega)]^2 + [Den_{\text{imag}}(\omega)]^2}} \ . \qquad (6\text{-}34)$$

Furthermore, the filter's phase response $\varnothing_{\text{IIR}}(\omega)$ is the phase of the numerator minus the phase of the denominator, or

$$\varnothing_{\text{IIR}}(\omega) = \varnothing_{Num}(\omega) - \varnothing_{Den}(\omega)$$

$$= \tan^{-1}\left(\frac{Num_{\text{imag}}(\omega)}{Num_{\text{real}}(\omega)}\right) - \tan^{-1}\left(\frac{Den_{\text{imag}}(\omega)}{Den_{\text{real}}(\omega)}\right) \ . \qquad (6\text{-}35)$$

To reiterate our intent here, we've gone through the above algebraic gymnastics to develop expressions for an IIR filter's frequency magnitude response $|H_{IIR}(\omega)|$ and phase response $\varnothing_{IIR}(\omega)$ in terms of the filter coefficients in Eqs. (6-30) and (6-32). Shortly, we'll use these expressions to analyze an actual IIR filter.

Pausing a moment to gather our thoughts, we realize that we can use Eqs. (6-34) and (6-35) to compute the magnitude and phase response of IIR filters as a function of the frequency ω. And again, just what is ω? It's the normalized radian frequency represented by the angle around the unit circle in Figure 6-13, having a range of $-\pi \le \omega \le +\pi$. In terms of a discrete sampling frequency ω_s measured in radians/s, from Figure 6-14, we see that ω covers the range $-\omega_s/2 \le \omega \le +\omega_s/2$. In terms of our old friend f_s Hz, Eqs. (6-34) and (6-35) apply over the equivalent frequency range of $-f_s/2$ to $+f_s/2$ Hz. So, for example, if digital data is arriving at the filter's input at a rate of $f_s = 1000$ samples/s, we could use Eq. (6-34) to plot the filter's frequency magnitude response over the frequency range of –500 Hz to +500 Hz.

Although the equations describing the transfer function $H_{IIR}(\omega)$, its magnitude response $|H_{IIR}(\omega)|$, and phase response $\varnothing_{IIR}(\omega)$ look somewhat complicated at first glance, let's illustrate their simplicity and utility by analyzing the simple second-order low-pass IIR filter in Figure 6-19 whose positive cutoff frequency is $\omega_s/10$. By inspection, we can write the filter's time-domain difference equation as

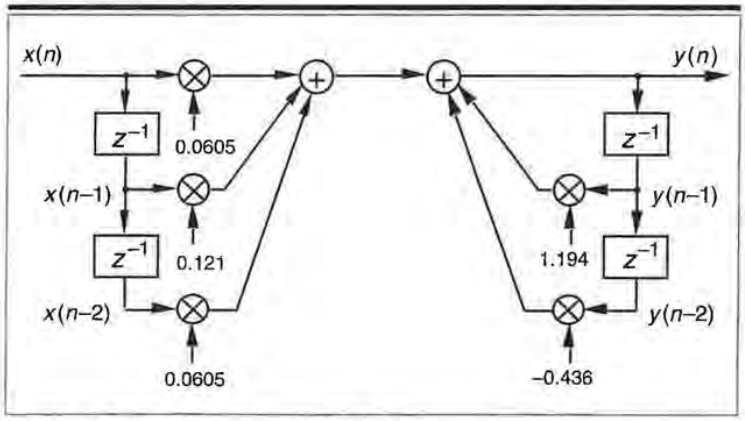


Figure 6-19 Second-order low-pass IIR filter example.

$$y(n) = 0.0605 \cdot x(n) + 0.121 \cdot x(n-1) + 0.0605 \cdot x(n-2) + 1.194 \cdot y(n-1) - 0.436 \cdot y(n-2) \ , \quad (6\text{-}36)$$

or the z-domain expression as

$$Y(z) = 0.0605 \cdot X(z) + 0.121 \cdot X(z)z^{-1} + 0.0605 \cdot X(z)z^{-2} + 1.194 \cdot Y(z)z^{-1} - 0.436 \cdot Y(z)z^{-2} \ . \quad (6\text{-}37)$$

Using Eq. (6-25), we write the z-domain transfer function $H(z)$ as

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.0605 \cdot z^0 + 0.121 \cdot z^{-1} + 0.0605 \cdot z^{-2}}{1 - 1.194 \cdot z^{-1} + 0.436 \cdot z^{-2}} \ . \quad (6\text{-}38)$$

Replacing z with $e^{j\omega}$, we see that the frequency response of our example IIR filter is

$$H_{IIR}(\omega) = \frac{0.0605 \cdot e^{-j0\omega} + 0.121 \cdot e^{-j1\omega} + 0.0605 \cdot e^{-j2\omega}}{1 - 1.194 \cdot e^{-j1\omega} + 0.436 \cdot e^{-j2\omega}} \ . \quad (6\text{-}39)$$

We're almost there. Remembering Euler's equations and that cos(0) = 1 and sin(0) = 0, we can write the rectangular form of $H_{IIR}(\omega)$ as

$$H_{IIR}(\omega) = \frac{0.0605 + 0.121 \cdot \cos(1\omega) + 0.0605 \cdot \cos(2\omega) - j[0.121 \cdot \sin(1\omega) + 0.0605 \cdot \sin(2\omega)]}{1 - 1.194 \cdot \cos(1\omega) + 0.436 \cdot \cos(2\omega) + j[1.194 \cdot \sin(1\omega) - 0.436 \cdot \sin(2\omega)]} \ . \quad (6\text{-}40)$$

Equation (6-40) is what we're after here, and if we calculate its magnitude over the frequency range of $-\pi \le \omega \le \pi$, we get the $|H_{IIR}(\omega)|$ shown as the solid curve in Figure 6-20(a). For comparison purposes we also show a 5-tap low-pass FIR filter magnitude response in Figure 6-20(a). Although both filters require the same computational workload, five multiplications per filter output sample, the low-pass IIR filter has the superior frequency magnitude response. Notice the steeper roll-off and lower sidelobes of the IIR filter relative to the FIR filter.†

† To make this IIR and FIR filter comparison valid, the coefficients used for both filters were chosen so that each filter would approximate the ideal low-pass frequency response shown in Figure 5-17(a).

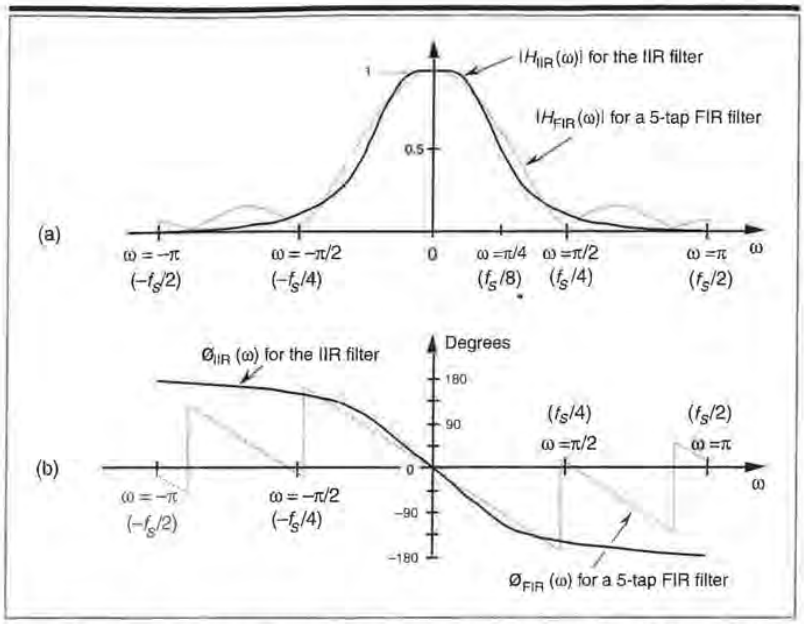


Figure 6-20 Frequency responses of the example IIR filter (solid line) in Figure 6-19 and a 5-tap FIR filter (dashed line): (a) magnitude responses; (b) phase responses.

A word of warning here. It's easy to reverse some of the signs for the terms in the denominator of Eq. (6-40), so be careful if you attempt these calculations at home. Some authors avoid this problem by showing the $a(k)$ coefficients in Figure 6-18 as negative values, so that the summation in the denominator of Eq. (6-25) is always positive. Moreover, some commercial software IIR design routines provide $a(k)$ coefficients whose signs must be reversed before they can be applied to the IIR structure in Figure 6-18. (If, while using software routines to design or analyze IIR filters, your results are very strange or unexpected, the first thing to do is reverse the signs of the $a(k)$ coefficients and see if that doesn't solve the problem.)

The solid curve in Figure 6-20(b) is our IIR filter's $\varnothing_{IIR}(\omega)$ phase response. Notice its nonlinearity relative to the FIR filter's phase response. (Remember now, we're only interested in the filter phase responses over the low-pass filters' passband. So those phase discontinuities for the FIR filter are of no consequence.) Phase nonlinearity is inherent in IIR filters and, based on the ill effects of nonlinear phase introduced in the group delay discussion of Section 5.8, we must carefully consider its implications whenever we decide to use an IIR filter instead of an FIR

filter in any given application. The question any filter designer must ask and answer, is "How much phase distortion can I tolerate to realize the benefits of the reduced computational load and high data rates afforded by IIR filters?"

We've arrived at an appropriate point to illustrate an important relationship between IIR and FIR filters. We can use our IIR analysis equations to characterize FIR filters. Notice that, if there were no feedback coefficients in Figure 6-18, equivalent to setting $a(1)$ through $a(M)$ in Eq. (6-21) all to zero, we'd have an N-tap FIR filter. This simplifies Eqs. (6-25) through (6-28) because their denominators would be equal to one. In addition, $Den(\omega)$ in Eq. (6-33) and $|Den(\omega)|$ in Eq. (6-34) would both be unity, and $\varnothing_{Den}(\omega)$ would be zero in Eq. (6-35). Thus, for an FIR filter, with the $a(k)$ coefficients in Figure 6-18 being all zeros, we can rewrite Eqs. (6-26) and (6-28) as

Frequency response of an N-stage FIR filter (exponential form): →

$$H_{FIR}(\omega) = \sum_{k=0}^{N} b(k)e^{-jk\omega}, \qquad (6\text{-}41)$$

and

Frequency response of an N-stage FIR filter (rectangular form): →

$$H_{FIR}(\omega) = \sum_{k=0}^{N} b(k) \cdot \cos(k\omega) - j\sum_{k=0}^{N} b(k) \cdot \sin(k\omega) \ . \qquad (6\text{-}42)$$

Just as in Eqs. (6-34) and (6-35), the ω in Eqs. (6-41) and (6-42), is the normalized radian frequency represented by the angle around the unit circle in Figure 6-13, having a range of $-\pi \le \omega \le +\pi$, corresponding to the equivalent frequency range of $-f_s/2$ to $+f_s/2$ Hz.

6.3.3 An Improved IIR Filter Structure

The structure of the IIR filter in Figure 6-18 can be converted to a more efficient and popular form. It's easy to explore this idea by assuming that there are an equal number of feed forward and feedback stages, letting $M = N$ in Figure 6-18 and thinking of the feed forward and feedback stages as two separate filters. Specifically, the $b(k)$ coefficients are used to calculate the sequence $d(n)$ from $x(n)$, and the $a(k)$ coefficients are used to calculate the filter output $y(n)$ sequence from $d(n)$. Because the two halves of the filter are both linear, we can swap them, as shown in Figure 6-21(a), with no change in the final $y(n)$ output. Of course, after this reorientation,

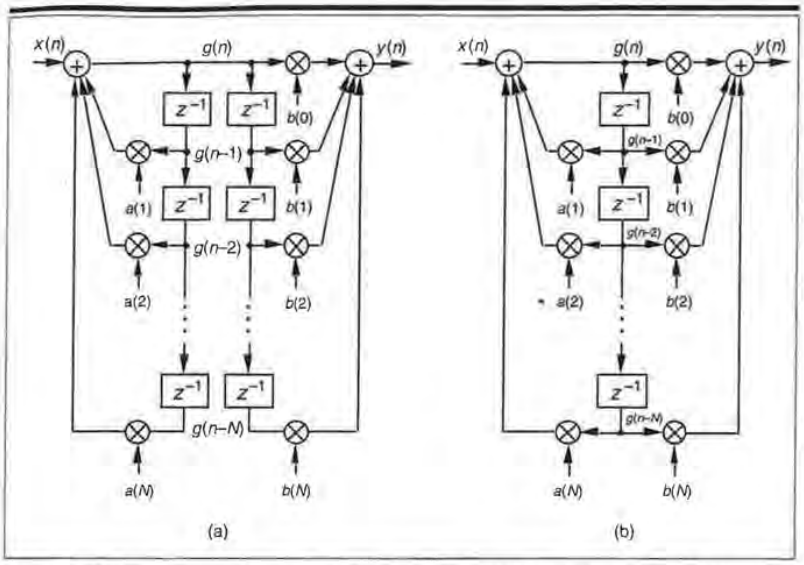


Figure 6-21 Rearranged *N*th-order IIR filter structure: (a) feed forward and feedback paths swapped from the structure in Figure 6-3; (b) final simplified Direct Form II filter structure.

a new sequence that we'll call g(n) is generated. The two identical delay paths in Figure 6-21(a) provide the motivation for this reorientation. Because the sequence g(n) is being shifted down along both delay paths in Figure 6-21(a), we can eliminate one of the paths and arrive at the simplified *Direct Form II* filter structure shown in Figure 6-21(b). Notice that eliminating a delay path simplifies the structure, so that a hardware implementation would need only half the number of delay storage registers required by the original IIR structure in Figure 6-18.

Although the filter structure depicted in Figure 6-21(b) is very popular, in the literature we'll also encounter the equivalent representation shown in Figure 6-22. It's interesting to compare Figure 6-22's IIR structure with the FIR structure in Figure 6-2(a). If the feedback coefficients $a(k)$ in Figure 6-22 were all zeros, for example, then this IIR filter would be equivalent to the FIR filter in Figure 6-2(a) as long as the $b(k)$ IIR coefficients were equal to the $h(k)$ FIR coefficients. It's very important to keep in mind that Eq. (6-21), associated with the IIR structure in Figure 6-18, does not explicitly describe the filter structures in Figures 6-21(b) and 6-22. However, the frequency-domain expressions in Eqs. (6-25) through (6-35) can be used to analyze the performance of the popular filter structures in Figures 6-21(b) and 6-22.

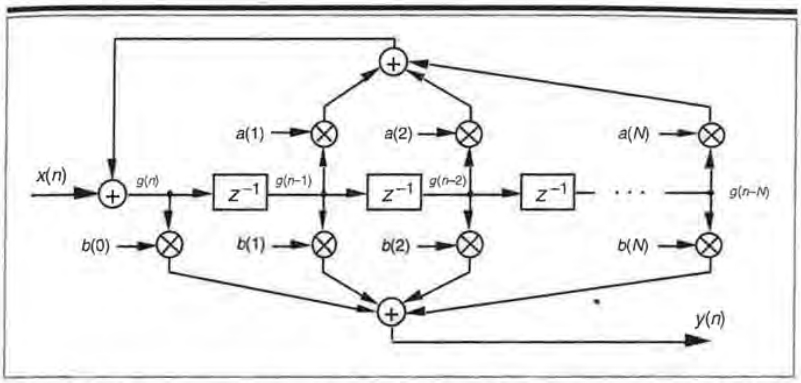


Figure 6-22 Alternate depiction of the IIR filter Direct Form II structure in Figure 6-21(b).

By the way, because of the feedback nature of IIR filters, they're sometimes referred to as *recursive* filters. Similarly, FIR filters are often called *nonrecursive* filters. (The term *recursive* originally signified filters with both poles and zeros on the complex z-plane whereas *nonrecursive* signified filters with only zeros on the z-plane.) We'll avoid the terms *recursive* and *nonrecursive*, however, because it's been shown that filters with only zeros can be designed with recursive structures and that filters with both poles and zeros on the z-plane can be designed with nonrecursive structures[10]. So the terms *recursive* and *nonrecursive* more correctly apply to the structures used to implement a digital filter, and the terms *IIR* and *FIR* better describe whether a filter has only zeros (FIR) or both poles and zeros (IIR) on the z-plane[11].

Now that we have a feeling for the performance and implementation structures of IIR filters, let's briefly introduce three filter design techniques. These IIR design methods go by the impressive names of impulse *invariance*, *bilinear transform*, and *optimized methods*. The first two methods use *analytical* techniques to design digital filters that approximate continuous analog filters.† The impulse invariance and bilinear transform IIR design methods both start with a closed-form Laplace equation describing the desired analog filter and yield z-transform expressions for the approximating IIR digital filter. The filter designer then solves the z-transform equations to obtain the $a(k)$ and $b(k)$ coefficients for use in the IIR structures in Figures 6-21(b) or 6-22. Because analog filter design

† Due to its popularity, throughout the rest of this chapter we'll use the phrase *analog filter* to designate those filters made up of resistors, capacitors, and inductors, designed to operate on continuous signals.

methods are very well understood, designers can take advantage of an abundant variety of analog filter design techniques to design, say, a Butterworth filter with its very flat passband response or, perhaps, go with a Chebyshev filter with its fluctuating passband response and sharper passband-to-stopband cutoff characteristics.

6.4 Impulse Invariance IIR Filter Design Method

The impulse invariance method of IIR filter design is based upon the notion that we can design a discrete filter whose time-domain impulse response is a sampled version of the impulse response of a continuous analog filter. If that analog filter (often called the *prototype filter*) has some desired frequency response, then our IIR filter will yield a discrete approximation of that desired response. The impulse response equivalence of this design method is depicted in Figure 6-23 where we use the conventional notation of δ to represent an impulse function and $h_c(t)$ is the analog filter's impulse response. We use the subscript "c" in Figure 6-23(a) to emphasize the continuous nature of the analog filter. Figure 6-23(b) illustrates the definition of the discrete filter's impulse response: the filter's time-domain output sequence when the input is a single unity-valued sample (impulse) preceded and followed by all zero-valued samples. Our goal is to design a digital filter whose impulse response is a sampled version of the analog filter's continuous impulse response. Implied in the correspondence of the continuous and discrete impulse responses is the property that we can map each pole on the s-plane for the analog filter's

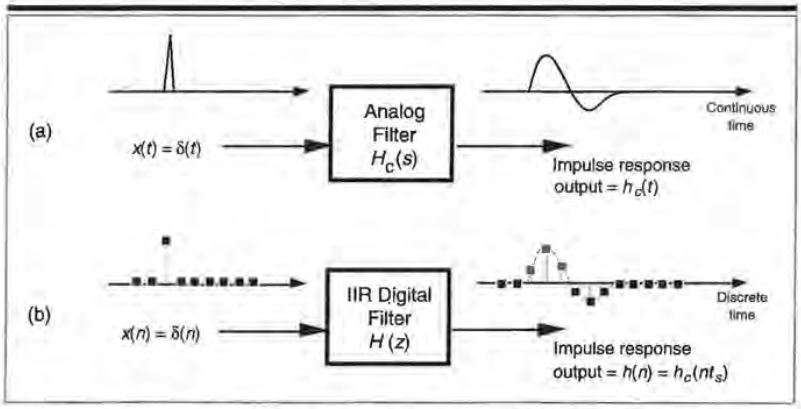


Figure 6-23 Impulse invariance design equivalence of (a) analog filter continuous impulse response; (b) digital filter discrete impulse response.

$H_c(s)$ transfer function to a pole on the z-plane for the discrete IIR filter's $H(z)$ transfer function. What designers have found is that the impulse invariance method does yield useful IIR filters, as long as the sampling rate is high relative to the bandwidth of the signal to be filtered. In other words, IIR filters designed using the impulse invariance method are susceptible to aliasing problems because practical analog filters cannot be perfectly band-limited. Aliasing will occur in an IIR filter's frequency response as shown in Figure 6-24.

From what we've learned in Chapter 2 about the spectral replicating effects of sampling, if Figure 6-4(a) is the spectrum of the continuous $h_c(t)$ impulse response, then the spectrum of the discrete $h_c(nt_s)$ sample sequence is the replicated spectra in Figure 6-24(b).

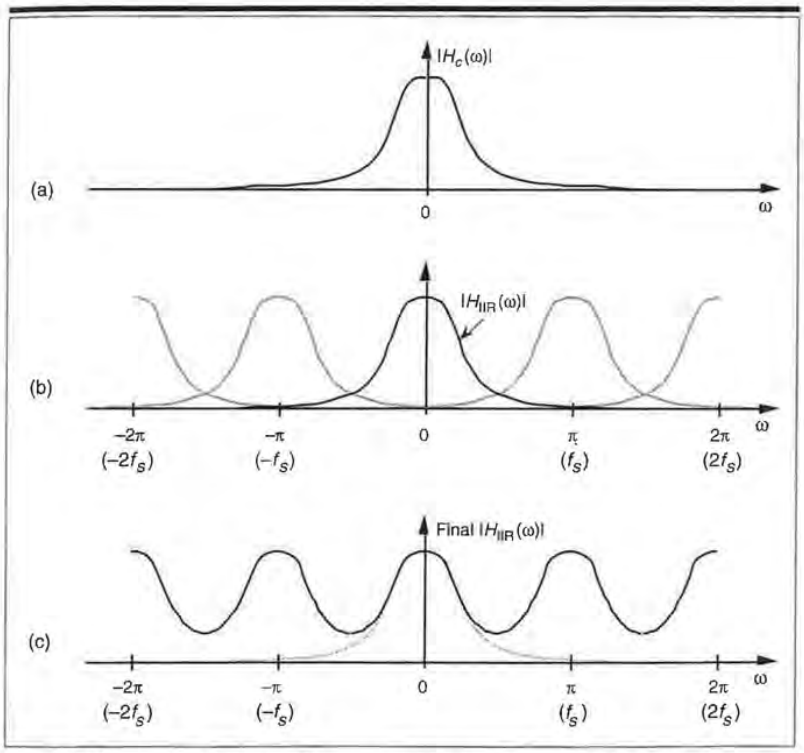


Figure 6-24 Aliasing in the impulse invariance design method: (a) prototype analog filter magnitude response; (b) replicated magnitude responses where $H_{IIR}(\omega)$ is the discrete Fourier transform of $h(n) = h_c(nt_s)$; (c) potential resultant IIR filter magnitude response with aliasing effects.

In Figure 6-24(c) we show the possible effect of aliasing where the dashed curve is a desired $H_{IIR}(\omega)$ frequency magnitude response. However, the actual frequency magnitude response, indicated by the solid curve, can occur when we use the impulse invariance design method. For this reason, we prefer to make the sample frequency f_s as large as possible to minimize the overlap between the primary frequency response curve and its replicated images spaced at multiples of $\pm f_s$ Hz. To see how aliasing can affect IIR filters designed with this method, let's list the necessary impulse invariance design steps and then go through a filter design example.

There are two different methods for designing IIR filters using impulse invariance. The first method, which we'll call Method 1, requires that an inverse Laplace transform as well as a z-transform be performed[12,13]. The second impulse invariance design technique, Method 2, uses a direct substitution process to avoid the inverse Laplace and z-transformations at the expense of needing partial fraction expansion algebra necessary to handle polynomials[14–17]. Both of these methods seem complicated when described in words, but they're really not as difficult as they sound. Let's compare the two methods by listing the steps required for each of them. The impulse invariance design Method 1 goes like this:

Method 1, Step 1: Design (or have someone design for you) a prototype analog filter with the desired frequency response.† The result of this step is a continuous Laplace transfer function $H_c(s)$ expressed as the ratio of two polynomials, such as,

$$H_c(s) = \frac{b(N)s^N + b(N-1)s^{N-1} + \ldots + b(1)s + b(0)}{a(M)s^M + a(M-1)s^{M-1} + \ldots + a(1)s + a(0)} = \frac{\sum_{k=0}^{N} b(k)s^k}{\sum_{k=0}^{M} a(k)s^k} \qquad (6\text{-}43)$$

which is the general form of Eq. (6-10) with $N < M$, and $a(k)$ and $b(k)$ are constants.

Method 1, Step 2: Determine the analog filter's continuous time-domain impulse response $h_c(t)$ from the $H_c(s)$ Laplace

† In a low-pass filter design, for example, the filter type (Chebyshev, Butterworth, elliptic), filter order (number of poles), and the cutoff frequency are parameters to be defined in this step.

transfer function. I hope, this can be done using Laplace tables as opposed to actually evaluating an inverse Laplace transform equation.

Method 1, Step 3: Determine the digital filter's sampling frequency f_s, and calculate the sample period as $t_s = 1/f_s$. The f_s sampling rate is chosen based on the absolute frequency, in Hz, of the prototype analog filter. Because of the aliasing problems associated with this impulse invariance design method, later, we'll see why f_s should be made as large as practical.

Method 1, Step 4: Find the z-transform of the continuous $h_c(t)$ to obtain the IIR filter's z-domain transfer function $H(z)$ in the form of a ratio of polynomials in z.

Method 1, Step 5: Substitute the value (not the variable) t_s for the continuous variable t in the $H(z)$ transfer function obtained in Step 4. In performing this step, we are ensuring, as in Figure 6-23, that the IIR filter's discrete $h(n)$ impulse response is a sampled version of the continuous filter's $h_c(t)$ impulse response so that $h(n) = h_c(nt_s)$, for $0 \le n \le \infty$.

Method 1, Step 6: Our $H(z)$ from Step 5 will now be of the general form

$$H(z) = \frac{b(N)z^{-N} + b(N-1)z^{-(N-1)} + \ldots + b(1)z^{-1} + b(0)}{a(M)z^{M} + a(M-1)z^{-(M-1)} + \ldots + a(1)z^{-1} + a(0)} = \frac{\sum_{k=0}^{N} b(k)z^{-k}}{1 - \sum_{k=1}^{M} a(k)z^{-k}} \quad (6\text{-}44)$$

Because the process of sampling the continuous impulse response results in a digital filter frequency response that's scaled by a factor of $1/t_s$, many filter designers find it appropriate to include the t_s factor in Eq. (6-44). So we can rewrite Eq. (6-44) as

$$H(z) = \frac{Y(z)}{X(z)} = \frac{t_s \sum_{k=0}^{N} b(k)z^{-k}}{1 - \sum_{k=1}^{M} a(k)z^{-k}} . \quad (6\text{-}45)$$

Incorporating the value of t_s in Eq. (6-45), then, makes the IIR filter time-response scaling independent of the sampling rate, and the discrete filter will have the same gain as the prototype analog filter.†

Method 1, Step 7: Because Eq. (6-44) is in the form of Eq. (6-25), by inspection, we can express the filter's time-domain difference equation in the general form of Eq. (6-21) as

$$y(n) = b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + \ldots + b(N)x(n-N) + a(1)y(n-1) + a(2)y(n-2) + \ldots + a(M)y(n-M) \quad . \tag{6-46}$$

Choosing to incorporate t_s, as in Eq. (6-45), to make the digital filter's gain equal to the prototype analog filter's gain, by multiplying the $b(k)$ coefficients by the sample period t_s, leads to an IIR filter time-domain expression of the form

$$y(n) = t_s \cdot [b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + \ldots + b(N)x(n-N)] + a(1)y(n-1) + a(2)y(n-2) + \ldots + a(M)y(n-M). \tag{6-47}$$

Notice how the signs changed for the $a(k)$ coefficients from Eqs. (6-44) and (6-45) to Eqs. (6-46) and (6-47). These sign changes always seem to cause problems for beginners, so watch out. Also, keep in mind that the time-domain expressions in Eq. (6-46) and Eq. (6-47) apply only to the filter structure in Figure 6-18. The $a(k)$ and $b(k)$, or $t_s \cdot b(k)$, coefficients, however, can be applied to the improved IIR structure shown in Figure 6-22 to complete our design.

Before we go through an actual example of this design process, let's discuss the other impulse invariance design method.

† Some authors have chosen to include the t_s factor in the discrete $h(n)$ impulse response in the above Step 4; that is, make $h(n) = t_s h_c(nt_s)$ [14, 18]. The final result of this, of course, is the same as that obtained by including t_s as described in Step 5.

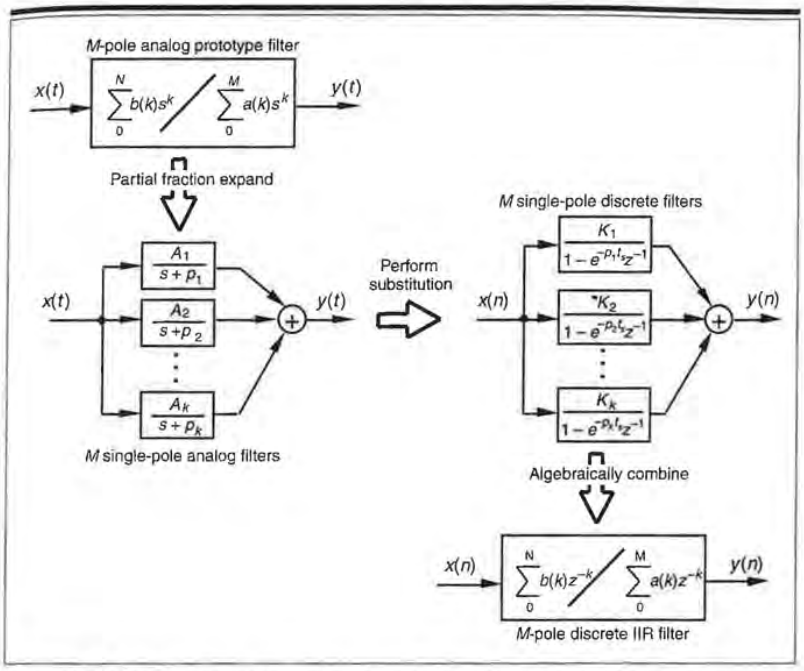


Figure 6-25 Mathematical flow of the impulse invariance design Method 2.

The impulse invariance Design Method 2, also called the standard z-transform method, takes a different approach. It mathematically partitions the prototype analog filter into multiple single-pole continuous filters and then approximates each one of those by a single-pole digital filter. The set of M single-pole digital filters is then algebraically combined to form an M-pole, Mth-ordered IIR filter. This process of breaking the analog filter to discrete filter approximation into manageable pieces is shown in Figure 6-25. The steps necessary to perform an impulse invariance Method 2 design are

Method 2, Step 1: Obtain the Laplace transfer function $H_c(s)$ for the prototype analog filter in the form of Eq. (6-43). (Same as Method 1, Step 1.)

Method 2, Step 2: Select an appropriate sampling frequency f_s and calculate the sample period as $t_s = 1/f_s$. (Same as Method 1, Step 3.)

Method 2, Step 3: Express the analog filter's Laplace transfer function $H_c(s)$ as the sum of single-pole filters. This requires us to use partial fraction expansion methods to express the ratio of polynomials in Eq. (6-43) in the form of

$$H_c(s) = \frac{b(N)s^N + b(N-1)s^{N-1} + \ldots + b(1)s + b(0)}{a(M)s^M + a(M-1)s^{M-1} + \ldots + a(1)s + a(0)}$$

$$= \sum_{k=1}^{M} \frac{A_k}{s + p_k} = \frac{A_1}{s + p_1} + \frac{A_2}{s + p_2} + \ldots + \frac{A_M}{s + p_M}, \quad (6\text{-}48)$$

where the individual A_k factors are constants and the kth pole is located at $-p_k$ on the s-plane. We'll denote the kth single-pole analog filter as $H_k(s)$, or

$$H_k(s) = \frac{A_k}{s + p_k}. \quad (6\text{-}49)$$

Method 2, Step 4: Substitute $1 - e^{-p_k t_s} z^{-1}$ for $s + p_k$ in Eq. (6-48). This mapping of each $H_k(s)$ pole, located at $s = -p_k$ on the s-plane, to the $z = e^{-p_k t_s}$ location on the z-plane is how we approximate the impulse response of each single-pole analog filter by a single-pole digital filter. (The reader can find the derivation of this $1 - e^{-p_k t_s} z^{-1}$ substitution, illustrated in our Figure 6-25, in references [14] through [16].) So, the kth analog single-pole filter $H_k(s)$ is approximated by a single-pole digital filter whose z-domain transfer function is

$$H_k(z) = \frac{A_k}{1 - e^{-p_k t_s} z^{-1}}. \quad (6\text{-}50)$$

The final combined discrete filter transfer function $H(z)$ is the sum of the single-poled discrete filters, or

$$H(z) = \sum_{k=1}^{M} H_k(z) = \sum_{k=1}^{M} \frac{A_k}{1 - e^{-p_k t_s} z^{-1}}. \quad (6\text{-}51)$$

Keep in mind that the above $H(z)$ is not a function of time. The t_s factor in Eq. (6-51) is a constant equal to the discrete-time sample period.

Method 2, Step 5: Calculate the z-domain transfer function of the sum of the M single-pole digital filters in the form of a ratio of two polynomials in z. Because the $H(z)$ in Eq. (6-51) will be a series of fractions, we'll have to combine those fractions over a common denominator to get a single ratio of polynomials in the familiar form of

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} b(k)z^{-k}}{1 - \sum_{k=1}^{M} a(k)z^{-k}} \quad . \tag{6-52}$$

Method 2, Step 6: Just as in Method 1 Step 6, by inspection, we can express the filter's time-domain equation in the general form of

$$\begin{aligned} y(n) = {} & b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + \ldots + b(N)x(n-N) \\ & + a(1)y(n-1) + a(2)y(n-2) + \ldots + a(M)y(n-M) \quad . \end{aligned} \tag{6-53}$$

Again, notice the $a(k)$ coefficient sign changes from Eq. (6-52) to Eq. (6-53). As described in Method 1 Steps 6 and 7, if we choose to make the digital filter's gain equal to the prototype analog filter's gain by multiplying the $b(k)$ coefficients by the sample period t_s, then the IIR filter's time-domain expression will be in the form

$$\begin{aligned} y(n) = {} & t_s \cdot [b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + \ldots + b(N)x(n-N)] \\ & + a(1)y(n-1) + a(2)y(n-2) + \ldots + a(M)y(n-M) \quad , \end{aligned} \tag{6-54}$$

yielding a final $H(z)$ z-domain transfer function of

$$H(z) = \frac{Y(z)}{X(z)} = \frac{t_s \cdot \sum_{k=0}^{N} b(k)z^{-k}}{1 - \sum_{k=1}^{M} a(k)z^{-k}} \quad (6\text{-}54')$$

Finally, we can implement the improved IIR structure shown in Figure 6-22 using the $a(k)$ and $b(k)$ coefficients from Eq. (6-53) or the $a(k)$ and $t_s \cdot b(k)$ coefficients from Eq. (6-54).

To provide a more meaningful comparison between the two impulse invariance design methods, let's dive in and go through an IIR filter design example using both methods.

6.4.1 Impulse Invariance Design Method 1 Example

Assume that we need to design an IIR filter that approximates a second-order Chebyshev prototype analog low-pass filter whose passband ripple is 1 dB. Our f_s sampling rate is 100 Hz ($t_s = 0.01$), and the filter's 1 dB cutoff frequency is 20 Hz. Our prototype analog filter will have a frequency magnitude response like that shown in Figure 6-26.

Given the above filter requirements, assume that the analog prototype filter design effort results in the $H_c(s)$ Laplace transfer function of

$$H_c(s) = \frac{17410.145}{s^2 + 137.94536s + 17410.145} \quad (6\text{-}55)$$

It's the transfer function in Eq. (6-55) that we intend to approximate with our discrete IIR filter. To find the analog filter's impulse response, we'd

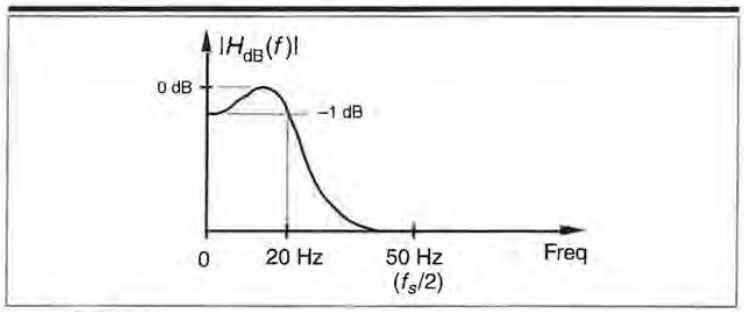


Figure 6-26 Frequency magnitude response of the example prototype analog filter.

like to get $H_c(s)$ into a form that allows us to use Laplace transform tables to find $h_c(t)$. Searching through systems analysis textbooks we find the following Laplace transform pair:

$X(s)$, Laplace transform of $x(t)$:	$x(t)$:
$\frac{A\omega}{(s+\alpha)^2+\omega^2}$	$Ae^{-\alpha t}\cdot\sin(\omega t)$. (6-56)

Our intent, then, is to modify Eq. (6-55) to get it into the form on the left side of Eq. (6-56). We do this by realizing that the Laplace transform expression in Eq. (6-56) can be rewritten as

$$\frac{A\omega}{(s+\alpha)^2+\omega^2}=\frac{A\omega}{s^2+2\alpha s+\alpha^2+\omega^2} \quad . \tag{6-57}$$

If we set Eq. (6-55) equal to the right side of Eq. (6-57), we can solve for A, α, and ω. Doing that,

$$H_c(s)=\frac{17410.145}{s^2+137.94536s+17410.145}=\frac{A\omega}{s^2+2\alpha s+\alpha^2+\omega^2} \quad . \tag{6-58}$$

Solving Eq. (6-58) for A, α, and ω, we first find

$$\alpha=\frac{137.94536}{2}=68.972680 \; ; \tag{6-59}$$

$$\alpha^2+\omega^2=17410.145 \; , \tag{6-60}$$

so

$$\omega=\sqrt{17410.145-\alpha^2}=112.485173 ; \tag{6-61}$$

and

$$A=\frac{17410.145}{\omega}=154.77724 \quad . \tag{6-62}$$

OK, we can now express $H_c(s)$ in the desired form of the left side of Eq. (6-57) as

$$H_c(s) = \frac{(154.77724)(112.485173)}{(s+68.972680)^2 + (112.485173)^2} \quad . \tag{6-63}$$

Using the Laplace transform pair in Eq. (6-56), the time-domain impulse response of the prototype analog filter becomes

$$h_c(t) = Ae^{-\alpha t} \cdot \sin(\omega t) = 154.77724e^{-68.972680t} \cdot \sin(112.485173t) \quad . \tag{6-64}$$

OK, we're ready to perform Method 1, Step 4, to determine the discrete IIR filter's z-domain transfer function $H(z)$ by performing the z-transform of $h_c(t)$. Again, scanning through digital signal processing textbooks or a good math reference book, we find the following z-transform pair where the time-domain expression is in the same form as Eq. (6-64)'s $h_c(t)$ impulse response:

$x(t)$:	$X(z)$, z - transform of $x(t)$:
$Ce^{-\alpha t} \cdot \sin(\omega t)$	$\dfrac{Ce^{-\alpha t} \cdot \sin(\omega t)z^{-1}}{1-2[e^{-\alpha t} \cdot \cos(\omega t)]z^{-1} + e^{-2\alpha t}z^{-2}}$.

(6-65)

Remember now, the α and ω in Eq. (6-65) are generic and are not related to the α and ω values in Eq. (6-59) and Eq. (6-61). Substituting the constants from Eq. (6-64) into the right side of Eq. (6-65), we get the z-transform of the IIR filter as

$$H(z) = \frac{154.77724e^{-68.972680t} \cdot \sin(112.485173t)z^{-1}}{1-2[e^{-68.972680t} \cdot \cos(112.485173t)]z^{-1} + e^{-2\cdot 68.972680t}z^{-2}} \quad . \tag{6-66}$$

Performing Method 1, Step 5, we substitute the t_s value of 0.01 for the continuous variable t in Eq. (6-66), yielding the final $H(z)$ transfer function of

$$H(z) = \frac{154.77724e^{-68.972680\cdot 0.01} \cdot \sin(112.485173 \cdot 0.01)z^{-1}}{1-2[e^{-68.972680\cdot 0.01} \cdot \cos(112.485173 \cdot 0.01)]z^{-1} + e^{-2\cdot 68.972680\cdot 0.01}z^{-2}}$$

$$= \frac{154.77724e^{-0.68972680} \cdot \sin(1.12485173)z^{-1}}{1-2[e^{-0.68972680} \cdot \cos(1.12485173)]z^{-1} + e^{-2\cdot 0.68972680}z^{-2}}$$

$$= \frac{Y(z)}{X(z)} = \frac{70.059517z^{-1}}{1-0.43278805z^{-1} + 0.25171605z^{-2}} \quad . \tag{6-67}$$

OK, hang in there, we're almost finished. Here are the final steps of Method 1. Because of the transfer function $H(z) = Y(z)/X(z)$, we can cross-multiply the denominators to rewrite the bottom line of Eq. (6-67) as

$$Y(z)\cdot(1-0.43278805z^{-1}+0.25171605z^{-2}) = X(z)\cdot(70.059517z^{-1}),$$

or

$$Y(z) = 70.059517\cdot X(z)z^{-1} + 0.43278805\cdot Y(z)z^{-1} - 0.25171605\cdot Y(z)z^{-2}. \quad (6\text{-}68)$$

By inspection of Eq. (6-68), we can now get the time-domain expression for our IIR filter. Performing Method 1, Steps 6 and 7, we multiply the $x(n-1)$ coefficient by the sample period value of $t_s = 0.01$ to allow for proper scaling as

$$\begin{aligned} y(n) &= 0.01\cdot 70.059517\cdot x(n-1) + 0.43278805\cdot y(n-1) - 0.25171605\cdot y(n-2) \\ &= 0.70059517\cdot x(n-1) + 0.43278805\cdot y(n-1) - 0.25171605\cdot y(n-2), \end{aligned} \quad (6\text{-}69)$$

and there we (finally) are. The coefficients from Eq. (6-69) are what we use in implementing the improved IIR structure shown in Figure 6-22 to approximate the original second-order Chebyshev analog low-pass filter.

Let's see if we get the same result if we use the impulse invariance design Method 2 to approximate the example prototype analog filter.

6.4.2 Impulse Invariance Design Method 2 Example

Given the original prototype filter's Laplace transfer function as

$$H_c(s) = \frac{17410.145}{s^2 + 137.94536s + 17410.145}, \quad (6\text{-}70)$$

and the value of $t_s = 0.01$ for the sample period, we're ready to proceed with Method 2's Step 3. To express $H_c(s)$ as the sum of single-pole filters, we'll have to factor the denominator of Eq. (6-70) and use partial fraction expansion methods. For convenience, let's start by replacing the constants in Eq. (6-70) with variables in the form of

$$H_c(s) = \frac{c}{s^2 + bs + c}, \quad (6\text{-}71)$$

where $b = 137.94536$, and $c = 17410.145$. Next, using Eq. (6-15) with $a = 1$, we can factor the quadratic denominator of Eq. (6-71) into

$$H_c(s) = \frac{c}{\left(s + \frac{b}{2} + \sqrt{\frac{b^2 - 4c}{4}}\right) \cdot \left(s + \frac{b}{2} - \sqrt{\frac{b^2 - 4c}{4}}\right)} . \tag{6-72}$$

If we substitute the values for b and c in Eq. (6-72), we'll find that the quantity under the radical sign is negative. This means that the factors in the denominator of Eq. (6-72) are complex. Because we have lots of algebra ahead of us, let's replace the radicals in Eq. (6-72) with the *imaginary* term jR, where $j = \sqrt{-1}$ and $R = |(b^2-4c)/4|$, such that

$$H_c(s) = \frac{c}{(s + b/2 + jR)(s + b/2 - jR)} . \tag{6-73}$$

OK, partial fraction expansion methods allow us to partition Eq. (6-73) into two separate fractions of the form

$$\begin{aligned} H_c(s) &= \frac{c}{(s + b/2 + jR)(s + b/2 - jR)} \\ &= \frac{K_1}{(s + b/2 + jR)} + \frac{K_2}{(s + b/2 - jR)} , \end{aligned} \tag{6-74}$$

where the K_1 constant can be found to be equal to $jc/2R$ and constant K_2 is the complex conjugate of K_1, or $K_2 = -jc/2R$. (To learn the details of partial fraction expansion methods, the interested reader should investigate standard college algebra or engineering mathematics textbooks.) Thus, $H_c(s)$ can be of the form in Eq. (6-48) or

$$H_c(s) = \frac{jc/2R}{(s + b/2 + jR)} + \frac{-jc/2R}{(s + b/2 - jR)} . \tag{6-75}$$

We can see from Eq. (6-75) that our second-order prototype filter has two poles, one located at $p_1 = -b/2 - jR$ and the other at $p_2 = -b/2 + jR$. We're now ready to map those two poles from the s-plane to the z-plane as called out in Method 2, Step 4. Making our $1 - e^{-p_k t_s} z^{-1}$ substitution for the $s + p_k$ terms in Eq. (6-75), we have the following expression for the z-domain single-pole digital filters,

$$H(z) = \frac{jc/2R}{1 - e^{-(b/2+jR)t_s}z^{-1}} + \frac{-jc/2R}{1 - e^{-(b/2-jR)t_s}z^{-1}} \quad . \tag{6-76}$$

Our objective in Method 2, Step 5 is to massage Eq. (6-76) into the form of Eq. (6-52), so that we can determine the IIR filter's feed forward and feedback coefficients. Putting both fractions in Eq. (6-76) over a common denominator gives us

$$H(z) = \frac{(jc/2R)(1 - e^{-(b/2-jR)t_s}z^{-1}) - (jc/2R)(1 - e^{-(b/2+jR)t_s}z^{-1})}{(1 - e^{-(b/2+jR)t_s}z^{-1})(1 - e^{-(b/2-jR)t_s}z^{-1})} . \tag{6-77}$$

Collecting like terms in the numerator and multiplying out the denominator gives us

$$H(z) = \frac{(jc/2R)(1 - e^{-(b/2-jR)t_s}z^{-1} - 1 + e^{-(b/2+jR)t_s}z^{-1})}{1 - e^{-(b/2-jR)t_s}z^{-1} - e^{-(b/2+jR)t_s}z^{-1} + e^{-bt_s}z^{-2}} \quad . \tag{6-78}$$

Factoring the exponentials and collecting like terms of powers of z in Eq. (6-78),

$$H(z) = \frac{(jc/2R)(e^{-(b/2+jR)t_s} - e^{-(b/2-jR)t_s})z^{-1}}{1 - (e^{-(b/2-jR)t_s} + e^{-(b/2+jR)t_s})z^{-1} + e^{-bt_s}z^{-2}} \quad . \tag{6-79}$$

Continuing to simplify our $H(z)$ expression by factoring out the real part of the exponentials,

$$H(z) = \frac{(jc/2R)e^{-bt_s/2}(e^{-jRt_s} - e^{jRt_s})z^{-1}}{1 - e^{-bt_s/2}(e^{jRt_s} + e^{-jRt_s})z^{-1} + e^{-bt_s}z^{-2}} \quad . \tag{6-80}$$

We now have $H(z)$ in a form with all the like powers of z combined into single terms, and Eq. (6-80) looks something like the desired form of Eq. (6-52). Knowing that the final coefficients of our IIR filter must be real numbers, the question is "What do we do with those imaginary j terms in Eq. (6-80)?" Once again, Euler to the rescue.† Using Euler's equations for sinusoids, we can eliminate the imaginary exponentials and Eq. (6-80) becomes

† From Euler, we know that $\sin(\varnothing) = (e^{j\varnothing} - e^{-j\varnothing})/2j$, and $\cos(\varnothing) = (e^{j\varnothing} + e^{-j\varnothing})/2$.

$$H(z) = \frac{(jc/2R)e^{-bt_s/2}[-2j\sin(Rt_s)]z^{-1}}{1 - e^{-bt_s/2}[2\cos(Rt_s)]z^{-1} + e^{-bt_s}z^{-2}}$$

$$= \frac{(c/R)e^{-bt_s/2}[\sin(Rt_s)]z^{-1}}{1 - e^{-bt_s/2}[2\cos(Rt_s)]z^{-1} + e^{-bt_s}z^{-2}} . \quad (6\text{-}81)$$

If we plug the values c = 17410.145, b = 137.94536, R = 112.48517, and t_s = 0.01 into Eq. (6-81), we get the following IIR filter transfer function:

$$H(z) = \frac{(154.77724)(0.50171312)(0.902203655)z^{-1}}{1 - (0.50171312)(0.86262058)z^{-1} + 0.25171605z^{-2}}$$

$$= \frac{70.059517z^{-1}}{1 - 0.43278805z^{-1} + 0.25171605z^{-2}} . \quad (6\text{-}82)$$

Because the transfer function $H(z) = Y(z)/X(z)$, we can again cross-multiply the denominators to rewrite Eq. (6-82) as

$$Y(z)\cdot(1 - 0.43278805z^{-1} + 0.25171605z^{-2}) = X(z)\cdot(70.059517z^{-1}) ,$$

or

$$Y(z) = 70.059517\cdot X(z)z^{-1} + 0.43278805\cdot Y(z)z^{-1} - 0.25171605\cdot Y(z)z^{-2} . \quad (6\text{-}83)$$

Now we take the inverse z-transform of Eq. (6-83), by inspection, to get the time-domain expression for our IIR filter as

$$y(n) = 70.059517\cdot x(n-1) + 0.43278805\cdot y(n-1) - 0.25171605\cdot y(n-2) . \quad (6\text{-}84)$$

One final step remains. To force the IIR filter gain equal to the prototype analog filter's gain, we multiply the $x(n-1)$ coefficient by the sample period t_s as suggested in Method 2, Step 6. In this case, there's only one $x(n)$ coefficient, giving us

$$y(n) = 0.01\cdot 70.059517\cdot x(n-1) + 0.43278805\cdot y(n-1) - 0.25171605\cdot y(n-2)$$

$$= 0.70059517\cdot x(n-1) + 0.43278805\cdot y(n-1) - 0.25171605\cdot y(n-2), \quad (6\text{-}85)$$

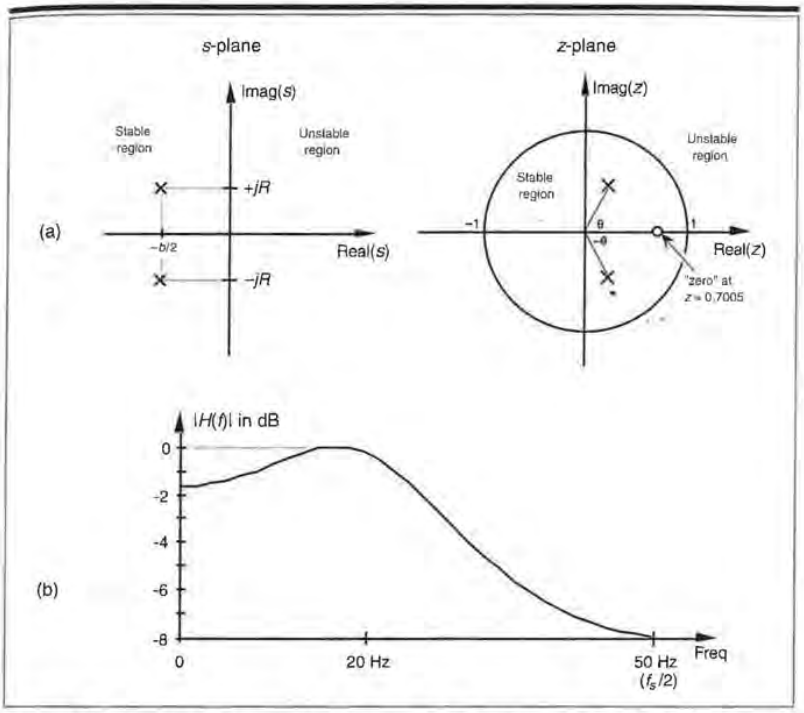


Figure 6-27 Impulse invariance design example filter characteristics: (a) *s*-plane pole locations of prototype analog filter and *z*-plane pole locations of discrete IIR filter; (b) frequency magnitude response of the discrete IIR filter.

that compares well with the Method 1 result in Eq. (6-69). (Isn't it comforting to work a problem two different ways and get the same result?)

Figure 6-27 shows, in graphical form, the result of our IIR design example. The *s*-plane pole locations of the prototype filter and the *z*-plane poles of the IIR filter are shown in Figure 6-27(a). Because the *s*-plane poles are to the left of the origin and the *z*-plane poles are inside the unit circle, both the prototype analog and the discrete IIR filters are stable. We find the prototype filter's *s*-plane pole locations by evaluating $H_c(s)$ in Eq. (6-75). When $s = -b/2 - jR$, the denominator of the first term in Eq. (6-75) becomes zero and $H_c(s)$ is infinitely large. That $s = -b/2 - jR$ value is the location of the lower *s*-plane pole in Figure 6-27(a). When $s = -b/2 + jR$ the denominator of the second term in Eq. (6-75) becomes zero and $s = -b/2 + jR$ is the location of the second *s*-plane pole.

The IIR filter's *z*-plane pole locations are found from Eq. (6-76). If we multiply the numerators and denominators of Eq. (6-76) by z,

$$H(z)\cdot\frac{z}{z}=\frac{z(jc/2R)}{z(1-e^{-(b/2+jR)t_s}z^{-1})}+\frac{z(-jc/2R)}{z(1-e^{-(b/2-jR)t_s}z^{-1})}$$

$$=\frac{(jc/2R)z}{z-e^{-(b/2+jR)t_s}}+\frac{(-jc/2R)z}{z-e^{-(b/2-jR)t_s}}\ . \qquad (6\text{-}86)$$

In Eq. (6-86), when z is set equal to $e^{(-b/2+jR)t_s}$, the denominator of the first term in Eq. (6-86) becomes zero and $H(z)$ becomes infinitely large. The value of z of

$$z=e^{-(b/2+jR)t_s}=e^{-bt_s/2}e^{-jRt_s}=e^{-bt_s/2}\angle -Rt_s \text{ radians} \qquad (6\text{-}87)$$

defines the location of the lower z-plane pole in Figure 6-27(a). Specifically, this lower pole is located at a distance of $e^{-bt_s/2} = 0.5017$ from the origin, at an angle of $\theta = -Rt_s$ radians, or –64.45°. Being conjugate poles, the upper z-plane pole is located the same distance from the origin at an angle of $\theta = Rt_s$ radians, or +64.45°. Figure 6-27(b) illustrates the frequency magnitude response of the IIR filter in Hz.

Two different implementations of our IIR filter are shown in Figure 6-28. Figure 6-28(a) is an implementation of our second-order IIR filter based on the general IIR structure given in Figure 6-22, and Figure 6-28(b) shows the second-order IIR filter implementation based on the alternate structure from Figure 6-21(b). Knowing that the $b(0)$ coefficient on the left side of Figure 6-28(b) is zero, we arrive at the simplified structure on the right side of Figure 6-28(b). Looking carefully at Figure 6-28(a) and the right side of Figure 6-28(b), we can see that they are equivalent.

Although both impulse invariance design methods are covered in the literature, we might ask, "Which one is preferred?" There's no definite answer to that question because it depends on the $H_c(s)$ of the prototype analog filter. Although our Method 2 example above required more algebra than Method 1, if the prototype filter's s-domain poles were located only on the real axis, Method 2 would have been much simpler because there would be no complex variables to manipulate. In general, Method 2 is more popular for two reasons: (1) the inverse Laplace and z-transformations, although straightforward in our Method 1 example, can be very difficult for higher-order filters, and (2) unlike Method 1, Method 2 can be coded in a software routine or a computer spreadsheet.

Upon examining the frequency magnitude response in Figure 6-27(b), we can see that this second-order IIR filter's roll-off is not particularly

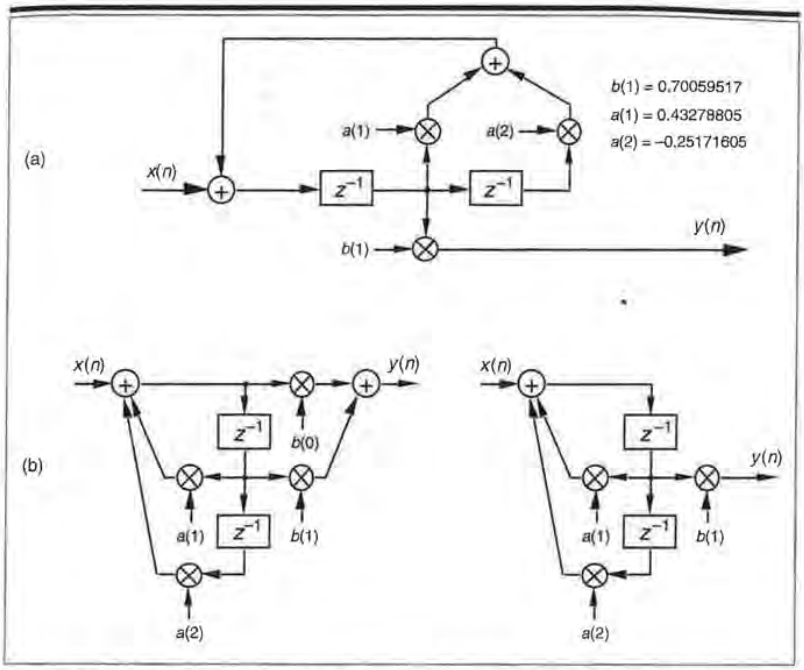


Figure 6-28 Implementations of the impulse invariance design example filter.

steep. This is, admittedly, a simple low-order filter, but it's attenuation slope is so gradual that it doesn't appear to be of much use as a low-pass filter.† We can also see that the filter's passband ripple is greater than the desired value of 1 dB in Figure 6-26. What we'll find is that it's not the low order of the filter that contributes to it's poor performance, but the sampling rate used. That second-order IIR filter response is repeated as the shaded curve in Figure 6-29. If we increased the sampling rate to 200 Hz, we'd get the frequency response shown by the dashed curve in Figure 6-29. Increasing the sampling rate to 400 Hz results in the much improved frequency response indicated by the solid line in the figure. Sampling rate changes do not affect our filter order or implementation structure. Remember, if we change the sampling rate, only the sample period t_s changes in our design equations, resulting in a different set of filter

† A piece of advice. Whenever you encounter any frequency representation (be it a digital filter magnitude response or a signal spectrum) that has nonzero values at $+f_s/2$, be suspicious—be very suspicious—that aliasing is taking place.

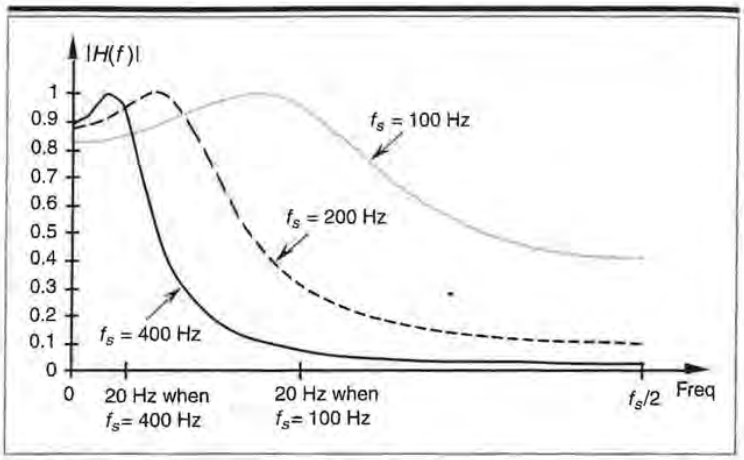


Figure 6-29 IIR filter frequency magnitude response, on a linear scale, at three separate sampling rates. Notice how the filter's absolute cutoff frequency of 20 Hz shifts relative to the different f_s sampling rates.

coefficients for each new sampling rate. So we can see that the smaller we make t_s (larger f_s) the better the resulting filter when either impulse invariance design method is used because the replicated spectral overlap indicated in Figure 6-24(b) is reduced due to the larger f_s sampling rate. The bottom line here is that impulse invariance IIR filter design techniques are most appropriate for narrowband filters; that is, low-pass filters whose cutoff frequencies are much smaller than the sampling rate.

The second analytical technique for analog filter approximation, the bilinear transform method, alleviates the impulse invariance method's aliasing problems at the expense of what's called frequency warping. Specifically, there's a nonlinear distortion between the prototype analog filter's frequency scale and the frequency scale of the approximating IIR filter designed using the bilinear transform. Let's see why.

6.5 Bilinear Transform IIR Filter Design Method

There's a popular analytical IIR filter design technique known as the *bilinear transform* method. Like the impulse invariance method, this design technique approximates a prototype analog filter defined by the continuous Laplace transfer function $H_c(s)$ with a discrete filter whose transfer function is $H(z)$. However, the bilinear transform method has great utility because

- it allows us to simply substitute a function of z for s in $H_c(s)$ to get $H(z)$, thankfully, eliminating the need for Laplace and z-transformations as well as any necessity for partial fraction expansion algebra;
- it maps the entire s-plane to the z-plane, enabling us to completely avoid the frequency-domain aliasing problems we had with the impulse invariance design method; and
- it induces a nonlinear distortion of $H(z)$'s frequency axis, relative to the original prototype analog filter's frequency axis, that sharpens the final roll-off of digital low-pass filters.

Don't worry. We'll explain each one of these characteristics and see exactly what they mean to us as we go about designing an IIR filter.

If the transfer function of a prototype analog filter is $H_c(s)$, then we can obtain the discrete IIR filter z-domain transfer function $H(z)$ by substituting the following for s in $H_c(s)$

$$s = \frac{2}{t_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right), \tag{6-88}$$

where, again, t_s is the discrete filter's sampling period ($1/f_s$). Just as in the impulse invariance design method, when using the bilinear transform method, we're interested in where the analog filter's poles end up on the z-plane after the transformation. This s-plane to z-plane mapping behavior is exactly what makes the bilinear transform such an attractive design technique.[†]

Let's investigate the major characteristics of the bilinear transform's s-plane to z-plane mapping. First we'll show that any pole on the left side of the s-plane will map to the inside of the unit circle in the z-plane. It's easy to show this by solving Eq. (6-88) for z in terms of s. Multiplying Eq. (6-88) by $(t_s/2)(1 + z^{-1})$ and collecting like terms of z leads us to

$$z = \frac{1+(t_s/2)s}{1-(t_s/2)s}, \tag{6-89}$$

[†] The bilinear transform is a technique in the theory of complex variables for mapping a function on the complex plane of one variable to the complex plane of another variable. It maps circles and straight lines to straight lines and circles, respectively.

If we designate the real and imaginary parts of s as

$$s = \sigma + j\omega_a , \tag{6-90}$$

where the subscript in the radian frequency ω_a signifies *analog*, Eq. (6-89) becomes

$$z = \frac{1+\sigma t_s/2 + j\omega_a t_s/2}{1-\sigma t_s/2 - j\omega_a t_s/2} = \frac{(1+\sigma t_s/2) + j\omega_a t_s/2}{(1-\sigma t_s/2) - j\omega_a t_s/2} . \tag{6-91}$$

We see in Eq. (6-91) that z is complex, comprising the ratio of two complex expressions. As such, if we denote z as a magnitude at an angle in the form of $z = |z|\angle\theta_z$, we know that the magnitude of z is given by

$$|z| = \frac{\text{Mag}_{\text{numerator}}}{\text{Mag}_{\text{denominator}}} = \sqrt{\frac{(1+\sigma t_s/2)^2 + (\omega_a t_s/2)^2}{(1-\sigma t_s/2)^2 + (\omega_a t_s/2)^2}} . \tag{6-92}$$

OK, if σ is negative ($\sigma < 0$) the numerator of the ratio on the right side of Eq. (6-92) will be less than the denominator, and $|z|$ will be less than 1. On the other hand, if σ is positive ($\sigma > 0$), the numerator will be larger than the denominator, and $|z|$ will be greater than 1. This confirms that, when using the bilinear transform defined by Eq. (6-88), any pole located on the left side of the s-plane ($\sigma < 0$) will map to a z-plane location inside the unit circle. This characteristic ensures that any stable s-plane pole of a prototype analog filter will map to a stable z-plane pole for our discrete IIR filter. Likewise, any analog filter pole located on the right side of the s-plane ($\sigma > 0$) will map to a z-plane location outside the unit circle when using the bilinear transform. This reinforces our notion that, to avoid filter instability, during IIR filter design, we should avoid allowing any z-plane poles to lie outside the unit circle.

Next, let's show that the $j\omega_a$ axis of the s-plane maps to the perimeter of the unit circle in the z-plane. We can do this by setting $\sigma = 0$ in Eq. (6-91) to get

$$z = \frac{1+j\omega_a t_s/2}{1-j\omega_a t_s/2} . \tag{6-93}$$

Here, again, we see in Eq. (6-93) that z is a complex number comprising the ratio of two complex numbers, and we know the magnitude of this z is given by

$$|z|_{\sigma=0} = \frac{\text{Mag}_{\text{numerator}}}{\text{Mag}_{\text{denominator}}} = \sqrt{\frac{(1)^2 + (\omega_a t_s/2)^2}{(1)^2 + (\omega_a t_s/2)^2}} \quad . \tag{6-94}$$

The magnitude of z in Eq. (6-94) is *always* 1. So, as we stated, when using the bilinear transform, the $j\omega_a$ axis of the s-plane maps to the perimeter of the unit circle in the z-plane. However, this frequency mapping from the s-plane to the unit circle in the z-plane is not linear. It's important to know why this frequency nonlinearity, or warping, occurs and to understand its effects. So we shall, by showing the relationship between the s-plane frequency and the z-plane frequency that we'll designate as ω_d.

If we define z on the unit circle in polar form as $z = re^{-j\omega_d}$ as we did for Figure 6-13, where r is 1 and ω_d is the angle, we can substitute $z = e^{j\omega_d}$ in Eq. (6-88) as

$$s = \frac{2}{t_s}\left(\frac{1 - e^{-j\omega_d}}{1 + e^{-j\omega_d}}\right) . \tag{6-95}$$

If we show s in its rectangular form and partition the ratio in brackets into half-angle expressions,

$$s = \sigma + j\omega_a = \frac{2}{t_s} \cdot \frac{e^{-j\omega_d/2}(e^{j\omega_d/2} - e^{-j\omega_d/2})}{e^{-j\omega_d/2}(e^{j\omega_d/2} + e^{-j\omega_d/2})} \quad . \tag{6-96}$$

Using Euler's relationships of $\sin(\phi) = (e^{j\phi} - e^{-j\phi})/2j$ and $\cos(\phi) = (e^{j\phi} + e^{-j\phi})/2$, we can convert the right side of Eq. (6-96) to rectangular form as

$$\begin{aligned} s = \sigma + j\omega_a &= \frac{2}{t_s} \cdot \frac{e^{-j\omega_d/2}[2j\sin(\omega_d/2)]}{e^{-j\omega_d/2}[2\cos(\omega_d/2)]} \\ &= \frac{2}{t_s} \cdot \frac{2e^{-j\omega_d/2}}{2e^{-j\omega_d/2}} \cdot \frac{j\sin(\omega_d/2)}{\cos(\omega_d/2)} \\ &= \frac{j2}{t_s}\tan(\omega_d/2). \end{aligned} \tag{6-97}$$

If we now equate the real and imaginary parts of Eq. (6-97), we see that $\sigma = 0$, and

$$\omega_a = \frac{2}{t_s}\tan\left(\frac{\omega_d}{2}\right) . \tag{6-98}$$

Let's rearrange Eq. (6-98) to give us the useful expression for the z-domain frequency ω_d, in terms of the s-domain frequency ω_a, of

$$\omega_d = 2\tan^{-1}\left(\frac{\omega_a t_s}{2}\right) . \tag{6-99}$$

The important relationship in Eq. (6-99), which accounts for the so-called frequency warping due to the bilinear transform, is illustrated in Figure 6-30. Notice that, because $\tan^{-1}(\omega_a t_s/2)$ approaches $\pi/2$ as ω_a gets large, ω_d must, then, approach twice that value, or π. This means that no matter how large ω_a gets, the z-plane ω_d will never be greater than π.

Remember how we considered Figure 6-14 and stated that only the $-\pi f_s$ to $+\pi f_s$ radians/s frequency range for ω_a can be accounted for on the z-plane? Well, our new mapping from the bilinear transform maps the entire s-plane to the z-plane, not just the primary strip of the s-plane shown in Figure 6-14. Now, just as a walk along the $j\omega_a$ frequency axis on the s-plane takes us to infinity in either direction, a trip halfway around the unit circle in a counterclockwise direction takes us from $\omega_a = 0$ to $\omega_a = +\infty$ radians/s. As such, the bilinear transform maps the s-plane's entire $j\omega_a$ axis onto the unit circle in the z-plane. We illustrate these bilinear transform mapping properties in Figure 6-31.

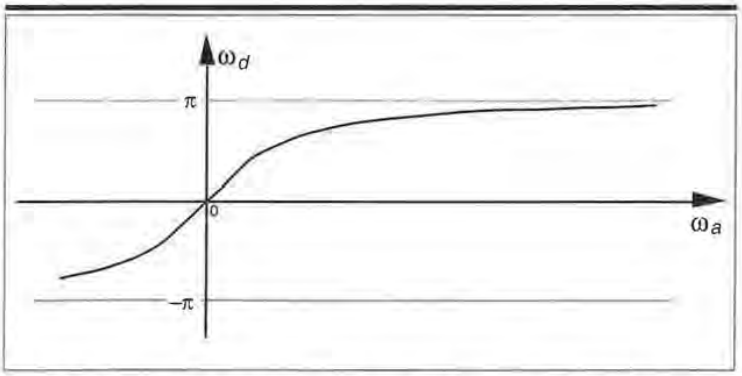


Figure 6-30 Nonlinear relationship between the z-domain frequency ω_d and the s-domain frequency ω_a.

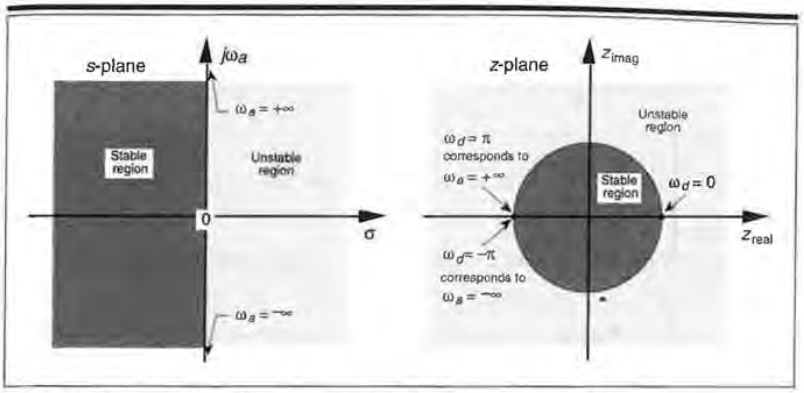


Figure 6-31 Bilinear transform mapping of the *s*-plane to the *z*-plane.

To show the practical implications of this frequency warping, let's relate the *s*-plane and *z*-plane frequencies to the more practical measure of the f_s sampling frequency. We do this by remembering the fundamental relationship between radians/s and Hz of $\omega = 2\pi f$ and solving for f to get

$$f = \frac{\omega}{2\pi} \quad . \tag{6-100}$$

Applying Eq. (6-100) to Eq. (6-99),

$$2\pi f_d = 2\tan^{-1}\left(\frac{2\pi f_a t_s}{2}\right) \quad . \tag{6-101}$$

Substituting $1/f_s$ for t_s, we solve Eq. (6-101) for f_d to get

$$f_d = \left(\frac{2}{2\pi}\right)\tan^{-1}\left(\frac{2\pi f_a / f_s}{2}\right) = \frac{\tan^{-1}(\pi f_a / f_s)}{\pi} \quad . \tag{6-102}$$

Equation (6-102), the relationship between the *s*-plane frequency f_a in Hz and the *z*-plane frequency f_d in Hz, is plotted in Figure 6-32(a) as a function of the IIR filter's sampling rate f_s.

The distortion of the f_a frequency scale, as it translates to the f_d frequency scale, is illustrated in Figure 6-32(b) where an *s*-plane bandpass frequency magnitude response $|H_a(f_a)|$ is subjected to

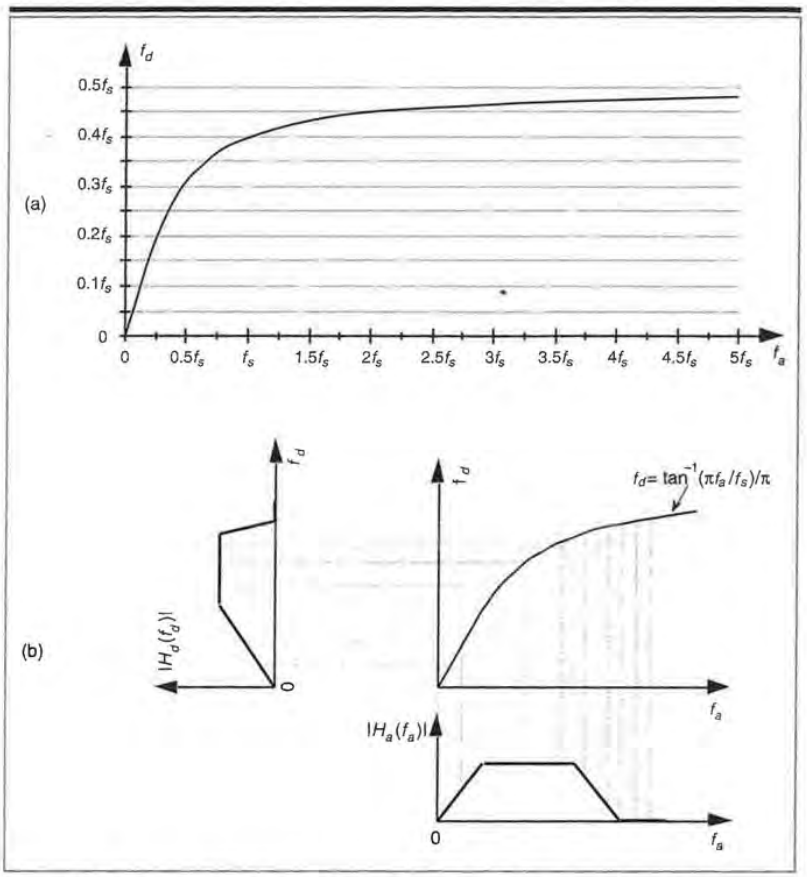


Figure 6-32 Nonlinear relationship between the f_d and f_c frequencies: (a) frequency warping curve scaled in terms of the IIR filter's f_s sampling rate; (b) *s*-domain frequency response $H_c(f_c)$ transformation to a *z*-domain frequency response $H_d(f_d)$.

frequency compression as it is transformed to $|H_d(f_d)|$. Notice how the low-frequency portion of the IIR filter's $|H_d(f_d)|$ response is reasonably linear, but the higher frequency portion has been squeezed in toward zero Hz. That's frequency warping. This figure shows why no IIR filter aliasing can occur with the bilinear transform design method. No matter what the shape or bandwidth of the $|H_a(f_a)|$ prototype analog filter, none of its spectral content can extend beyond half the sampling rate of

$f_s/2$ in $|H_d(f_d)$—and that's what makes the bilinear transform design method as popular as it is.

The steps necessary to perform an IIR filter design using the bilinear transform method are as follows:

Step 1: Obtain the Laplace transfer function $H_c(s)$ for the prototype analog filter in the form of Eq. (6-43).

Step 2: Determine the digital filter's equivalent sampling frequency f_s and establish the sample period $t_s = 1/f_s$.

Step 3: In the Laplace $H_c(s)$ transfer function, substitute the expression

$$\frac{2}{t_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) \tag{6-103}$$

for the variable s to get the IIR filter's $H(z)$ transfer function.

Step 4: Multiply the numerator and denominator of $H(z)$ by the appropriate power of $(1 + z^{-1})$ and grind through the algebra to collect terms of like powers of z in the form

$$H(z) = \frac{\sum_{k=0}^{N} b(k)z^{-k}}{1-\sum_{k=1}^{M} a(k)z^{-k}}. \tag{6-104}$$

Step 5: Just as in the impulse invariance design methods, by inspection, we can express the IIR filter's time-domain equation in the general form of

$$y(n) = b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + \ldots + b(N)x(n-N)$$
$$+ a(1)y(n-1) + a(2)y(n-2) + \ldots + a(M)y(n-M)\ . \tag{6-105}$$

Although the expression in Eq. (6-105) only applies to the filter structure in Figure 6-18, to complete our design, we can apply the $a(k)$ and $b(k)$ coefficients to the improved IIR structure shown in Figure 6-22.

To show just how straightforward the bilinear transform design method really is, let's use it to solve the IIR filter design problem first presented for the impulse invariance design method.

6.5.1 Bilinear Transform Design Example

Again, our goal is to design an IIR filter that approximates the second-order Chebyshev prototype analog low-pass filter, shown in Figure 6-26, whose passband ripple is 1 dB. The f_s sampling rate is 100 Hz ($t_s = 0.01$), and the filter's 1 dB cutoff frequency is 20 Hz. As before, given the original prototype filter's Laplace transfer function as

$$H_c(s) = \frac{17410.145}{s^2 + 137.94536s + 17410.145}, \tag{6-106}$$

and the value of $t_s = 0.01$ for the sample period, we're ready to proceed with Step 3. For convenience, let's replace the constants in Eq. (6-106) with variables in the form of

$$H_c(s) = \frac{c}{s^2 + bs + c}, \tag{6-107}$$

where $b = 137.94536$ and $c = 17410.145$. Performing the substitution of Eq. (6-103) in Eq. (6-107),

$$H(z) = \frac{c}{\left(\frac{2}{t_s}\cdot\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + b\frac{2}{t_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) + c}. \tag{6-108}$$

To simplify our algebra a little, let's substitute the variable a for the fraction $2/t_s$ to give

$$H(z) = \frac{c}{a^2\left(\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + ab\left(\frac{1-z^{-1}}{1+z^{-1}}\right) + c}. \tag{6-109}$$

Proceeding with Step 4, we multiply Eq. (109)'s numerator and denominator by $(1 + z^{-1})^2$ to yield

$$H(z) = \frac{c(1+z^{-1})^2}{a^2(1-z^{-1})^2 + ab(1+z^{-1})(1-z^{-1}) + c(1+z^{-1})^2}. \quad (6\text{-}110)$$

Multiplying through by the factors in the denominator of Eq. (6-110), and collecting like powers of z,

$$H(z) = \frac{c(1+2z^{-1}+z^{-2})}{(a^2+ab+c) + (2c-2a^2)z^{-1} + (a^2+c-ab)z^{-2}}. \quad (6\text{-}111)$$

We're almost there. To get Eq. (6-111) into the form of Eq. (6-104) with a constant term of one in the denominator, we divide Eq. (6-111)'s numerator and denominator by $(a^2 + ab + c)$ giving us

$$H(z) = \frac{\frac{c}{(a^2+ab+c)}(1+2z^{-1}+z^{-2})}{1 + \frac{(2c-2a^2)}{(a^2+ab+c)}z^{-1} + \frac{(a^2+c-ab)}{(a^2+ab+c)}z^{-2}}. \quad (6\text{-}112)$$

We now have $H(z)$ in a form with all the like powers of z combined into single terms, and Eq. (6-112) looks something like the desired form of Eq. (6-104). If we plug the values $a = 2/t_s = 200$, $b = 137.94536$, and $c = 17410.145$ into Eq. (6-112) we get the following IIR filter transfer function:

$$H(z) = \frac{0.20482712(1+2z^{-1}+z^{-2})}{1 - 0.53153089z^{-1} + 0.35083938z^{-2}}$$

$$= \frac{0.20482712 + 0.40965424z^{-1} + 0.20482712z^{-2}}{1 - 0.53153089z^{-1} + 0.35083938z^{-2}}, \quad (6\text{-}113)$$

and there we are. Now, by inspection of Eq. (6-113), we get the time-domain expression for our IIR filter as

$$y(n) = 0.20482712 + 0.40965424 \cdot x(n-1) + 0.20482712 \cdot x(n-2) + 0.53153089 \cdot y(n-1) - 0.35083938 \cdot y(n-2). \quad (6\text{-}114)$$

The frequency magnitude response of our bilinear transform IIR design example is shown as the dark curve in Figure 6-33(a), where, for

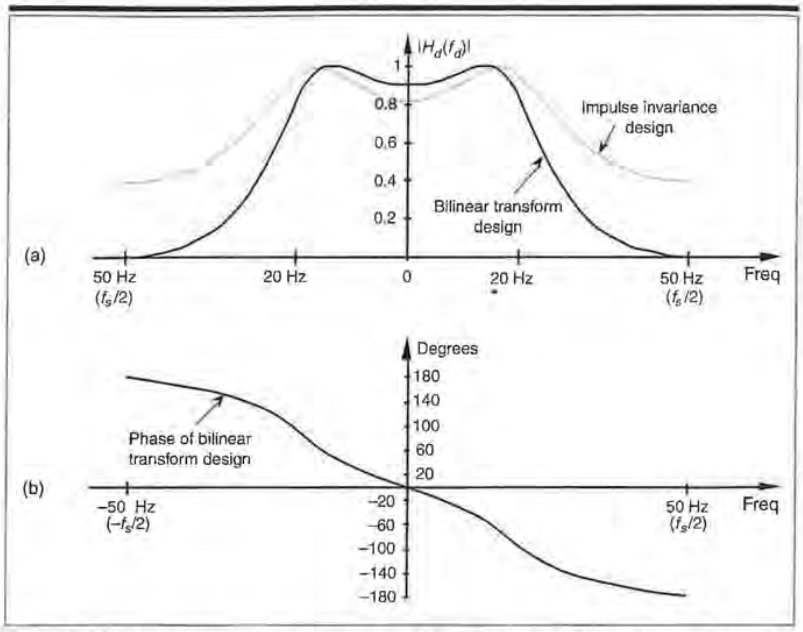


Figure 6-33 Comparison of the bilinear transform and impulse invariance design IIR filters: (a) frequency magnitude responses; (b) phase of the bilinear transform IIR filter.

comparison, we've shown the result of that impulse invariance design example as the shaded curve. Notice how the bilinear transform designed filter's magnitude response approaches zero at the folding frequency of $f_s/2 = 50$ Hz. This is as it should be—that's the whole purpose of the bilinear transform design method. Figure 6-33(b) illustrates the nonlinear phase response of the bilinear transform designed IIR filter.

We might be tempted to think that not only is the bilinear transform design method easier to perform than the impulse invariance design method, but that it gives us a much sharper roll-off for our low-pass filter. Well, the frequency warping of the bilinear transform method does compress (sharpen) the roll-off portion of a low-pass filter, as we saw in Figure 6-32, but an additional reason for the improved response is the price we pay in terms of the additional complexity of the implementation of our IIR filter. We see this by examining the implementation of our IIR filter as shown in Figure 6-34. Notice that our new filter requires five multiplications per filter output sample where the impulse invariance design filter in Figure 6-28(a) required only three multiplications per filter output sample. The additional multiplications are, of course, required by the additional feed forward z terms in the numerator of Eq. (6-113). These added

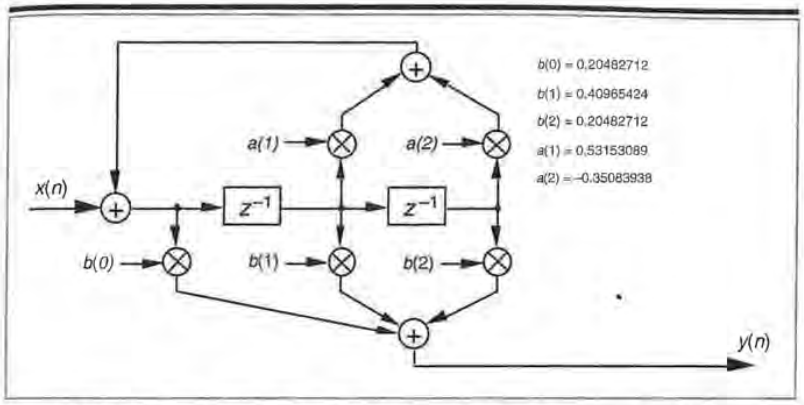


Figure 6-34 Implementation of the bilinear transform design example filter.

$b(k)$ coefficient terms in the $H(z)$ transfer function correspond to zeros in the z-plane created by the bilinear transform that did not occur in the impulse invariance design method.

Because our example prototype analog low-pass filter had a cutoff frequency that was $f_s/5$, we don't see a great deal of frequency warping in the bilinear transform curve in Figure 6-33. (In fact, Kaiser has shown that, when f_s is large, the impulse invariance and bilinear transform design methods result in essentially identical $H(z)$ transfer functions[19].) Had our cutoff frequency been a larger percentage of f_s, bilinear transform warping would have been more serious, and our resultant $|H_d(f_d)|$ cutoff frequency would have been below the desired value. What the pros do to avoid this is to *prewarp* the prototype analog filter's cutoff frequency requirement before the analog $H_c(s)$ transfer function is derived in Step 1.

In that way, they compensate for the bilinear transform's frequency warping before it happens. We can use Eq. (6-98) to determine the prewarped prototype analog filter low-pass cutoff frequency that we want mapped to the desired IIR low-pass cutoff frequency. We plug the desired IIR cutoff frequency ω_d in Eq. (6-98) to calculate the prototype analog ω_a cutoff frequency used to derive the prototype analog filter's $H_c(s)$ transfer function.

Although we explained how the bilinear transform design method avoided the impulse invariance method's inherent frequency response aliasing, it's important to remember that we still have to avoid filter input data aliasing. No matter what kind of digital filter or filter design method is used, the original input signal data must always be obtained using a sampling scheme that avoids the aliasing described in Chapter 2. If the original input data contains errors due to sample rate aliasing, no filter can remove those errors.

Our introductions to the impulse invariance and bilinear transform design techniques have, by necessity, presented only the essentials of those two design methods. Although rigorous mathematical treatment of the impulse invariance and bilinear transform design methods is inappropriate for an introductory text such as this, more detailed coverage is available to the interested reader[13–16]. References [13] and [15], by the way, have excellent material on the various prototype analog filter types used as a basis for the analytical IIR filter design methods. Although our examples of IIR filter design using the impulse invariance and bilinear transform techniques approximated analog low-pass filters, it's important to remember that these techniques apply equally well to designing bandpass and highpass IIR filters. To design a highpass IIR filter, for example, we'd merely start our design with a Laplace transfer function for the prototype analog highpass filter. Our IIR digital filter design would then proceed to approximate that prototype highpass filter.

As we have seen, the impulse invariance and bilinear transform design techniques are both powerful and a bit difficult to perform. The mathematics is intricate and the evaluation of the design equations is arduous for all but the simplest filters. As such, we'll introduce a third class of IIR filter design methods based on software routines that take advantage of *iterative optimization* computing techniques. In this case, the designer defines the desired filter frequency response, and the algorithm begins generating successive approximations until the IIR filter coefficients converge (hopefully) to an optimized design.

6.6 Optimized IIR Filter Design Method

The final class of IIR filter design methods we'll introduce are broadly categorized as optimization methods. These IIR filter design techniques were developed for the situation when the desired IIR filter frequency response was not of the standard low-pass, bandpass, or highpass form. When the desired response has an arbitrary shape, closed-form expressions for the filter's z-transform do not exist, and we have no explicit equations to work with to determine the IIR filter's coefficients. For this general IIR filter design problem, algorithms were developed to solve sets of linear, or nonlinear, equations on a computer. These software routines mandate that the designer describe, in some way, the desired IIR filter frequency response. The algorithms, then, assume a filter transfer function $H(z)$ as a ratio of polynomials in z and start to calculate the filter's frequency response. Based on some error criteria, the algorithm begins iteratively adjusting the filter's coefficients to minimize the error between the desired and the actual filter frequency response. The process ends when the error cannot

be further minimized, or a predefined number of iterations has occurred, and the final filter coefficients are presented to the filter designer. Although these optimization algorithms are too mathematically complex to cover in any detail here, descriptions of the most popular optimization schemes are readily available in the literature[14,16,20–25].

The reader may ask, "If we're not going to cover optimization methods in any detail, why introduce the subject here at all?" The answer is that if we spend much time at all designing IIR filters, we'll end up using optimization techniques in the form of computer software routines most of the time. The vast majority of commercially available digital signal processing software packages include one or more IIR filter design routines that are based on optimization methods. When a computer-aided design technique is available, filter designers are inclined to use it to design the simpler low-pass, bandpass, or highpass forms even though analytical techniques exist. With all due respect to Laplace, Heaviside, and Kaiser, why plow through all the z-transform design equations when the desired frequency response can be applied to a software routine to yield acceptable filter coefficients in a few seconds?

As it turns out, using commercially available optimized IIR filter design routines is very straightforward. Although they come in several flavors, most optimization routines only require the designer to specify a few key amplitude and frequency values, the desired order of the IIR filter (the number of feedback taps), and the software computes the final feed forward and feedback coefficients. In specifying a low-pass, IIR filter for example, a software design routine might require us to specify the values for δ_p, δ_s, f_1, and f_2 shown in Figure 6-35. Some optimization design routines require the user to specify the order of the IIR

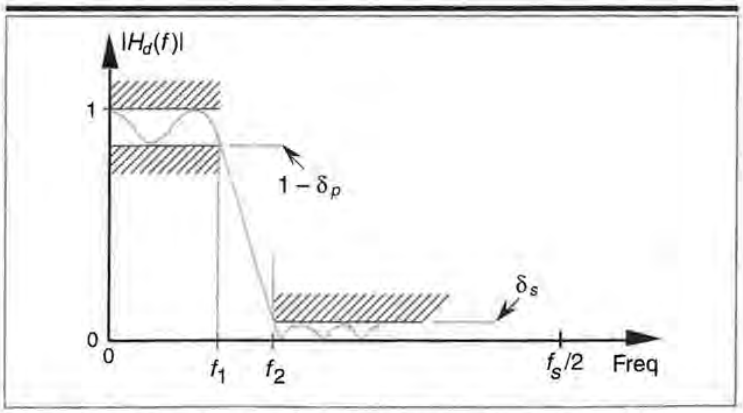


Figure 6-35 Example low-pass IIR filter design parameters required for the optimized IIR filter design method.

filter. Those routines then compute the filter coefficients that best approach the required frequency response. Some software routines, on the other hand, don't require the user to specify the filter order. They compute the minimum order of the filter that actually meets the desired frequency response.

6.7 Pitfalls in Building IIR Digital Filters

There's an old saying in engineering: "It's one thing to design a system on paper, and another thing to actually build one and make it work." (Recall the Tacoma Narrows Bridge episode!) Fabricating a working system based on theoretical designs can be difficult in practice. Let's see why this is often true for IIR digital filters.

Again, the IIR filter structures in Figures 6-18, 6-21(b), and 6-22 are called Direct Form implementations of an IIR filter because they're all equivalent to directly implementing the general time-domain expression for an Mth-order IIR filter given in Eq. (6-21). As it turns out, there can be stability problems and frequency response distortion errors when direct form implementations are used. Such problems arise because we're forced to represent the IIR filter coefficients and the results of intermediate filter calculations with binary numbers having a finite number of bits. These finite word length effects are particularly serious if a direct form IIR filter's $H(z)$ transfer function polynomial in z is of high order (i.e., a filter having a large number of delay elements). There are three major categories of finite word length errors that plague IIR filter implementations: coefficient quantization, overflow errors, and roundoff errors. Although Chapter 9 discusses the nature and effects of using finite word lengths in more detail, we'll briefly discuss those error types as they relate to IIR filters and see what precautions can be taken to minimize their effects.

Remember, all of our IIR filter calculations, thus far, have resulted in very accurate filter coefficient values. In fact, the fractional coefficient values in our previous IIR design examples were accurate to eight decimal digits as indicated in Eq. (6-114). Knowing that it takes four binary bits to store a single decimal digit, we can say that the coefficient values in Eq. (6-114) require the equivalent of 32 bits of accuracy. Consider the situation where we try to actually build the IIR filter in Figure 6-34, but we're constrained to store our $a(k)$ and $b(k)$ coefficients in four-bit hardware registers. Well, we can, indeed, represent the fractional coefficients with only one decimal digit, but they lose some of their precision. For example, we can represent Eq. (6-114)'s original $b(0) = 0.20482712$ coefficient as $b(0) = 0.2$ and lose some accuracy due to coefficient quantization.

Limited-precision coefficients will result in slight filter pole and zero shifting on the z plane, and a frequency magnitude response that may not meet our requirements.

For example, if we used only four-bits (one decimal digit) to represent the coefficients in Figure 6-34, our reduced-accuracy filter magnitude response is shown by the solid curve in Figure 6-36. For contrast, we've also shown the original high-accuracy coefficient response as the shaded curve in Figure 6-36. Sure enough, using four-bit coefficients distorted our original second-order IIR frequency response. Had our filter cutoff frequency (20 Hz) been higher relative to the sample rate (100 Hz), coefficient quantization would result in a frequency response that was even more skewed from the original. As it turns out, higher order filters are very sensitive to coefficient accuracy and, for them, coefficient quantization is more likely to cause unacceptable frequency response distortion. An even bigger problem can occur in high-order polynomials of $H(z)$ when quantized coefficients are rounded up to the nearest quantization level. After factoring a high-order error-containing $H(z)$ polynomial into its single-order factors, this can result in actually moving a pole outside the unit circle, yielding an unstable filter.

Overflow, the second finite word length effect that troubles digital filters, is what happens when the result of an arithmetic operation is too large to be represented in the fixed-length hardware registers designed to contain that result. Because we perform so many additions when we implement IIR filters, overflow is always a potential problem. With no precautions being made to handle overflow, large nonlinearity errors

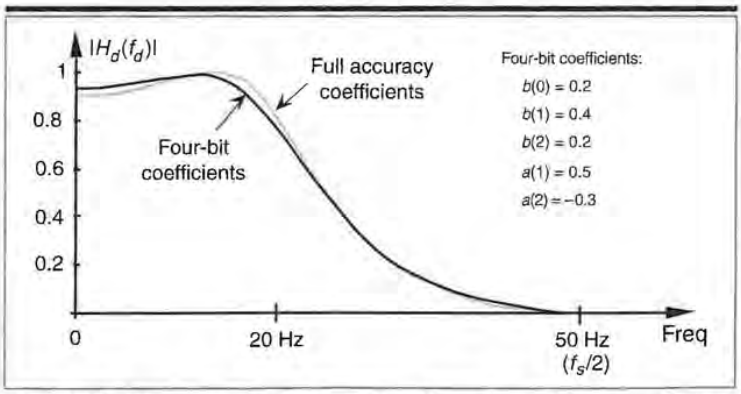


Figure 6-36 Comparison of the bilinear transform designed filter magnitude responses using full-accuracy coefficients (shaded curve) and four-bit coefficients (solid curve).

can result in our filter output samples—often in the form of *overflow oscillations*.

The most common way of dealing with binary overflow errors is called roundoff, or rounding, where a data value is represented by, or rounded off to, the b-bit binary number that's nearest the unrounded data value. It's usually valid to treat roundoff errors as a random process, but conditions occur in IIR filters where rounding can cause the filter output to oscillate forever, even when the filter input sequence is all zeros. This situation, going by the names *limit cycles* and *deadband effects*, has been well analyzed in the literature[26,27]. We can demonstrate limit cycles by considering the second-order IIR filter in Figure 6-37(a) whose time-domain expression is

$$y(n) = x(n) + 1.3y(n-1) - 0.76y(n-2)\,. \tag{6-115}$$

Let's assume that this filter rounds the adder's output to the nearest integral value. If the situation ever arises where $y(-2) = 0$, $y(-1) = 8$, and $x(0)$ and all successive $x(n)$ inputs are zero, the filter output goes into endless oscillation, as shown in Figure 6-37(b). If this filter were to be used in an audio application, when the input signal went silent, the listener could end up hearing an audio tone instead of silence. The shaded line in Figure 6-37(b) shows the filter's stable response to this particular situation if no rounding is used. With rounding, however, this IIR filter certainly lives up to its name.

There are several ways to reduce the ill effects of quantization errors and limit cycles. We can increase the word widths of the hardware registers that contain the results of intermediate calculations. Because roundoff limit cycles affect the least significant bits of an arithmetic result, larger word sizes diminish the impact of limit cycles should they occur. To avoid filter input sequences of all zeros, some practitioners add a *dither sequence* to the filter's input signal sequence. A dither sequence is a sequence of low-amplitude pseudorandom numbers that interferes with an IIR filter's roundoff error generating tendency, allowing the filter output to reach zero should the input signal remain at zero. Dithering, however, decreases the effective signal-to-noise ratio of the filter output[11]. Finally, to avoid limit cycle problems, we can just use an FIR filter. Because FIR filters, by definition, have finite-length impulse responses and have no feedback paths, they cannot support output oscillations of any kind.

As for overflow oscillations, we can eliminate them if we increase the word width of hardware registers, so that overflow never takes place in the IIR filter. Alternatively, it has been shown that the effects of overflow oscillations can be avoided by modifying hardware adders so that their

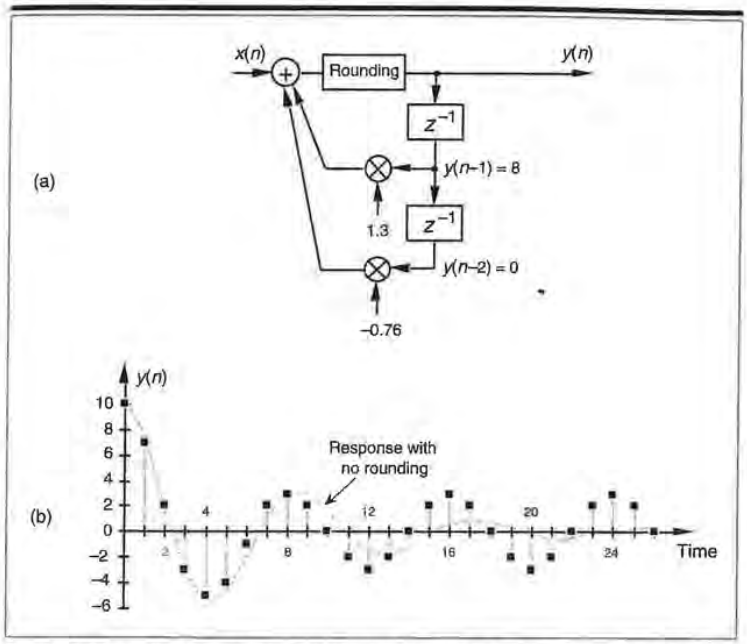


Figure 6-37 Limit cycle oscillations due to rounding in an IIR filter: (a) second-order IIR filter with rounding taking place following the adder; (b) one possible time-domain response of the IIR filter.

sum saturates to a fixed limit when an overflow condition is detected[28, 29]. It may be useful for the reader to keep in mind that, when the signal data is represented in two's complement arithmetic, multiple summations resulting in intermediate overflow errors cause no problems if we can guarantee that the final magnitude of the sum of the numbers is not too large for the final accumulator register. This potential relief from overflow errors, as discussed under the data overflow topic in Section 9.3, is one reason for the popularity of the two's complement format for binary numbers. Some digital filters incorporate additional control circuitry to detect overflow conditions and allow the designer to control (by scaling down) the amplitudes of signals at critical points in the system[14,17]. If your budget can afford floating-point hardware, it has been shown that both standard floating-point and block floating-point data formats can greatly reduce the errors associated with overflow oscillations and limit cycles[30]. (We discuss floating-point number formats in Section 9.4)

Perhaps the most popular technique for minimizing the errors associated with finite word widths is to design filters comprising a cascade string, or parallel combination, of low-order filters. The next section tells us why.

6.8 Cascade and Parallel Combinations of Digital Filters

In practice, we're likely to encounter multiple digital filters connected in cascade or parallel like those in Figure 6-38. As indicated in Figure 6-38(a), the resultant transfer function of two cascaded filter transfer functions is the product of those functions, or

$$H_{cascade}(z) = H_1(z)H_2(z) . \quad (6\text{-}116)$$

If the filters in Figure 6-38(a) are linear, we can swap their order, having $H_2(z)$ precede $H_1(z)$, with no change in the $Y(z)$ output. By the way, this property of the combined transfer function equaling the product of the two individual transfer functions applies to any combination of linear time invariant IIR and FIR filters. As shown in Figure 6-38(b), the combined transfer function of two parallel filters is the sum of their transfer functions, or

$$H_{parallel}(z) = H_1(z) + H_2(z) . \quad (6\text{-}117)$$

Hardware filter designers routinely partition high-order digital filters into a string of lower order filters arranged in cascade because the cascaded system usually requires fewer multiplications for a given filter frequency response. It's common practice to subdivide a high-order digital

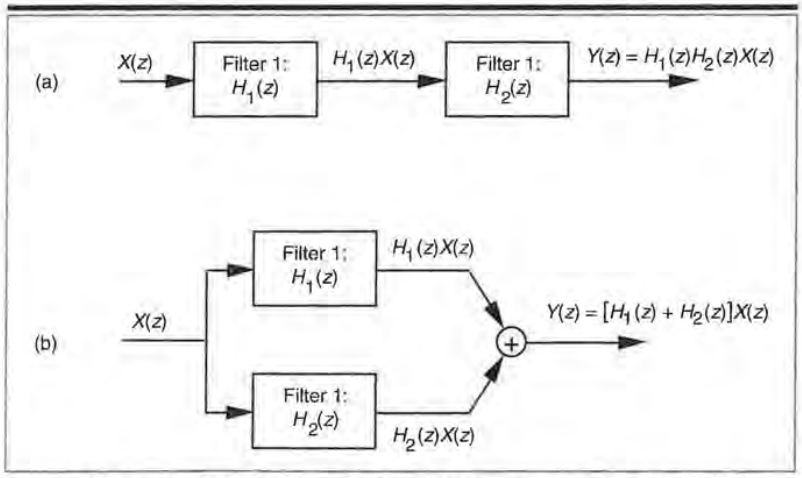


Figure 6-38 Combinations of two digital filters: (a) cascaded filters; (b) parallel filters.

filter into multiple second-order building blocks because second-order filters are easier to design, are less susceptible to coefficient quantization and roundoff errors, and their implementations allow easier data word scaling to reduce the potential overflow effects of data word size growth.

Optimizing the partitioning of a high-order filter into multiple second-order filter sections is a challenging task, however. For example, say we have the sixth-order filter in Figure 6-39(a) that we want to partition into three second-order sections. In factoring the sixth-order filter's $H(z)$ polynomial, we could get up to three separate sets of feedback coefficients in the factored $H(z)$ numerator: a'(n), a"(n), and a'"(n). Likewise, we could have up to three separate sets of feed forward coefficients in the factored

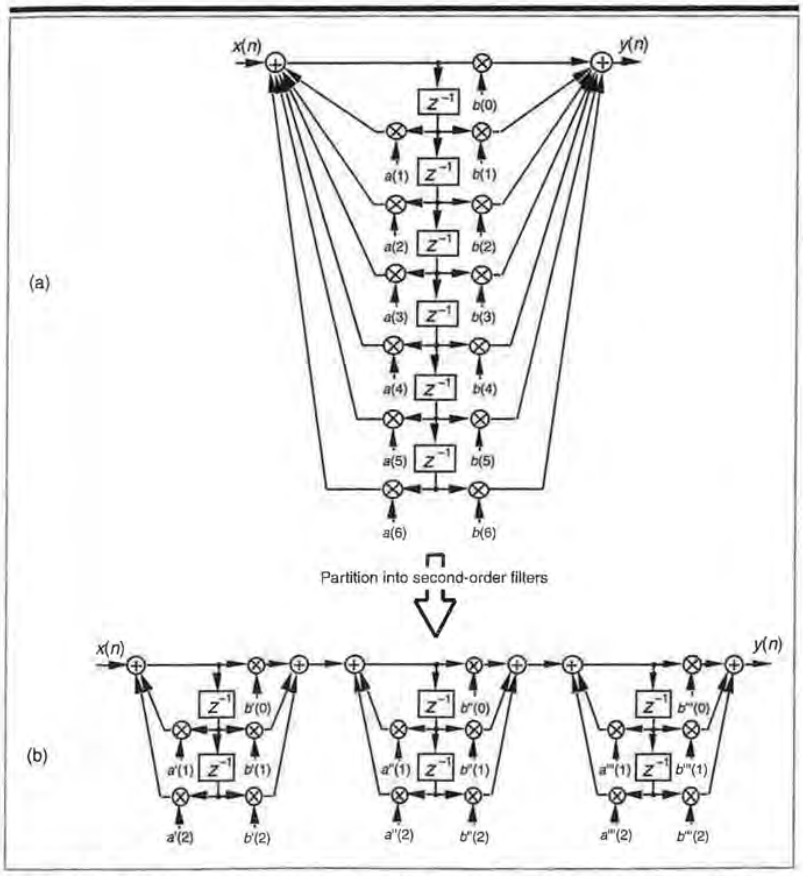


Figure 6-39 IIR filter partitioning: (a) original sixth-order IIR filter; (b) three second-order sections.

denominator: $b'(n)$, $b''(n)$, and $b'''(n)$. Because there are three second-ordered sections, there are three factorial, or six, ways of *pairing* the sets of coefficients. Notice in Figure 6-39(b) how the first section uses the $a'(n)$ and $b'(n)$ coefficients, and the second section uses the $a''(n)$ and $b''(n)$ coefficients. We could just as well have interchanged the sets of coefficients so that the first second-order section uses the a'(n) and b''(n) coefficients, and the second section uses the a''(n) and b'(n) coefficients. So there are six different mathematically equivalent ways of combining the sets of coefficients. Add to this the fact that, for each different combination of low-order sections, there are three factorially distinct ways those three separate second-order sections can be arranged in cascade.

This means that if we want to partition a $2M$-order IIR filter into M distinct second-order sections, there are M factorial squared, or $(M!)^2$, ways to do so. As such, there are then $(3!)^2 = 36$ different cascaded filters we could obtain when going from Figure 6-39(a) to 6-39(b).† To further complicate this filter sectioning problem, the errors due to quantization will, in general, be different for each possible filter combination. Although this subject is outside the scope of this introductory text, ambitious readers can find further material on optimizing cascaded filter sections in references [19,27], and in part 3 of reference [31].

There's a considerable amount of material in the literature concerning finite word effects as they relate to digital filters. (References [14,16] and [19] discuss quantization noise effects in some detail as well as providing extensive bibliographies on the subject.) Although these books and papers provide useful guidelines in predicting and minimizing quantization errors, prudent designers eventually resort to computer simulations of filter designs. This way they can vary filter hardware characteristics, such as coefficient word widths, accumulator register sizes, sequencing of cascaded sections, and input signal sets. This experimental approach is attractive because it's both reliable and economical.

6.9 A Brief Comparison of IIR and FIR Filters

The question naturally arises as to which filter type, IIR or FIR, is best suited for a given digital filtering application. That's not an easy question to answer, but we can point out a few factors that should be kept in mind. First, we can assume that the differences in the ease of design between the two filter types are unimportant. There are usually more important performance and implementation properties to consider than

† These combinations can get very large. For example, there are $(5!)^2 = 14{,}400$ ways to combine five second-order filters to implement a ten-order IIR filter.

design difficulty when choosing between an IIR and an FIR filter. One design consideration that may be significant is the IIR filter's ability to simulate a predefined prototype analog filter. FIR filters do not have this design flexibility.

From a hardware standpoint, with so many fundamental differences between IIR and FIR filters, our choice must be based on those filter characteristics that are most and least important to us. For example, if we needed a filter with exactly linear phase, then an FIR filter is the only way to go. On the other hand, if our design required that a filter accept very high data rates and slight phase nonlinearity is tolerable, we might lean toward IIR filters with their reduced number of necessary multipliers per output sample. Table 6-1 presents a brief comparison between IIR and FIR filters based on several different performance and implementation properties.

Table 6-1 IIR and FIR Filter Characteristics Comparison

Characteristic	*IIR*	*FIR*
Number of necessary multiplications	Least	Most
Sensitivity to filter coefficient quantization	Can be high for Direct Form*	Very low
Probability of overflow errors	Can be high for Direct Form*	Very low
Stability	Must be designed in	Guaranteed
Linear phase	No	Guaranteed **
Can simulate prototype analog filters	Yes	No
Required hardware memory	Least	Most
Hardware filter control complexity	Moderate	Simple
Availability of design software	Good	Very good
Ease of design or complexity of design software	Moderately complicated	Simple
Difficulty of quantization noise analysis	Most complicated	Least complicated
Supports adaptive filtering	Yes	Yes

* These problems can be minimized though cascade or parallel implementations.
** Guaranteed so long as the FIR coefficients are symmetrical.

References

[1] Churchill, R. V. *Modern Operational Mathematics in Engineering*, McGraw-Hill, New York, 1944, pp. 307–34.

[2] Aseltine, J. A. *Transform Method in Linear System Analysis*, McGraw-Hill, New York, 1958, pp. 287–92.

[3] Nixon, F. E. *Handbook of Laplace Transformation, Tables and Examples*, Prentice-Hall, Englewood Cliffs, New Jersey, 1960.

[4] Kaiser, J. F. "Digital Filters," in *System Analysis by Digital Computer*. Ed. by F. F. Kuo and J. F. Kaiser, John Wiley and Sons, New York, 1966, pp. 218–77.

[5] Kaiser, J. F. "Design Methods for Sampled Data Filters," Chapter 7, in *S1963 Proc. 1st Allerton Conference*, pp. 221-36.

[6] Ragazzini, J. R. and Franklin, G. F. *Sampled-Data Control Systems*, McGraw-Hill, New York, 1958, pp. 52–83.

[7] Milne-Thomson, L. M. *The Calculus of Finite Differences*, Macmillan, London, 1951, pp. 232–51.

[8] Truxal, J. G. 1955. *Automatic Feedback Control System Synthesis*, McGraw-Hill, New York, 1955, pp. 283.

[9] Blackman, R. B. *Linear Data-Smoothing and Prediction in Theory and Practice*, Addison-Wesley, Reading, Mass., 1965, pp. 81–84.

[10] Gold, B. and Jordan, K. L., Jr. "A Note on Digital Filter Synthesis," *Proceedings of the IEEE*, Vol. 56, October 1968, pp. 1717.

[11] Rabiner, L. R., et al. "Terminology in Digital Signal Processing," *IEEE Trans. on Audio and Electroacoustics*, Vol. AU-20, No. 5, December 1972, pp. 327.

[12] Stearns, S. D. *Digital Signal Analysis*, Hayden Book Co. Inc., Rochelle Park, New Jersey, 1975, pp. 114.

[13] Stanley, W. D., et al., *Digital Signal Processing*, Reston Publishing Co. Inc., Reston, Virginia, 1984, pp. 191.

[14] Oppenheim, A. V., and Schafer, R. W. *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989, pp. 406.

[15] Williams, C. S. *Designing Digital Filters*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986, pp. 166–86.

[16] Rabiner, L. R., and Gold, B. *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975, pp. 216.

[17] Johnson, M. "Implement Stable IIR Filters Using Minimal Hardware," *EDN*, 14 April 1983.

[18] Oppenheim, A. V., Willsky, A. S., and Young, I. T. *Signals and Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983, pp. 659.

[19] Kaiser, J. F. "Some Practical Considerations in the Realization of Linear Digital Filters," *Proc. Third Annual Allerton Conference on Circuit and System Theory*, 1965, pp. 621-33

[20] Deczky, A. G. "Synthesis of Digital Recursive Filters Using the Minimum P Error Criterion," *IEEE Trans. on Audio and Electroacoustics*, Vol. AU-20, No. 2, October 1972, pp. 257.

[21] Steiglitz, K. "Computer-Aided Design of Recursive Digital Filters," *IEEE Trans. on Audio and Electroacoustics*, Vol. 18, No. 2, 1970, pp. 123.

[22] Richards, M. A. "Application of Deczky's Program for Recursive Filter Design to the Design of Recursive Decimators," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-30, October 1982, pp. 811.

[23] Parks, T. W., and Burrus, C. S. *Digital Filter Design*, John Wiley and Sons, New York, 1987, pp. 244.

[24] Rabiner, L., Graham, Y., and Helms, H. "Linear Programming Design of IIR Digital Filters with Arbitrary Magnitude Functions," *IEEE Trans. on Acoustics, Speech, and Signal Processing.*, Vol. ASSP-22, No. 2, April 1974, pp. 117.

[25] Friedlander, B., and Porat, B. "The Modified Yule-Walker Method of ARMA Spectral Estimation," *IEEE Trans. on Aerospace Electronic Systems*, Vol. AES-20, No. 2, March 1984, pp. 158-73.

[26] Jackson, L. B. "On the Interaction of Roundoff Noise and Dynamic Range and Dynamic Range in Digital Filters," *Bell System Technical Journal*, Vol. 49, February 1970, pp. 159-84.

[27] Jackson, L. B. "Roundoff Noise Analysis for Fixed-Point Digital Filters Realized in Cascade or Parallel Form," *IEEE Trans. Audio Electroacoustics*, Vol. AU-18, June 1970, pp. 107-22.

[28] Sandberg, I. W. "A Theorem Concerning Limit Cycles in Digital Filters," *Proc. Seventh Annual Allerton Conference on Circuit and System Theory*, Monticello, Illinois, October 1969.

[29] Ebert, P. M., et al. "Overflow Oscillations in Digital Filters," *Bell Sys. Tech. Journal*, Vol. 48, November 1969, pp. 2999–3020.

[30] Oppenheim, A. V. "Realization of Digital Filters Using Block Floating Point Arithmetic," *IEEE Trans. Audio Electroacoustics*, Vol. AU-18, June 1970, pp. 130-36.

[31] Rabiner, L. R., and Rader, C. M., Eds., *Digital Signal Processing*, IEEE Press, New York, 1972, pp. 361.

CHAPTER SEVEN

Advanced Sampling Techniques

Beyond the low-pass and bandpass sampling schemes discussed in Chapter 2 there are specialized sampling techniques that we're likely to encounter in practice. In this chapter we introduce several of those advanced sampling methods and show why and when they're used.

7.1 Quadrature Sampling

Many digital signal processing applications use the complex data format with its real (in-phase) and imaginary (quadrature phase) parts. These applications fall in the general category known as *quadrature processing*. For example, the phase preservation characteristic of complex data representation is exploited in high data rate digital communications systems, radar systems, and time difference of arrival (TDOA) processing in radio direction-finding schemes[1,2]. The enhanced phase measurement capabilities of quadrature processing are used in coherent pulse measurement systems, antenna beamforming applications, and single sideband modulators[3–6]. Complex data representation allows us to realize additional processing power through the coherent measurement of the phase of sinusoids comprising an input signal. To obtain the time-domain representation of a continuous signal using complex notation, the signal must be digitized by a technique called *quadrature sampling*.[†]

[†] Quadrature sampling goes by various other names in the literature, such as vector demodulation, complex demodulation, complex down-conversion, quadrature heterodyning, I/Q sampling, and complex sampling[7–9]. Care must be exercised here from a semantic viewpoint. There's a testing scheme referred to as *coherent sampling*, used to characterize the dynamic performance of A/D converters, that is unrelated to quadrature sampling[10,11]. This clever A/D testing technique is discussed in Section 10.9.

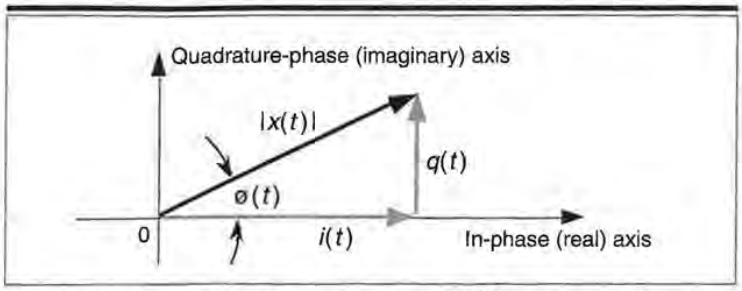


Figure 7-1 In-phase $i(t)$ and quadrature-phase $q(t)$ components of a continuous signal $x(t)$.

To illustrate this idea, let's consider a continuous time-domain signal $x(t)$ that has some instantaneous magnitude at a phase angle of $\text{ø}(t)$. This signal can, equally well, be defined by its in-phase $i(t)$ and quadrature-phase $q(t)$ components as shown in Figure 7-1. We could use the low-pass, or bandpass, sampling schemes in Chapter 2 to obtain discrete samples representing $x(t)$ and then calculate the $i(t)$ and $q(t)$ components. Quadrature sampling, on the other hand, allows us to obtain the discrete samples representing $i(t)$ and $q(t)$ directly. Thus, quadrature sampling produces two separate sampled data sequences, one sequence representing the in-phase and one sequence representing the quadrature-phase components of the original continuous signal $x(t)$.

Let's keep two things in mind as we begin our discussion of quadrature sampling. First, in studying quadrature processing, we'll encounter frequency-domain spectra that are not symmetrical about the 0 Hz axis. Although frequency-domain symmetry is inherent in real signals, the complex nature of quadrature signals does not have this restriction. It's the lifting of this symmetry restriction that makes quadrature processing so powerful. Second, recall from Euler's relationships discussed in Appendix C that

$$\sin(2\pi ft) = \frac{e^{j2\pi ft} - e^{-j2\pi ft}}{2j} = \frac{-je^{j2\pi ft}}{2} + \frac{je^{-j2\pi ft}}{2}, \tag{7-1}$$

and

$$\cos(2\pi ft) = \frac{e^{j2\pi ft} + e^{-j2\pi ft}}{2} = \frac{e^{j2\pi ft}}{2} + \frac{e^{-j2\pi ft}}{2}, \tag{7-2}$$

reminding us that real sinusoids or cosinusoids can be represented as complex phasor quantities through the convention of negative frequency, that is, negative exponentials. With these thoughts in mind, we view the hybrid processing known as I/Q demodulation (or Weaver demodulation for those readers with experience in communications theory), shown in Figure 7-2(a).

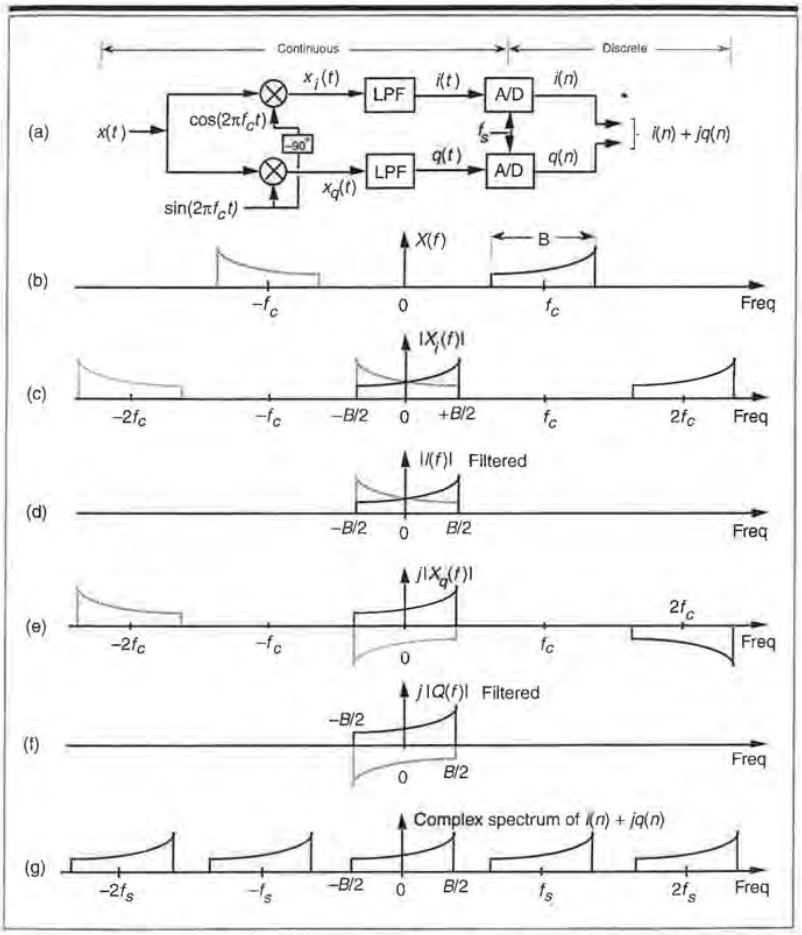


Figure 7-2 Quadrature bandpass sampling: (a) sampling block diagram; (b) continuous spectrum of continuous input $x(t)$; (c) continuous in-phase component of $x(t)$; (d) filtered in-phase component of $x(t)$; (e) continuous quadrature-phase component of $x(t)$; (f) filtered quadrature-phase component of $x(t)$; (g) sampled sum of the filtered in-phase component and the filtered quadrature-phase component of $x(t)$.

Consider a bandpass signal $x(t)$, centered at carrier frequency f_c Hz, that has the real spectrum $X(f)$ shown in Figure 7-2(b). When we mix $x(t)$ with a cosine wave of f_c Hz, the positive phasor portion of the cosine (that is, $e^{j2\pi f_c t}/2$) will translate the spectral components located at $-f_c$ (dashed curve) up to baseband centered at 0 Hz, resulting in what is sometimes called a zero-frequency IF format[4]. Likewise, the negative phasor portion of the cosine ($e^{-j2\pi f_c t}/2$) will translate the spectral components located at $+f_c$ (solid curve) down to baseband also centered at 0 Hz. The resultant continuous signal spectrum is shown as $X_i(f)$ in Figure 7-2(c) where we show the two translated spectra as overlaid, not summed, to see their individuality. $X_i(f)$ is low-pass filtered by an analog filter with a cutoff frequency slightly greater than $B/2$ Hz, called the in-phase component of $x(t)$, and shown as the $I(f)$ magnitude spectrum in Figure 7-2(d). (For the reader who's not at ease with the notion of negative frequencies, it's a good idea to review Appendix C to see why negative frequencies exist when we use complex signal notation.)

The continuous heterodyned product of $x(t)$ and the sinewave of f_c Hz is shown in Figure 7-2(e) where the spectra lying below the frequency axis result from the negative sign of the positive frequency $-je^{j2\pi f_c t}/2$ component in Eq. (7-2). This product is low-pass filtered and represents the quadrature-phase component of $x(t)$, shown as the $jQ(f)$ spectrum in Figure 7-2(f). The original real signal $x(t)$ is now represented in a complex format as $i(t) + jq(t)$. (To recover the real $x(t)$, we need only multiply $i(t)$ by $\cos(2\pi f_c t)$, multiply $q(t)$ by $\sin(2\pi f_c t)$, and sum the two products.)

Figure 7-2 helps remind us that real signals can be represented by a single sequence of sampled values and that their spectra show symmetry about zero Hz. Complex, or quadrature, signals must be represented by two sequences of sampled values, one sequence for the in-phase part and one for the quadrature part. Again, complex signal spectra are not required to be symmetrical about zero Hz. Returning to Figure 7-2(a), the two separate, continuous signals, $i(t)$ and $q(t)$, are digitized by an A/D converter at a sample rate of f_s samples/s. The resultant spectrum for the complex $i(n)+jq(n)$ is shown in Figure 7-2(g). Again, note the spectral replications every f_s Hz. So what's the big deal here? In Figure 7-2(g), notice that the sample rate need only be greater than twice $B/2$ to avoid aliasing.

In quadrature sampling f_s need only be greater than the signal bandwidth, as opposed to twice the signal bandwidth previously described in Chapter 2 for low-pass and bandpass sampling. This means that a given A/D converter can be used to sample signal bandwidths twice as wide with the quadrature sampling technique. Notice, however, that two low-pass filters and two A/D converters are necessary with this scheme.

What we have is two A/D converters operating at half the equivalent first-order bandpass sampling rate. Are we violating the Nyquist criterion? Not really—we still have two discrete sample values being generated for every cycle in the bandwidth of our original signal. So, when we need to digitize very wide bandwidths, the quadrature sampling scheme is attractive, particularly, when our goal is spectral analysis using the FFT. When the two data streams representing the in-phase and quadrature-phase components are applied to a complex radix-2 FFT algorithm, the FFT operates more efficiently than when its inputs are real signals[3,12]. (That particular topic is discussed in Section 10.5.)

7.2 Quadrature Sampling with Digital Mixing

To obtain quadrature samples in practice, the process shown in Figure 7-2(a) is difficult to implement without errors. Over moderately wide bandwidths, it's not easy to configure two analog signal paths that have identical frequency responses (gain and phase)—phase differences of a few degrees are common. Moreover, large DC components are routinely induced by the mixers in the $f_i(t)$ and $f_q(t)$ signals before filtering. Finally, if the low-pass filters are active devices, a kind of continuous noise known as *1/f noise* from the semiconductor components in the mixers/filters induces increased noise energy near 0 Hz.

These disadvantages can be avoided with digital mixers and filters where the sample rate is four times the carrier frequency of the bandpass signal[13–16]. This process is illustrated in Figure 7-3(a). The input signal $x(t)$ from Figure 7-2(a), with the spectrum shown in Figure 7-3(b), is digitized at a sample rate of $f_s = 4f_c$ providing the discrete samples $x(nt_s)$ whose spectrum, $X(m)$, is shown in Figure 7-3(c). The quadrature mixing with $f_c = f_s/4$, to center the input signal's in-phase and quadrature phase components about 0 Hz, is performed digitally with the results of the in-phase replicated components depicted in Figure 7-3(d) and the low-pass filtered in-phase spectrum in Figure 7-3(e). Although the A/D converter must sample at four times the carrier frequency, the difficulties in signal-path phase and amplitude matching are eliminated because only one A/D converter is necessary. In addition, the DC bias problems associated with analog signal mixing to zero Hz are avoided.

The real advantage of this technique is how easy the quadrature mixing by f_c can be performed digitally. For example, the input sampled values $x(nt_s)$ can be multiplied by the repetitive four-element sequence 1, 0, –1, 0, etc., to provide $i(n)$. Likewise, $x(nt_s)$ is multiplied by the four-element sequence 0, 1, 0, –1, to generate the $q(n)$ samples. The trick is that

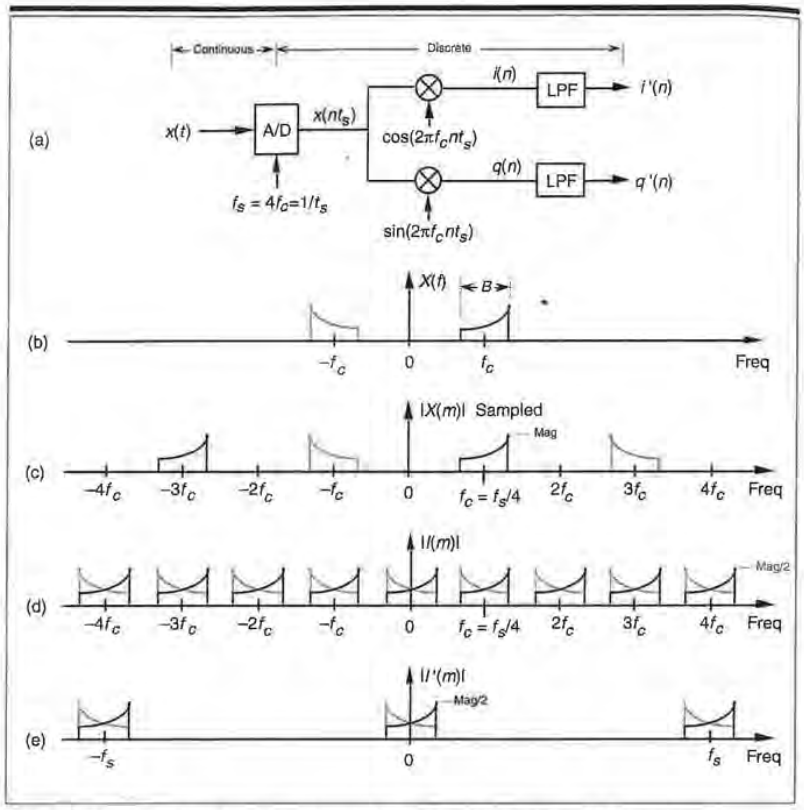


Figure 7-3 Quadrature bandpass sampling with digital mixing: (a) quadrature sampling block diagram; (b) spectrum of continuous bandpass signal $x(t)$; (c) discrete spectrum of sampled bandpass signal $x(nt_s)$; (d) frequency-translated $|I(m)|$ discrete spectrum of in-phase component of $x(nt_s)$; (e) low-pass filtered $|I'(m)|$ spectrum of in-phase component.

these two orthogonal ones sequences translate the input signal's spectrum by exactly $f_s/4$, and that's why our original f_s was made equal to $4f_c$. This clever technique enables complex sampling without multiplications because the mixing is implemented merely by sign changes on $x(nt_s)$.

If we implement this digital mixing process, we'll find that the spectral replication period in Figure 7-3(d) is half what is was in Figure 7-3(c). This comes about from the convolution of the real cosine's positive and negative frequency components with $|X(m)|$ in Figure 7-3(c). To use this digital mixing scheme, we must ensure that the condition $B < f_c = f_s/4$ holds true. In addition to the reduced spectral replication periodicity, we also

need to be aware that the DFT magnitude of the frequency-translated $i(n)$, $|I(m)|$, is related to the DFT magnitude of $x(nt_s)$, $|X(m)|$, by

$$|I(m)|_{1,0,-1,0} = \frac{|X(m)|}{2} \quad . \tag{7-3}$$

This is indicated by the Mag and Mag/2 factors in Figures 7-3(c) and (d). Likewise, the DFT magnitude of the frequency-translated $q(n)$, $|Q(m)|$, is related to the DFT magnitude $|X(m)|$, by

$$|Q(m)|_{0,1,0,-1} = \frac{|X(m)|}{2} \quad . \tag{7-4}$$

So what this all means is that the $i(n)$ and $q(n)$ signals, after translation down to zero Hz, have been reduced in amplitude by a factor of 2. We can find out why Figure 7-3(d)'s replication period and equations (7-3) and (7-4) are true by reviewing Section 10.1, which discusses several different digital quadrature mixing sequences in more detail.

Thus far, we've reviewed the fundamentals of, and an efficient technique for, quadrature bandpass sampling. Let's introduce three other advanced sampling topics, all of which fall in the general category of digital resampling.

7.3 Digital Resampling

The useful and fascinating process of digital resampling is a scheme for changing the effective sampling rate of a discrete signal *after* the signal has already been sampled. As such, resampling has many applications; it's used to minimize computations by reducing data rates when signal bandwidths are narrowed through low-pass filtering. Resampling is mandatory in real-time processing when a hardware digital processor must accept data at a rate that differs from that processor's fundamental clock rate. (A simple example of this is when a digital processor can accept input data at a 1-MHz rate and the incoming data is arriving at a rate of 1.5 MHz. In this case, the incoming data rate must be converted from a 1.5-MHz rate to a 1-MHz sample rate with a minimum of signal distortion.) In satellite and medical image processing, digital resampling is necessary for image enhancement, image scale change, and image rotation. Resampling is also used to reduce the computational complexity of certain narrowband digital filters.

We can define resampling as follows: Consider the process where a continuous signal y_c has been sampled at a rate of $f_{old} = 1/T_{old}$, and the discrete samples are $x_{old}(n) = y_c(nT_{old})$. Resampling is necessary when we need $x_{new}(n) = y_c(nT_{new})$, and direct sampling of the continuous y_c at the rate of $f_{new} = 1/T_{new}$ is not possible. For example, imagine we have an analog-to-digital (A/D) conversion system supplying a sample value every T_{old} seconds. But our processor can accept data only at a rate of one sample every T_{new} seconds. How do we obtain $x_{new}(n)$ directly from $x_{old}(n)$? One possibility is to digital-to-analog (D/A) convert the $x_{old}(n)$ sequence to regenerate the continuous y_c and, then, A/D convert y_c at a sampling rate of f_{new} to obtain $x_{new}(n)$. Due to the spectral distortions induced by D/A followed by A/D conversion, this technique limits our effective dynamic range and is typically avoided in practice. Fortunately, accurate all-digital resampling schemes have been developed, as we shall see.

7.3.1 Resampling by Decimation

Sampling rate changes come in two flavors: rate decreases and rate increases. Decreasing the sampling rate is known as decimation. (The term *decimation* is somewhat of a misnomer because decimation originally meant to reduce by a factor of ten. Currently, *decimation* is the term used for reducing the sample rate by any integral factor.) When the sampling rate is being increased, the process is known as interpolation, i.e., estimating intermediate sample values. Because it's the simplest of the two rate-changing schemes, let's explore decimation first.

We can decimate, or downsample, a sequence of sampled values by a factor of D by retaining every Dth sample and discarding the remaining samples. Relative to the original sample rate, f_{old}, the new sample rate is

$$f_{new} = \frac{f_{old}}{D} . \tag{7-5}$$

For example, to decimate a sequence $x_{old}(n)$ by a factor of $D = 3$, we retain $x_{old}(0)$ and discard $x_{old}(1)$ and $x_{old}(2)$, retain $x_{old}(3)$ and discard $x_{old}(4)$ and $x_{old}(5)$, retain $x_{old}(6)$, and so on. So $x_{new}(n) = x_{old}(3n)$, where $n = 0, 1, 2$, etc. The result of this decimation process is identical to the result of originally sampling at a rate of $f_{new} = f_{old}/3$ to obtain $x_{new}(n)$. The spectral implications of decimation are what we should expect, as shown in Figure 7-4, where the spectrum of an original bandlimited continuous signal is indicated by the solid lines. Figure 7-4(a) shows the discrete replicated spectrum of $x_{old}(n)$, $X_{old}(m)$. With $x_{new}(n) = x_{old}(3n)$, the discrete spectrum

$X_{new}(m)$ is shown in Figure 7-4(b). Two important features are illustrated in Figure 7-4. First, $X_{new}(m)$ could have been obtained directly by sampling the original continuous signal at a rate of f_{new}, as opposed to decimating $x_{old}(n)$ by a factor of 3. And second, there is, of course, a limit to the amount of decimation that can be performed relative to the bandwidth of the original signal B. We must ensure that $f_{new} > 2B$ to prevent aliasing after decimation. When an application requires that f_{new} be less than $2B$, then $x_{old}(n)$ must be low-pass filtered before the decimation process is performed. An example of the need for low-pass filtering before decimation is shown in Figure 7-4(c). If the original signal has a bandwidth B, and we're interested in retaining only the band B', the signal between B' and $f_{new}/2$ must be low-pass filtered before the decimation

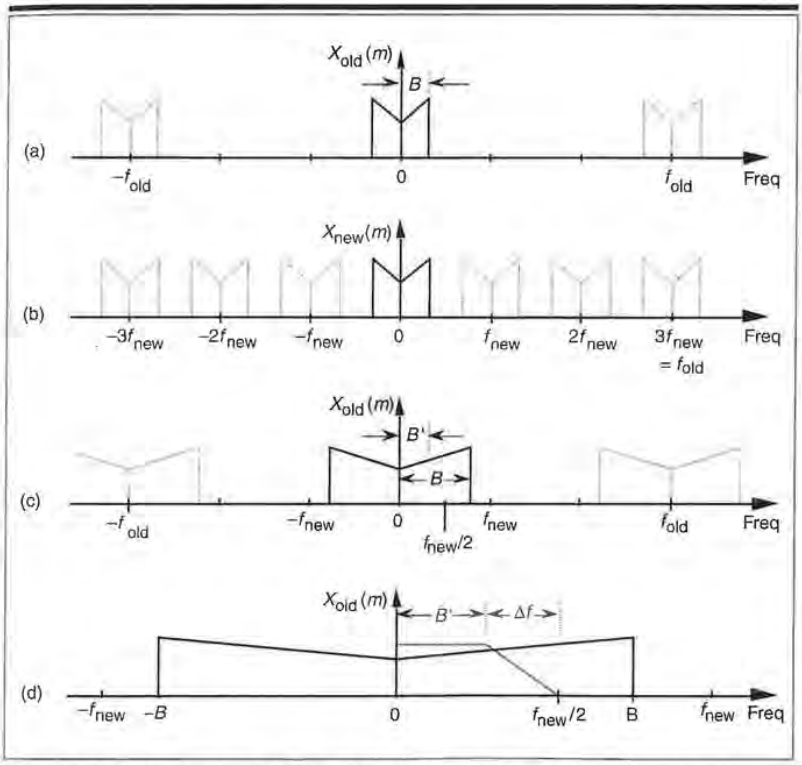


Figure 7-4 Decimation by a factor of three: (a) spectrum and replications of original signal; (b) spectrum of signal decimated by a factor of three; (c) example where only the bandwidth B' from the original bandwidth B is to be retained; (d) relationship of a low-pass filter's cutoff frequency relative to the bandwidth B'.

process is performed. Figure 7-4(d) shows this in more detail where the frequency response of the low-pass filter, shaded, must attenuate the signal amplitude above $f_{new}/2$. In practice, the direct form of FIR filters (the structure shown in Figure 5-13) is the prevailing choice for low-pass *decimation filters* because of the FIR filter's linear phase response[17].

When the desired decimation factor D is large, say $D > 10$, there is an important feature of the filter/decimation process to keep in mind. Significant computational savings may be had by implementing decimation in multiple stages. By way of example, let's assume that we have input data arriving at a sample frequency of 400 kHz, and we must decimate by a factor of $D = 100$ because our digital processor can only accept data at a rate of 4 kHz. Also, let's assume that the baseband frequency range of interest is from 0 to 1.8 kHz. So, with $f_{new} = 4$ kHz, we must filter out any signal above $f_{new}/2$ by having our filter transition band between 1.8 kHz and $f_{new}/2 = 2$ kHz. It's been shown that the number of stages S in the direct form of a FIR low-pass filter is proportional to the ratio of the original sample frequency over the filter transition band, Δf in Figure 7-4(d)[17,18]; that is,

$$S = k \cdot \frac{f_{old}}{(f_{new}/2 - B')} = k \cdot \frac{f_{old}}{\Delta f}, \tag{7-6}$$

where $2 < k < 4$ depending on the amount of filter passband and stopband ripple that can be tolerated. So for our case, if k is 3, for example, $S = 6000$. Think of it, a filter with 6000 stages! Fortunately there's a better way. Let's do something that appears to compound our problem and partition our decimation example into two stages: decimation by 50 followed by decimation by 2, as shown in Figure 7-5(a). We'll assume that the original input signal spectrum extends from zero Hz to something greater than 100 kHz, as shown in Figure 7-5(b). If the first low-pass filter LPF_1 has a cutoff frequency of 1.8 kHz and its stopband is defined to begin at 6 kHz, the output of the $D = 50$ decimator will have a spectrum, as shown in Figure 7-5(c), where our 1.8-kHz band of interest is shaded. Notice that there is aliasing of the signal between 2 kHz and 4 kHz—not to worry, LPF_2 will take care of this. When LPF_2 has a cutoff frequency of 1.8 kHz and its stopband is designed to be 2 kHz, the output of the $D = 2$ decimator will have our desired spectrum, shown in Figure 7-5(d). The point is that the total number of stages in the two low-pass filters, S_{Total}, is greatly reduced from 6000. From Eq. (7-6) for the combined LPF_1 and LPF_2 filters,

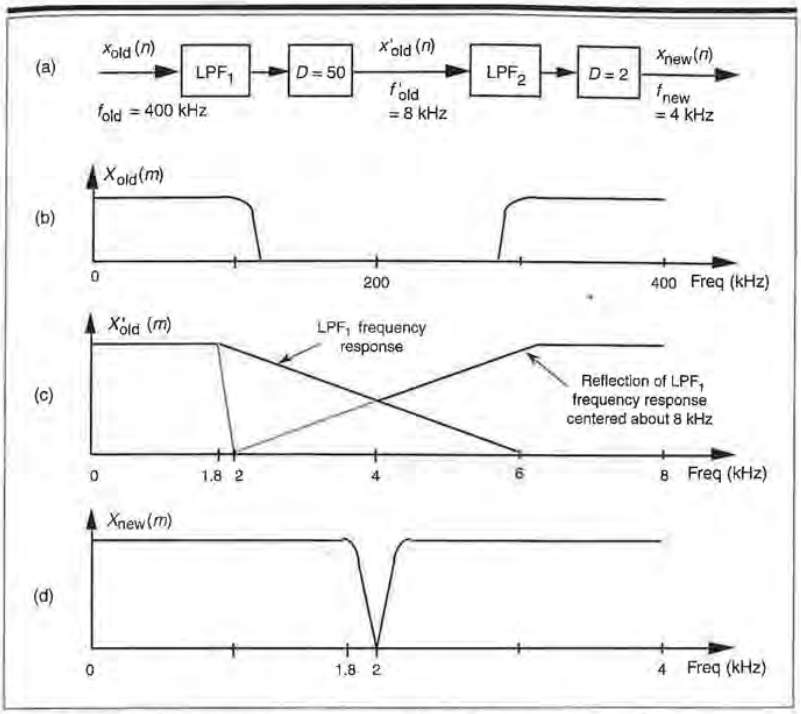


Figure 7-5 Multistage decimation: (a) decimation block diagram; (b) spectrum of original signal; (c) output of the $D = 50$ decimator; (d) output of the $D = 2$ decimator.

$$S_{Total} = S_{LPF_1} + S_{LPF_2} = 3 \cdot \frac{400}{(6-1.8)} + 3 \cdot \frac{8}{(2-1.8)} = 406 \text{ stages.} \qquad (7\text{-}7)$$

This is an impressive computational reduction. Reviewing Eq. (7-6) for each stage, in the first stage we see that, although $f_{old} = 400$ kHz remained constant, we increased Δf. In the second stage, both Δf and f_{old} were reduced. The fact to remember in Eq. (7-6) is that the ratio $f_{old}/\Delta f$ has a much more profound effect than k in determining the number of stages necessary in a low-pass filter. This example, although somewhat exaggerated, shows the kind of computational saving afforded by multistage decimation. Isn't it interesting that adding more processing stages to our original $D = 100$ decimation problem actually decreased the necessary computational requirement?

A sensible way to evaluate FIR decimation filter implementations or any filter implementation, for that matter, is to determine the necessary number of multiplications/second. Going back to our original $D = 3$ decimation example, let's assume $f_{old} = 60$ kHz and our decimation filter need have only $S = 5$ multiplier stages, as shown in Figure 7-6(a), with the filter coefficients being h_0 to h_4. Because the $x_{old}(n)$ samples are arriving at a rate of f_{old}, the low-pass filter requires C_D multiplications/s where

$$C_D = Sf_{old} = 5 \cdot 60{,}000 = 300{,}000 \text{ multiplications/s.} \quad (7\text{-}8)$$

In Figure 7-6(a), our decimation by three is symbolized by a rotary switch that rotates clockwise 120° each time an $x_{old}(n)$ sample arrives and only every third filter output sample is retained for $x_{new}(n)$. This structure is not so desirable because, for every 15 multiplications, we use only 5 products, an inefficient arrangement even when D is small. Fortunately, the switching (decimation) can be moved ahead of the multipliers, as illustrated in Figure 7-6(b), and we need perform only C_D' multiplications/s where

$$C_D' = \frac{C_D}{D} = \frac{Sf_{old}}{D} = 100{,}000 \text{ multiplications/s.} \quad (7\text{-}9)$$

By moving the decimation process ahead of the multipliers, we give the multipliers $D = 3$ times as long to perform a multiplication, and $x_{new}(n)$ is identical in the two filter structures. (If our odd-order FIR filter coefficients are symmetrical, so that $h(4) = h(0)$, and $h(3) = h(1)$, there's a clever way to further reduce the number of necessary multiplications, as described in Section 10.8.)

Let's step back for a moment to make sure we understand what's going on here. The rotating switches in Figure 7-6 are only conceptual. Again, the switch in Figure 7-6(a) means that we retain only every third output sample of x_{new}. In this case, the filter is operating at the input sample rate of f_{old} and we discard two out of every three filter output samples. In Figure 7-6(b), we don't even bother to calculate the two out of every three samples that are to be discarded; here, the filter is operating at the reduced output sample rate of f_{new} where we calculate only those output samples that we need. By the way, IIR filters cannot take advantage of the reduced computation scheme in Figure 7-6(b) because the various implementations of IIR filters mandate that they must always operate at the input sample rate f_{old}.

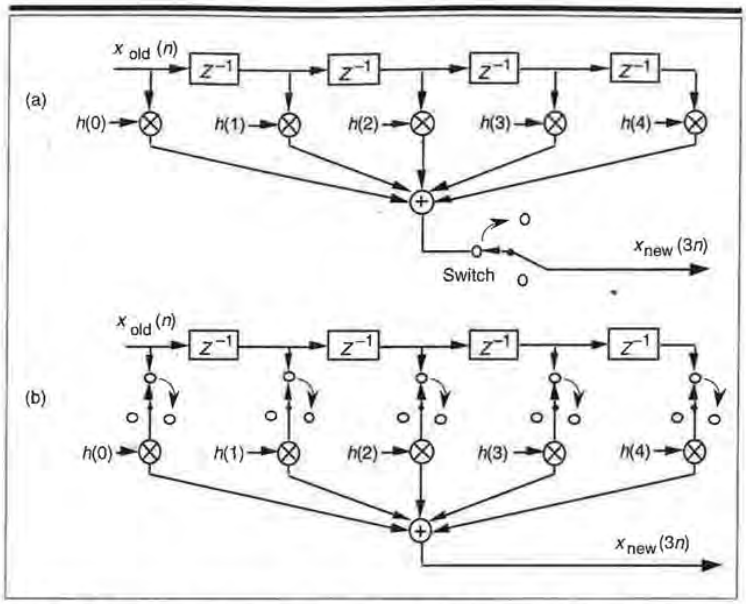


Figure 7-6 Decimation filter implementations: (a) five-stage FIR digital filter; (b) decimation taking place before the filter multiplications.

Before we leave the subject, it's interesting to realize that decimation is one of those rare processes that is not time invariant. From the very nature of its operation, we know that if we delay the input sequence by one sample, a downsampler will generate an entirely different output sequence. For example, if we apply an input sequence $x(n) = x(0), x(1), x(2), x(3), x(4)$, etc., to a downsampler and $D = 3$, the output $y(n)$ will be the sequence $x(0), x(3), x(6)$, etc. Should we delay the input sequence by one sample, our delayed $x'(n)$ input would be $x(1), x(2), x(3), x(4), x(5)$, etc. In this case, the delayed output sequence $y'(n)$ would be $x(1), x(4), x(7)$, etc., which is *not* a delayed version of $y(n)$. Thus, a decimator (downsampler) is not time invariant.

7.3.2 Resampling by Interpolation

As we said before, decimation is only part of the digital resampling story—let's now consider interpolation. Sample rate increase by interpolation is a bit more involved than decimation because, with interpolation, new sample values need to be calculated. Conceptually, interpolation

comprises generating a continuous curve that passes through our $x_{old}(n)$ sampled values, followed by resampling that curve at the new sample rate f_{new} to obtain the sequence $x_{new}(n)$. Of course, continuous curves can't exist inside a digital machine, so we'll just have to obtain $x_{new}(n)$ directly from $x_{old}(n)$. To increase a given sample rate or upsample by a factor of U, we have to calculate U–1 intermediate values between each sample in $x_{old}(n)$. The process is beautifully straightforward and best understood by way of an example.

Let's assume that we have the sequence $x_{old}(n)$, part of which is shown in Figure 7-7(a), and we want to increase its sample rate by a factor of 4, so, $U = 4$. The sequence's spectrum is provided in Figure 7-7(b) where only the signal spectrum between 0 Hz and $4f_{old}$ is shown. To upsample $x_{old}(n)$, by a factor of four, we must insert three zeros between each sample of $x_{old}(n)$ as shown in Figure 7-7(c), to create the new sequence $x'_{new}(n')$. (This insertion of zeros is called zero padding.) Notice that

$$x'_{new}(n') = x_{old}(n) \text{ , when } n' = 4n \text{ ;} \tag{7-10}$$

that is, the old sequence is now embedded in the new sequence. The insertion of the zeros establishes the sample index for the new sequence $x_{new}(n')$ where the interpolated values will be assigned. The spectrum of $x'_{new}(n')$, $X'_{new}(m)$, is shown in Figure 7-7(d) where $f_{new} = 4f_{old}$. In Figure 7-7(d), notice that $x_{old}(n)$'s spectrum is replicated $U = 4$ times in the frequency range of zero Hz to f_{new}. This shouldn't surprise us; $x_{old}(n)$'s spectrum was already replicated 4 times between 0 Hz and f_{new} in Figure 7-7(b). What we've done by adding the zeros is merely increase the effective sample frequency to $f_s = f_{new}$ in Figure 7-7(d). Likewise, we've also increased the new folding frequency by a factor of 4 to $2f_{old}$.

The final step in interpolation is to filter the $x'_{new}(n')$ sequence with a low-pass digital filter whose frequency response about zero Hz and f_{new} Hz is shown as the dashed lines in Figure 7-7(d). This low-pass filter is called an *interpolation filter*, and its output sequence is the desired $x_{new}(n')$, shown in Figure 7-7(e) while its output spectrum is given in Figure 7-7(f). Is that all there is to interpolation? Well, not quite—because we can't implement an ideal low-pass filter, $x_{new}(n')$ will not be an exact interpolation of $x_{old}(n)$. The error manifests itself as the unwanted images indicated by the low-level replications in Figure 7-7(f). With an ideal filter, these images would not exist. We can only approximate an ideal, low-pass filter, and the accuracy of our entire interpolation process depends on the frequency response of our filter approximation! As with decimation,

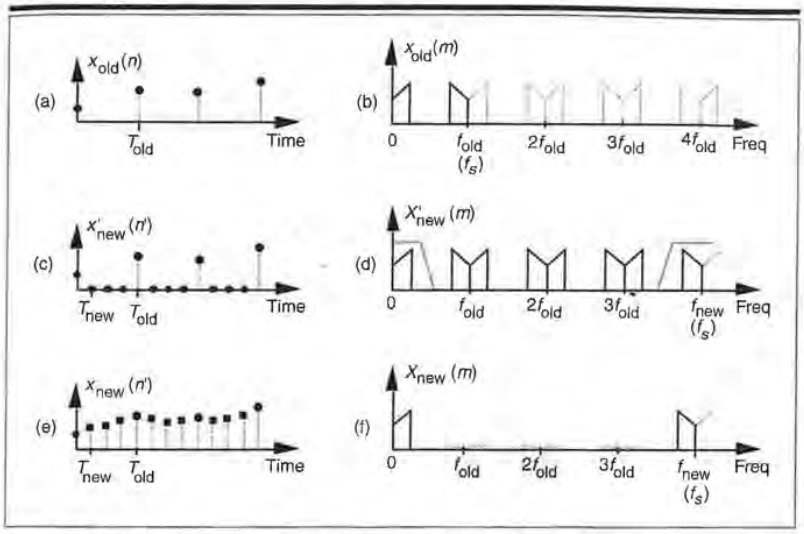


Figure 7-7 Interpolation by a factor of four: (a) original sampled sequence; (b) spectrum of original sequence; (c) zeros inserted in original sequence; (d) spectrum of sequence with inserted zeros; (e) output sequence of interpolation filter; (f) output spectrum of interpolation

interpolation can be thought of as, fundamentally, a low-pass filter design exercise, and the choice of interpolation filter structure deserves attention.

If the FIR interpolation filter design requires S filter stages, we need not perform S multiplications to get each of the $x_{new}(n')$ samples in Figure 7-7(e). By way of an example, returning to our $U = 4$ interpolation case, let's assume we've decided that we need a low-pass filter with $S = 19$ stages. Our example uses an FIR filter with an odd number of stages because this is the optimum structure used for interpolation filters[19]. The job of our low-pass filter is to convolve its impulse response with the $x'_{new}(n')$ sequence. Figure 7-8(a) shows the low-pass filter's impulse response being applied to a portion of the $x'_{new}(n')$ samples to calculate the ninth sample of $x_{new}(n')$, $x_{new}(9)$. The 19 filter coefficients are indicated by the Xs in Figure 7-8(a). With the dots in Figure 7-8(a) representing the $x'_{new}(n')$ sequence, we see that, although there are 19 Xs, only the bold Xs generate nonzero products that contribute to the convolution sum $x_{new}(9)$. The issue here is that we need not perform the multiplications associated with the zero-valued samples in $x'_{new}(n')$. In our example, we need only perform five multiplications to get $x_{new}(9)$. For the general case,

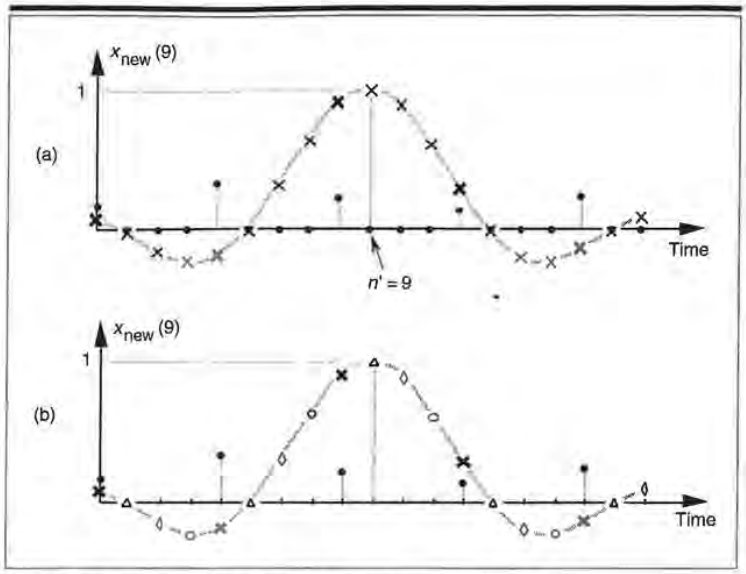


Figure 7-8 Nineteen-stage interpolation filter coefficients: (a) interpolation filter's impulse response shift position to calculate $x_{new}(9)$; (b) Xs mark the only filter coefficients that need be used to calculate $x_{new}(9)$.

the number of multiplications necessitated by an odd-order symmetrical interpolation FIR filter, for each sample of $x_{new}(n')$, cannot be expressed in a closed form for all possible values of S and U, but it can be shown that the minimum number of nonzero multiplications C_{Umin} is

$$C_{Umin} = \lceil \frac{S}{U} \rceil \text{ multiplications,} \tag{7-11}$$

where $\lceil\ \rceil$ means "the largest integer contained in." C_{Umin} is the number of $x'_{new}(n')$ values that will be used in the interpolation to determine each $x_{new}(n')$ sample. The number of necessary multiplications/s is $C_{Umin} \cdot f_{new}$.

So the good news is that the number of interpolation filter multiplications necessary is approximately equal to S/U, and that's quite a reduction in computational complexity. The bad news is that we still require that $S = 19$ filter coefficients be stored, and we can't take advantage of any filter coefficient symmetry to reduce the number of multiplications by a further factor of two as described in Section 10.8. Both of these minor

problems are the price we must pay to take advantage of Eq. (7-11), and they can be understood by examining Figure 7-8(b). To illustrate our $U = 4$ example, in Figure 7-8(b), we use the filter coefficients indicated by the Xs to calculate $x_{new}(9)$. When we slide the filter's impulse response to the right one sample, we use the coefficients indicated by the circles to calculate $x_{new}(10)$ because the nonzero values of $x'_{new}(n')$ will line up under the circled coefficients. Likewise, when we slide the impulse response to the right one more sample to calculate $x_{new}(11)$, we use the coefficients indicated by the diamonds. Finally, when we slide the impulse response to the right once more, we use the coefficients indicated by the triangles to calculate $x_{new}(12)$. Sliding the filter's impulse response once more to the right, we return to using the coefficients indicated by the Xs to calculate $x_{new}(13)$. You can see the pattern here—there are U, where $U = 4$ in our example, different sets of coefficients that are used to calculate $x_{new}(n')$ from the $x'_{new}(n')$ samples. Each time a new $x_{new}(n')$ sample value is to be calculated, we rotate one step through the sets of coefficients and use

the Δ coefficient set, when $n' = 0, 4, 8, 12, \ldots, 4n$,

the X coefficient set, when $n' = 1, 5, 9, 13, \ldots, 4n+1$,

the o coefficient set, when $n' = 2, 6, 10, 14, \ldots, 4n+2$, and

the $\Diamond$ coefficient set, when $n' = 3, 7, 11, 15, \ldots, 4n+3$.

This rotation through U sets of coefficients, where each set contains approximately S/U coefficients, is why interpolation filters are sometimes called *time-varying*, or *polyphase*, filters. At different times, we use different filter coefficients.

So, even though we don't have to perform $S = 19$ multiplications for each $x_{new}(n')$ value, we do have to have $S = 19$ filter coefficients stored and available to calculate each $x_{new}(n')$. (During the calculation of any given $x_{new}(n')$ sample, because the coefficients are not necessarily symmetrical, we can't take advantage of any symmetrical filter structure to reduce the number of multiplications by an additional factor of two as discussed in Section 10.8.) There's one final important feature of the FIR filter's coefficients indicated by the triangles. For our example, when the filter's impulse response peak is aligned above one of the nonzero values of $x'_{new}(n')$, notice that the other coefficients indicated by the remaining triangles have values of zero. This results in the agreeable characteristic that $x_{new}(n') = x'_{new}(n') = x'_{new}(4n)$, so that the output of the filter at the original sampling times is equal to the original samples.

7.3.3 Combining Decimation and Interpolation

While changing sampling rates through decimation or interpolation by integer factors can be useful, what can we do if we need a sample rate change that is not an integer? The good news is that we can implement sample rate changes by any rational fraction U/D with interpolation by a factor of U followed by decimation by a factor of D. Because the fraction U/D can be obtained as accurately as we want with the correct choice of integers U and D, we can change sample rates by almost any factor in practice. For example, a sample rate increase by a factor of 7.125 can be performed by an interpolation of $U = 57$ followed by a decimation of $D = 8$, because $7.125 = 57/8$.

This U/D sample rate change is illustrated by the processes shown in Figure 7-9(a). The really neat part here is that the computational burden of changing the sample rate by the combination of U/D is less than the sum of an individual interpolation followed by an individual decimation because we can combine the interpolation filter LPF_U and the decimation filter LPF_D into a single filter, shown as $\text{LPF}_{U/D}$ in Figure 7-9(b). The process in Figure 7-9(b) is normally called a sample rate converter because, if $U > D$ we have interpolation, and, when $D > U$, we have decimation. (The filter $\text{LPF}_{U/D}$ is sometimes referred to as an interpolation filter, regardless of the ratio U/D, and sometimes it's called a *multirate* filter.) $\text{LPF}_{U/D}$ serves two purposes for us now. First, it must sufficiently attenuate the interpolation images illustrated in Figure 7-7(f), so that, after decimation, they don't contaminate our desired signal beyond acceptable limits. Second, it must have a cutoff frequency that prevents aliasing of our desired signal images resulting from the final decimation.

To illustrate these filter requirements, let's assume we need to resample an input sequence by 4/3 or 1.25, so, $U = 4$, and $D = 3$. The original sequence $x_{\text{old}}(n)$ has the spectrum shown in Figure 7-9(c). The spectrum of the intermediate sequence $x'_{\text{new}}(n')$ after upsampling by 4 and low-pass filtering by $\text{LPF}_{U/D}$ is shown in Figure 7-9(d). The spectral result, after the $D = 3$ decimation to obtain the desired output sequence $x_{\text{new}}(n'')$, is shown in Figure 7-9(e). Notice that the frequency response of $\text{LPF}_{U/D}$ must be designed so that the stopband frequency, $f_{\text{stop}'}$ is less than $f_{\text{new}}/2$ to avoid aliasing after the decimation. Of course $\text{LPF}_{U/D}$ is a time-varying filter as described above for Figure 7-8. The attenuation of $\text{LPF}_{U/D}$ beyond f_{stop} must be great enough, so that the images (shown in Figure 7-9(d) at multiples of f_{old}) do not induce intolerable levels of noise when they're aliased by decimation into the final band of 0 to $f_{\text{new}}/2$ Hz. If the number of mul-

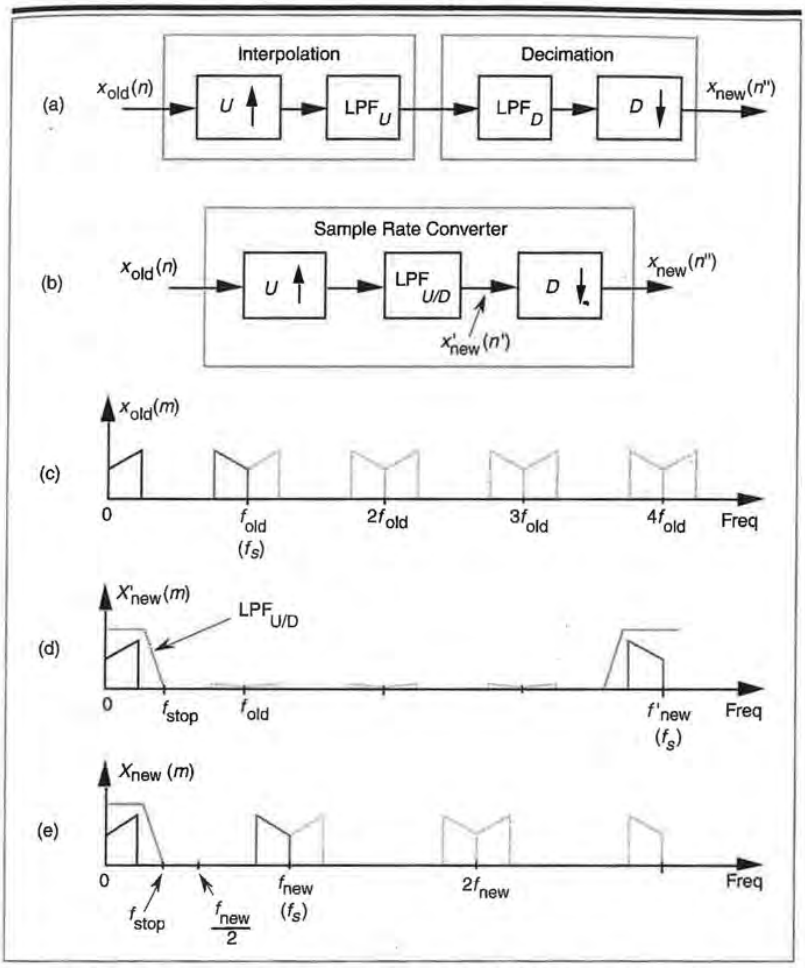


Figure 7-9 Sample rate conversion by a factor of 4/3: (a) combination of interpolation and decimation; (b) sample rate conversion with a single low-pass filter; (c) original discrete input sequence spectrum; (d) spectrum of the low-pass filter output; (e) final discrete spectrum after decimation by a factor of 3.

tiplication stages in $LPF_{U/D}$ is S, the computational complexity of our interpolator/decimator implementation is given by

$$C_{U/D} = \frac{S}{UD} \text{ multiplications/output sample.} \tag{7-12}$$

Because the input to $LPF_{U/D}$ is nonzero only for every Uth filter input sample, so, for each filter output sample we need only perform S/U multiplications. Moreover, our decimator discards all but every Dth filter output sample, so the actual number of multiplications necessary is $(S/U)/D$ multiplications per decimator output sample.

Again, our interpolator/decimator problem is an exercise in low-pass filter design, and all the knowledge and tools we have to design low-pass filters can be applied to this task. In software interpolator/decimator design, we want our low-pass filter algorithm to prevent aliasing images and be fast in execution time. For hardware interpolator/decimators, we strive to implement a design that optimizes the conflicting goals of high performance (minimum aliasing), simple architecture, high data throughput speed, and low power and parts count requirements.

This introduction to digital resampling has, by necessity, only touched the surface of this important signal processing technique. Fortunately for us, the excellent work of the first engineers and mathematicians to explore this subject is well documented in the literature. The inquisitive reader can probe further to learn how to choose the number of stages in a multistage process[17,20], the interrelated considerations of designing optimum FIR filters[17,21], the benefits of half-band FIR filters[18,22], when IIR filter structures may be advantageous[21], what special considerations are applicable to resampling in image processing[23–25], guidance in developing the control logic necessary for hardware implementations of resampling algorithms[21], how resampling improves the usefulness of commercial test equipment[26,27], and software development tools for designing multirate filters[28].

References

[1] Floyd, P., and Taylor, J. "Dual-Channel Space Quadrature-Interferometer System," *Microwave System Designer's Handbook*, Fifth Edition, Microwave Systems News, 1987.

[2] Hack, T. "IQ Sampling Yields Flexible Demodulators," *RF Design Magazine*, April 1991.

[3] Mirage Systems, "Automated-Radar-Measurement System for Pulsed I/Q Data Collection," *Microwave System News and Communications Technology*, December 1987.

[4] Chester, D. B., and Phillips, G. "Use DSP Filter Concepts in IF System Design," *Electronic Design*, July 11, 1994.

[5] Steyskal, H. "Digital Beamforming Antennas," *Microwave Journal*, January 1987.

[6] Maruta, R., and Tomozawa, A. "An Improved Method for Digital SSB-FDM Modulation and Demodulation," *IEEE Trans. on Communications*, Vol. COM-26, No. 5, May 1978.

[7] Albaugh, N. "New ADCs for RF Signal Processing," *RF Design*, November 1987.

[8] Oxaal, J. "DSP Hardware Improves Multiband Filters," *EDN*, March 1983.

[9] Sigman, E. "Simplified Digital Down-Converters," *NASA Tech Briefs*, July 1995.

[10] Coleman, B., Meehan, P., Reidy, J., and Weeks, P. "Coherent Sampling Helps When Specifying DSP A/D Converters," *EDN*, October 1987.

[11] Kester, W. "DSP Test Techniques Keep Flash ADCs in Check," *EDN Magazine*, January 18, 1990.

[12] Morris, G., Jr., and Wilck, H. "JPL 2^{20} Channel 330 MHz Bandwidth Digital Spectrum Analyzer," *IEEE Proc., Int. Conference, Acoust., Speech, Signal Processing*, 1978.

[13] Considine, V. "Digital Complex Sampling," *Electronics Letters*, 19, 4 August, 1983.

[14] Rader, C. M. "A Simple Method for Sampling In-Phase and Quadrature Components," *IEEE Trans. on Aerospace and Electronic Systems*, Vol. AES-20, No. 6, November 1984.

[15] Rice, D., and Wu, K. "Quadrature Sampling with High Dynamic Range," *IEEE Trans. on Aerospace and Electronic Systems*, Vol. AES-18, No. 4, November 1982.

[16] Pellon, L. E. "A Double Nyquist Digital Product Detector for Quadrature Sampling," *IEEE Trans. on Signal Processing*, Vol. 40, No. 7, July 1992.

[17] Crochiere, R. E., and Rabiner, L. R. "Optimum FIR Digital Implementations for Decimation, Interpolation, and Narrow-band Filtering," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-23, No. 5, October 1975.

[18] Ballanger, M. G. "Computation Rate and Storage Estimation in Multirate Digital Filtering with Half-Band Filters," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-25, No. 4, August 1977.

[19] Schafer, R. W., and Rabiner, L. R. "A Digital Signal Processing Approach to Interpolation," *Proceedings of the IEEE*, Vol. 61, No. 6, June 1973.

[20] Crochiere, R. E., and Rabiner, L. R. "Decimation and Interpolation of Digital Signals—A Tutorial Review," *Proceedings of the IEEE*, Vol. 69, No. 3, March 1981.

[21] Crochiere, R. E., and Rabiner, L. R. "Further Considerations in the Design of Decimators and Interpolators," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-24, No. 4, August 1976.

[22] Ballanger, M. G., Daguet, J. L., and Lepagnol, G. P. "Interpolation, Extrapolation, and Reduction of Computational Speed in Digital Filters," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-22, No. 4, August 1974.

[23] Hou, H. S., and Andrews, H. C. "Cubic Splines for Image Interpolation and Digital Filtering," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-26, No. 6, August 1978.

[24] Keys, R. G. "Cubic Convolution Interpolation for Digital Image Processing," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-29, No. 6, August 1981.

[25] Parker, J. A., Kenyon, R. V., and Troxel, D. E. "Comparison of Interpolating Methods for Image Resampling," *IEEE Trans. on Medical Imaging*, Vol. MI-2, No. 1, August 1983.

[26] Blue, K. J., et al. "Vector Signal Analyzers for Difficult Measurements on Time-Varying and Complex Modulated Signals," *Hewlett-Packard Journal*, December 1993.

[27] Bartz, M., et al., "Baseband Vector Signal Analyzer Hardware Design," *Hewlett-Packard Journal*, December 1993.

[28] Mitchell, J. A. "Multirate Filters Alter Sampling Rates Even After You've Captured the Data," *EDN*, August 20, 1992.

CHAPTER EIGHT

Signal Averaging

How do we determine the typical amount, a valid estimate, or the true value of some measured parameter? In the physical world, it's not so easy to do because unwanted random disturbances contaminate our measurements. These disturbances are due to both the nature of the variable being measured and the fallibility of our measuring devices. Each time we try to accurately measure some physical quantity, we'll get a slightly different value. Those unwanted fluctuations in a measured value are called *noise*, and digital signal processing practitioners have learned to minimize noise through the process of averaging. In the literature, we can see not only how averaging is used to improve measurement accuracy, but that averaging also shows up in signal detection algorithms as well as in low-pass filter schemes. This chapter introduces the mathematics of averaging and describes how and when this important process is used. Accordingly, as we proceed to quantify the benefits of averaging, we're compelled to make use of the statistical measures known as the mean, variance, and standard deviation.

In digital signal processing, averaging often takes the form of summing a series of time-domain signal samples and then dividing that sum by the number of individual samples. Mathematically, the average of N samples of sequence $x(n)$, denoted x_{ave}, is expressed as

$$x_{ave} = \frac{1}{N}\sum_{n=1}^{N} x(n) = \frac{x(1)+x(2)+x(3)+\ldots+x(N)}{N} \quad (8\text{-}1)$$

(What we call the average, statisticians call the *mean*.) In studying averaging, a key definition that we must keep in mind is the variance of the sequence σ^2 defined as

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}[x(n) - x_{ave}]^2 , \tag{8-2}$$

$$= \frac{[x(1) - x_{ave}]^2 + [x(2) - x_{ave}]^2 + [x(3) - x_{ave}]^2 + \ldots + [x(N) - x_{ave}]^2}{N} . \tag{8-2'}$$

As explained in Appendix D, the σ^2 variance in Eqs. (8-2) and (8-2') gives us a well-defined quantitative measure of how much the values in a sequence fluctuate about the sequence's average. That's because the $x(1) - x_{ave}$ value in the bracket, for example, is the difference between the $x(1)$ value and the sequence average x_{ave}. The other important quantity that we'll use is the standard deviation defined as the positive square root of the variance, or

$$\sigma = \sqrt{\frac{1}{N}\sum_{n=1}^{N}[x(n) - x_{ave}]^2} . \tag{8-3}$$

To reiterate our thoughts, the average value x_{ave} is the constant level about which the individual sequence values may vary. The variance σ^2 indicates the sum of the magnitudes squared of the noise fluctuations of the individual sequence values about the x_{ave} average value. If the sequence $x(n)$ represents a time series of signal samples, we can say that x_{ave} specifies the constant, or DC, value of the signal, the standard deviation σ reflects the amount of the fluctuating, or AC, component of the signal, and the variance σ^2 is an indication of the power in the fluctuating component. (Appendix D explains and demonstrates the nature of these statistical concepts for those readers who don't use them on a daily basis.)

We're now ready to investigate two kinds of averaging, *coherent* and *incoherent*, to learn how they're different from each other and to see under what conditions they should be used.

8.1 Coherent Averaging

In the coherent averaging process (also known as linear, predetection, or vector averaging), the key feature is the timing used to sample the original signal; that is, we collect multiple sets of signal plus noise samples, and we need the time phase of the signal in each set to be identical. For example, when averaging a sinewave embedded in noise, coherent averaging requires that the phase of the sinewave be the same at the beginning of each measured sample set. When this requirement is met, the sinewave

will average to its true sinewave amplitude value. The noise, however, is different in each sample set and will average toward zero.† The point is that coherent averaging reduces the variance of the noise, while preserving the amplitude of signals that are synchronous, or coherent, with the beginning of the sampling interval. With coherent averaging, we can actually improve the signal-to-noise ratio of a noisy signal. By way of example, consider the sequence of 128 data points plotted in Figure 8-1(a). Those data points represent the time-domain sampling of a single pulse contaminated with random noise. (For illustrative purposes the pulse, whose peak amplitude is 2.5, is shown in the background of Figure 8-1.) It's very difficult to see a pulse in the bold pulse-plus-noise waveform in the foreground of Figure 8-1(a). Let's say we collect 32 sets of 128 pulse-plus-noise samples of the form

$$\begin{aligned}
&\text{Sample Set}_1 = x_1(1), x_1(2), x_1(3), \ldots, x_1(128), \\
&\text{Sample Set}_2 = x_2(1), x_2(2), x_2(3), \ldots, x_2(128), \\
&\text{Sample Set}_3 = x_3(1), x_3(2), x_3(3), \ldots, x_3(128), \\
&\qquad\qquad \ldots \\
&\qquad\qquad \ldots \\
&\text{Sample Set}_{32} = x_{32}(1), x_{32}(2), x_{32}(3), \ldots, x_{31}(128). \qquad (8\text{-}4)
\end{aligned}$$

Here's where the coherent part comes in; the signal measurement times must be synchronized, in some manner, with the beginning of the pulse, so that the pulse is in a constant time relationship with the first sample of each sample set. Coherent averaging of the 32 sets of samples, adding up the columns of Eq. (8-4), takes the form of

$$x_{ave}(k) = \frac{1}{32}\sum_{n=1}^{32} x_n(k) = [x_1(k) + x_2(k) + x_3(k) + \ldots + x_{32}(k)]/32 ,$$

or

$$\begin{aligned}
x_{ave}(1) &= [x_1(1) + x_2(1) + x_3(1) + \ldots + x_{32}(1)]/32 \\
x_{ave}(2) &= [x_1(2) + x_2(2) + x_3(2) + \ldots + x_{32}(2)]/32 \\
x_{ave}(3) &= [x_1(3) + x_2(3) + x_3(3) + \ldots + x_{32}(3)]/32 \\
&\ldots \\
&\ldots \\
x_{ave}(128) &= [x_1(128) + x_2(128) + x_3(128) + \ldots + x_{32}(128)]/32 . \qquad (8\text{-}5)
\end{aligned}$$

† Noise samples are assumed to be uncorrelated with each other and uncorrelated with the sample rate. If some component of the noise is correlated with the sample rate, that noise component *will* be preserved after averaging.

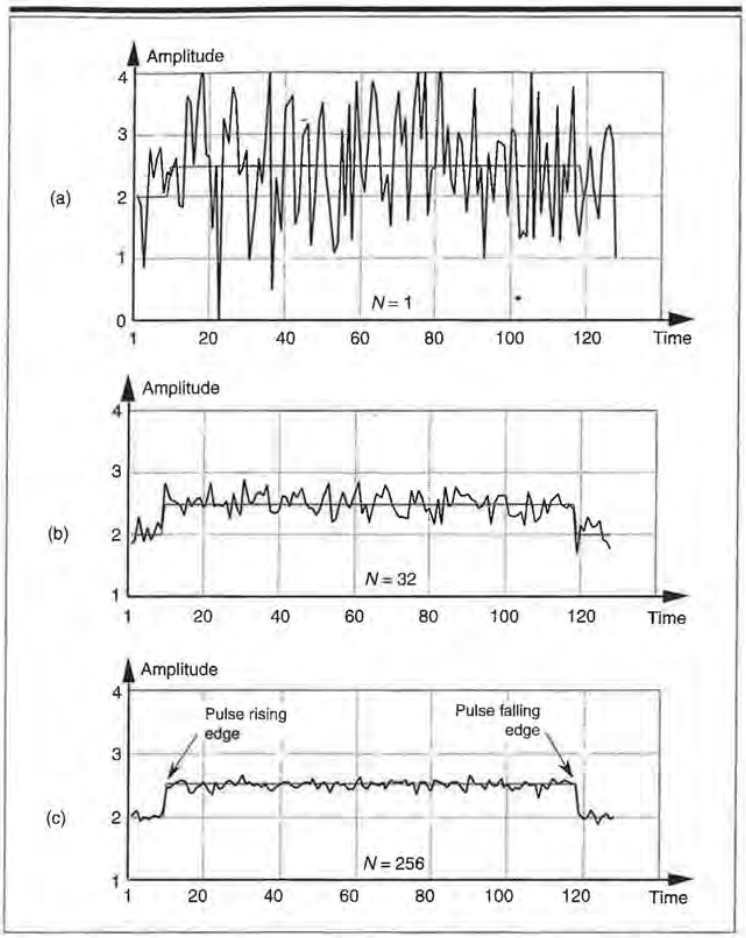


Figure 8-1 Signal pulse plus noise: (a) one sample set; (b) average of 32 sample sets; (c) average of 256 sample sets.

If we perform 32 averages indicated by Eq. (8-5) on a noisy pulse like that in Figure 8-1(a), we'd get the 128-point $x_{ave}(k)$ sequence plotted in Figure 8-1(b). Here, we've reduced the noise fluctuations riding on the pulse, and the pulse shape is beginning to become apparent. The coherent average of 256 sets of pulse measurement sequences results in the plot shown in Figure 8-1(c), where the pulse shape is clearly visible now. We've reduced the noise fluctuations while preserving the pulse amplitude. (An important concept to keep in mind is that summation

and averaging both reduce noise variance. Summation is merely implementing Eq. (8-5) without dividing the sum by $N = 32$. If we perform summations and don't divide by N, we merely change the vertical scales for the graphs in Figure 8-1(b) and (c). However, the noise fluctuations will remain unchanged relative to true pulse amplitude on the new scale.)

The mathematics of this averaging process in Eq. (8-5) is both straightforward and important. What we'd like to know is the signal-to-noise improvement gained by coherent averaging as a function of N, the number of sample sets averaged. Let's say that we want to measure some constant time signal with amplitude A, and each time we actually make a measurement we get a slightly different value for A. We realize that our measurements are contaminated with noise such that the nth measurement result $r(n)$ is

$$r(n) = A + \text{noise}(n)\,, \tag{8-6}$$

where noise(n) is the noise contribution. Our goal is to determine A when the $r(n)$ sequence of noisy measurements is all we have to work with. For a more accurate estimate of A, we average N separate $r(n)$ measurement samples and calculate a single average value r_{ave}. To get a feeling for the accuracy of r_{ave}, we decide to take a series of averages $r_{ave}(k)$, to see how that series fluctuates with each new average; that is,

$$\begin{aligned}
r_{ave}(1) &= [r(1) + r(2) + r(3) + \ldots + r(N)]/N\,, && \leftarrow \text{1st } N\text{-point average}\\
r_{ave}(2) &= [r(N+1) + r(N+2) + r(N+3) + \ldots + r(2N)]/N\,, && \leftarrow \text{2nd } N\text{-point average}\\
r_{ave}(3) &= [r(2N+1) + r(2N+2) + r(2N+3) + \ldots + r(3N)]/N\,, && \leftarrow \text{3rd } N\text{-point average}\\
&\ldots\\
&\ldots\\
r_{ave}(k) &= [r([k-1]\cdot N+1) + r([k-1]\cdot N+2) + r([k-1]\cdot N+3) + \ldots + r(k\cdot N)]/N\,. && (8\text{-}7)
\end{aligned}$$

or, more concisely,

$$r_{ave}(k) = \frac{1}{N}\sum_{n=1}^{N} r([k-1]\cdot N + n)\,. \tag{8-8}$$

To see how averaging reduces our measurement uncertainty, we need to compare the standard deviation of our $r_{ave}(k)$ sequence of averages with the standard deviation of the original $r(n)$ sequence.

If the standard deviation of our original series of measurements $r(n)$ is σ_{in}, it has been shown [1-5] that the standard deviation of our $r_{ave}(k)$ sequence of N-point averages, σ_{ave}, is given by

$$\sigma_{ave} = \frac{\sigma_{in}}{\sqrt{N}} \quad . \tag{8-9}$$

Equation (8-9) is significant because it tells us that the $r_{ave}(k)$ series of averages will not fluctuate as much about A as the original $r(n)$ measurement values did; that is, the $r_{ave}(k)$ sequence will be less noisy than any $r(n)$ sequence, and the more we average by increasing N, the more closely an individual $r_{ave}(k)$ estimate will approach the true value of A.†

In a different way, we can quantify the noise reduction afforded by averaging. If the quantity A represents the amplitude of a signal and σ_{in} represents the standard deviation of the noise riding on that signal amplitude, we can state that the original signal-amplitude-to-noise ratio is

$$SNR_{in} = \frac{A}{\sigma_{in}} \quad . \tag{8-10}$$

Likewise, the signal-amplitude-to-noise ratio at the output of an averaging process, SNR_{ave} is defined as

$$SNR_{ave} = \frac{r_{ave}}{\sigma_{ave}} = \frac{A}{\sigma_{ave}} \quad . \tag{8-11}$$

Continuing, the signal-to-noise ratio *gain*, SNR_{coh} gain, that we've realized through coherent averaging is the ratio of SNR_{ave} over SNR_{in}, or

$$SNR_{coh}\ \text{gain} = \frac{SNR_{ave}}{SNR_{in}} = \frac{A/\sigma_{ave}}{A/\sigma_{in}} = \frac{\sigma_{in}}{\sigma_{ave}} \quad . \tag{8-12}$$

Substituting σ_{ave} from Eq. (8-9) in Eq. (8-12), the SNR gain becomes

$$SNR_{coh}\ \text{gain} = \frac{\sigma_{in}}{\sigma_{in}/\sqrt{N}} = \sqrt{N} \quad . \tag{8-13}$$

† Equation (8-9) is based on the assumptions that the average of the original noise is zero and that neither A nor σ_{in} change during the time we're performing our averages.

Through averaging, we can realize a signal-to-noise ratio improvement proportional to the square root of the number of signal samples averaged. In terms of signal-to-noise ratio measured in decibels, we have a coherent averaging, or *integration*, gain of

$$SNR_{coh}\ \text{gain(dB)} = 20 \cdot \log_{10}(SNR_{coh}) = 20 \cdot \log_{10}(\sqrt{N}) = 10 \cdot \log_{10}(N)\ . \quad (8\text{-}14)$$

Again, Eqs. (8-13) and (8-14) are valid if A represents the *amplitude* of a signal and σ_{in} represents the original noise standard deviation.

Another way to view the integration gain afforded by coherent averaging is to consider the standard deviation of the input noise, σ_{in} and the probability of measuring a particular value for the Figure 8-1 pulse amplitude. Assume that we made many individual measurements of the pulse amplitude and created a fine-grained histogram of those measured values to get the dashed curve in Figure 8-2(a). The vertical axis of Figure 8-2(a) represents the probability of measuring a pulse-amplitude value corresponding to the values on the horizontal axis. If the noise fluctuations follow the well-known *normal*, or Gaussian distribution, that dashed probability distribution curve is described by

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} = Ke^{-(x-\mu)^2/2\sigma^2} \quad (8\text{-}15)$$

where $\sigma = \sigma_{in}$ and the true pulse amplitude is represented by $\mu = 2.5$. We see from that dashed curve that any given measured value will most likely (with highest probability) be near the actual pulse-amplitude value of 2.5. Notice, however, that there's a nonzero probability that the measured value could be as low as 1.0 or as high as 4.0. Let's say that the dashed curve represents the probability curve of the pulse-plus-noise signal in Figure 8-1(a). If we averaged a series of 32 pulse-amplitude values and plotted a probability curve of our averaged pulse-amplitude measurements, we'd get the solid curve in Figure 8-2. This curve characterizes the pulse-plus-noise values in Figure 8-1(b). From this solid curve, we see that there's a very low likelihood (probability) that a measured value, after 32-point averaging, will be less than 2.0 or greater than 3.0.

From Eq. (8-9), we know that the standard deviation of the result of averaging 32 signal sample sets

$$\sigma_{ave} = \frac{\sigma_{in}}{\sqrt{32}} = \frac{\sigma_{in}}{5.65}\ . \quad (8\text{-}16)$$

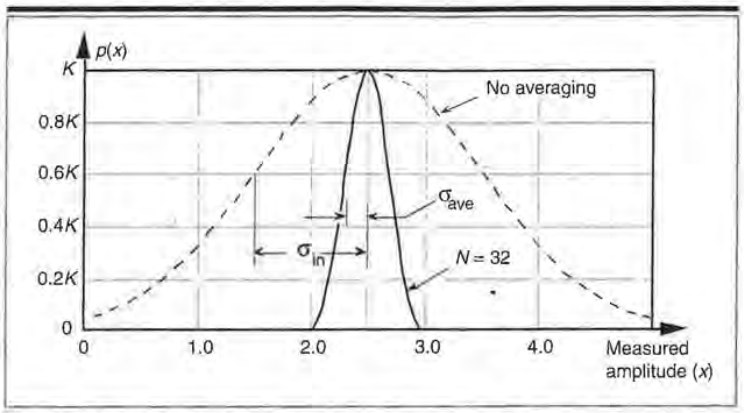


Figure 8-2 Probability density curves of measured pulse amplitudes with no averaging ($N = 1$) and with $N = 32$ averaging.

In Figure 8-2, we can see a statistical view of how an averager's output standard deviation is reduced from the averager's input standard deviation. Taking larger averages by increasing N beyond 32 would squeeze the solid curve in Figure 8-2 even more toward its center value of 2.5, the true pulse amplitude.†

Returning to the noisy pulse signal in Figure 8-1, and performing coherent averaging for various numbers of sample sets N, we see in Figure 8-3(a) that as N increases, the averaged pulse amplitude approaches the true amplitude of 2.5. Figure 8-3(b) shows how rapidly the variance of the noise riding on the pulse falls off as N is increased. An alternate way to see how the noise variance decreases with increasing N is the noise power plotted on a logarithmic scale as in Figure 8-3(c). In this plot, the noise variance is normalized to that noise variance when no averaging is performed, i.e., when $N = 1$. Notice that the slope of the curve in Figure 8-3(c) closely approximates that predicted by Eqs. (8-13) and (8-14); that is, as N increases by a factor of 10, we reduce the average noise power by 10 dB. Although the test signal in this discussion was a pulse signal, had the signal been sinusoidal, Eqs. (8-13) and (8-14) would still apply.

† The curves in Figure 8-2 are normalized for convenient illustration. From Eq. (8-15) and assuming that $\sigma = 1$ when $N = 1$, then $K = 0.3989$. When $N = 32$, the new standard deviation is $\sigma' = \sigma/\sqrt{N} = 1/\sqrt{32}$ and $K = 0.3989 \cdot \sqrt{32} = 2.23$.

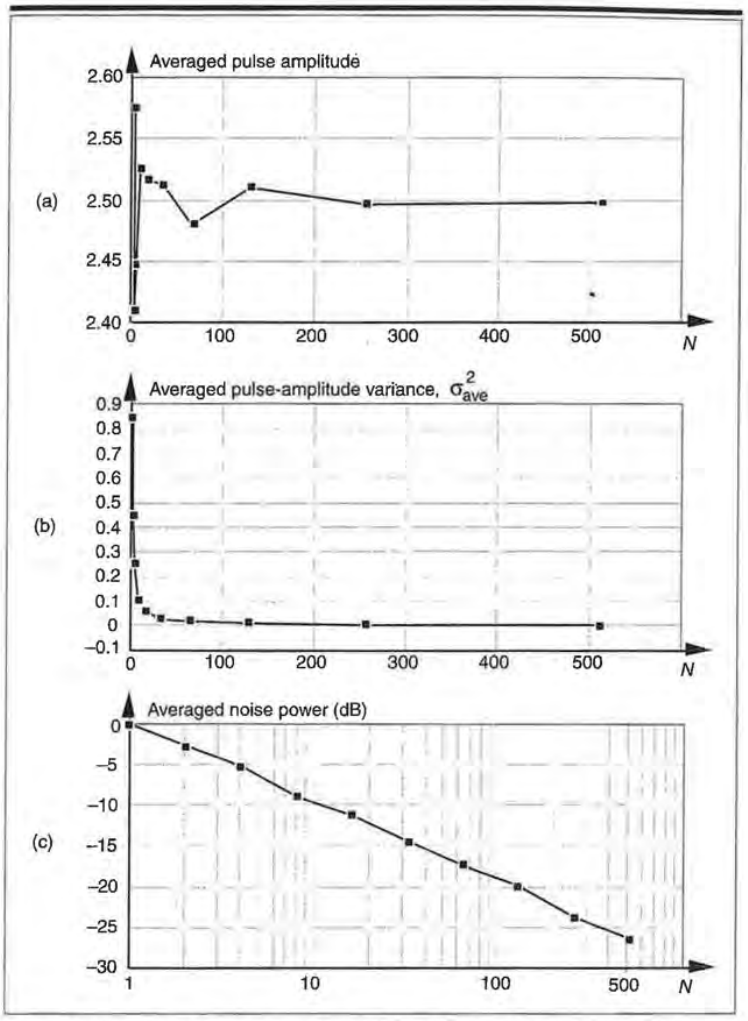


Figure 8-3 Results of averaging signal pulses plus noise: (a) measured pulse amplitude vs. N; (b) measured variance of pulse amplitude vs. N; (c) measured pulse-amplitude noise power vs. N on a logarithmic scale.

8.2 Incoherent Averaging

The process of incoherent averaging (also known as rms, postdetection, scalar, or video averaging) is the averaging of signal samples where no sample timing constraints are used; that is, signal measurement time

intervals are not synchronized in any way with the phase of the signal being measured. Think for a moment what the average would be of the noisy pulse signal in Figure 8-1(a) if we didn't in some way synchronize the beginning of the collection of the individual signal sample sets with the beginning of the pulse. The result would be pulses that begin at a different time index in each sample set. The averaging of multiple sample sets would then smear the pulse across the sample set, or just "average the pulse signal away." (For those readers familiar with using oscilloscopes, incoherent averaging would be like trying to view the pulse when the beginning of the scope sweep was not triggered by the signal.) As such, incoherent averaging is not so useful in the time domain.[†] In the frequency domain, however, it's a different story because incoherent averaging can provide increased accuracy in measuring relative signal powers. Indeed, incoherent averaging is used in many test instruments, such as spectrum, network, and signal analyzers.

In some analog test equipment, time-domain signals are represented in the frequency domain using a narrowband sweeping filter followed by a power detector. These devices measure signal power as a function of frequency. The power detector is necessary because the sweeping measurement is not synchronized, in time, with the signal being measured. Thus the frequency-domain data represents power only and contains no signal phase information. Although it's too late to improve the input's signal-amplitude-to-noise ratio, incoherent averaging can improve the accuracy of signal power measurements in the presence of noise; that is, if the signal-power spectrum is very noisy, we can reduce the power estimation fluctuations and improve the accuracy of signal-power and noise-power measurements. Figure 8-4(a) illustrates this idea where we see the power (magnitude squared) output of an FFT of a fundamental tone and several tone harmonics buried in background noise. Notice that the noise-power levels in Figure 8-4(a) fluctuate by almost 20 dB about the true average noise power indicated by the dashed line at –19 dB.

If we take 10 FFTs, average the square of their output magnitudes, and normalize those squared values, we get the power spectrum shown in Figure 8-4(b). Here, we've reduced the variance of the noise in the power spectrum but have not improved the tones' signal-power-to-noise-power ratios; that is, the average noise-power level remains unchanged. Averaging the output magnitudes squared of 100 FFTs results in the spec-

[†] The term *incoherent averaging* is a bit of a misnomer. Averaging a set of data is just that, averaging—we add up a set of data values and divide by the number of samples in the set. Incoherent averaging should probably be called *averaging data that's obtained incoherently*.

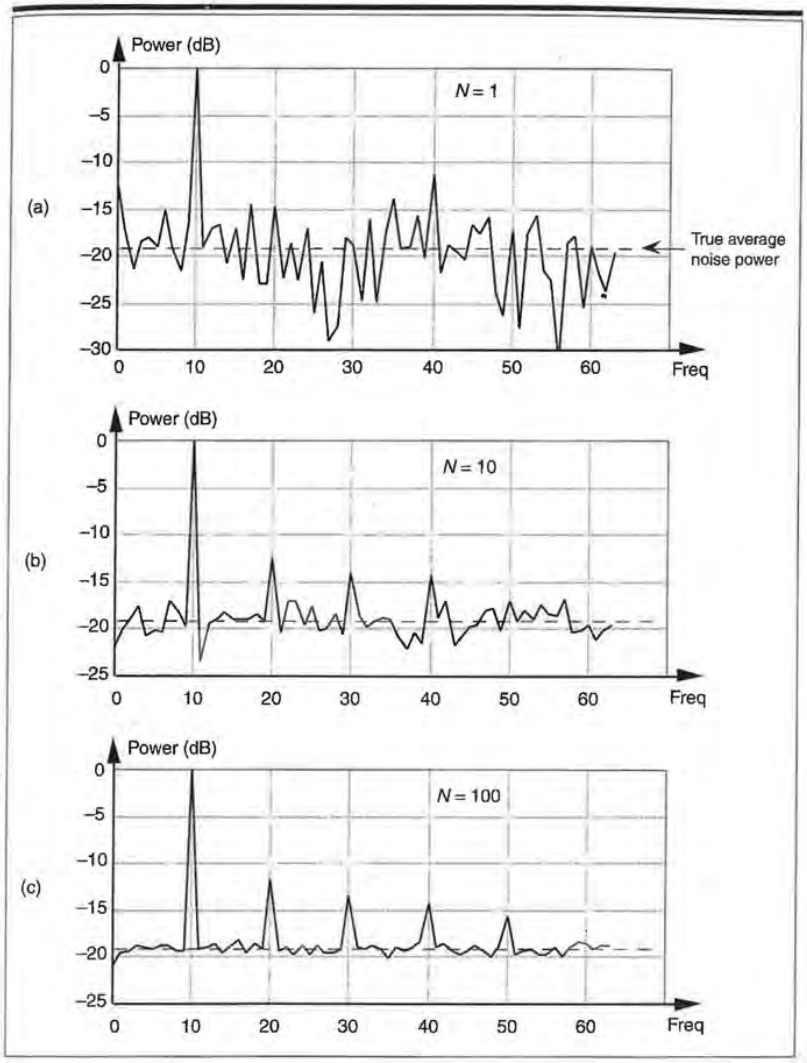


Figure 8-4 Results of averaging signal tones plus noise-power spectra: (a) no averaging, $N = 1$; (b) $N = 10$; (c) $N = 100$.

trum in Figure 8-4(c), which provides a more accurate measure of the relative power levels of the fundamental tone's harmonics.

Just as we arrived at a coherent integration SNR gain expression in Eq. (8-14), we can express an incoherent integration gain, SNR_{incoh} gain, in terms of SNR measured in dB as

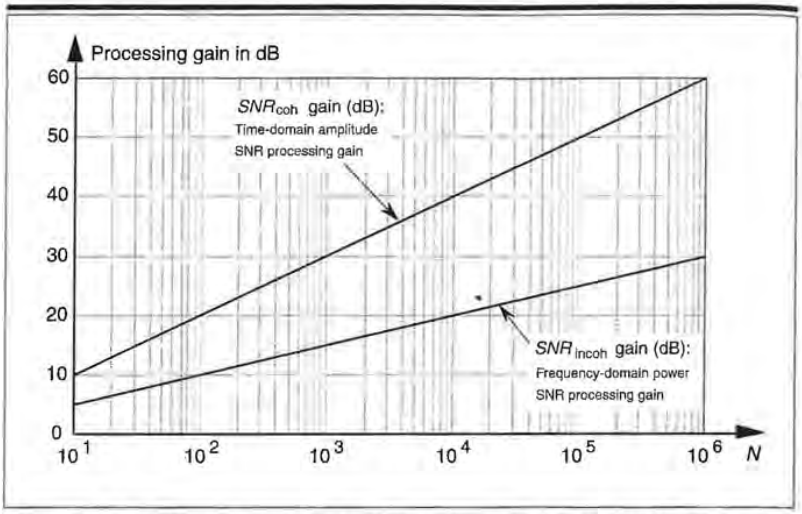


Figure 8-5 Time-domain amplitude SNR processing gain from Eq. (8-14), and the frequency-domain power SNR processing gain from Eq. (8-17), as functions of *N*.

$$SNR_{\text{incoh}}\,\text{gain(dB)} = 10 \cdot \log_{10}(\sqrt{N}) \ . \tag{8-17}$$

Equation (8-17) applies when the quantity being averaged represents the *power* of a signal. That's why we used the factor of 10 in Eq. (8-17) as opposed to the factor of 20 used in Eq. (8-14).† We can relate the processing gain effects of Eqs. (8-14) and (8-17) by plotting those expressions in Figure 8-5.

8.3 Averaging Multiple Fast Fourier Transforms

We discussed the processing gain associated with a single DFT in Section 3.12 and stated that we can realize further processing gain by increasing the point size of any given *N*-point DFT. Let's discuss this issue when the DFT is implemented using the FFT algorithm. The problem is that large FFTs require a lot of number crunching. Because addition is easier and faster to perform than multiplication, we can average the outputs of multiple FFTs to obtain further FFT signal detection sensitivity; that is, it's eas-

† Section E.1 of Appendix E explains why the multiplying factor is 10 for signal-power measurements and 20 when dealing with signal-amplitude values.

ier and typically faster to average the outputs of four 128-point FFTs than it is to calculate one 512-point FFT. The increased FFT sensitivity, or noise variance reduction, due to multiple FFT averaging is also called integration gain. So the random noise fluctuations in a FFT's output bins will decrease, while the magnitude of the FFT's signal bin output remains constant when multiple FFT outputs are averaged. (Inherent in this argument is the assumption that the signal is present throughout the observation intervals for all of the FFTs that are being averaged and that the noise sample values are independent both of each other and of the original sample rate.) There are two types of FFT averaging integration gain, incoherent and coherent.

Incoherent integration, relative to FFTs, is averaging the corresponding bin magnitudes of multiple FFTs; that is, to incoherently average k FFTs, the zeroth bin of the incoherent FFT average $F_{\text{incoh}}(0)$ is given by

$$F_{\text{incoh}}(0) = \frac{|F_1(0)| + |F_2(0)| + |F_3(0)| + \ldots + |F_k(0)|}{k}, \quad (8\text{-}18)$$

where $|F_n(0)|$ is the magnitude of the zeroth bin from the nth FFT. Likewise, the first bin of the incoherent FFT average, $F_{\text{incoh}}(1)$, is given by

$$F_{\text{incoh}}(1) = \frac{|F_1(1)| + |F_2(1)| + |F_3(1)| + \ldots + |F_k(1)|}{k}, \quad (8\text{-}18')$$

and, so on, out to the last bin of the FFT average, $F_{\text{incoh}}(N\text{–}1)$, which is

$$F_{\text{incoh}}(N-1) = \frac{|F_1(N-1)| + |F_2(N-1)| + |F_3(N-1)| + \ldots + |F_k(N-1)|}{k} \quad (8\text{-}18'')$$

Incoherent integration provides additional reduction in background noise variation to augment a single FFT's inherent processing gain. We can demonstrate this in Figure 8-6(a) where the shaded curve is a single FFT output of random noise added to a tone centered in the 16th bin of a 64-point FFT. The solid curve in Figure 8-6(a) is the incoherent integration of 10 individual 64-point FFT magnitudes. Both curves are normalized to their peak values, so that the vertical scales are referenced to 0 dB. Notice how the variations in the noise power in the solid curve have been reduced by the averaging of the 10 FFTs. The noise power values in the solid curve don't fluctuate as much as the shaded noise-power values. By

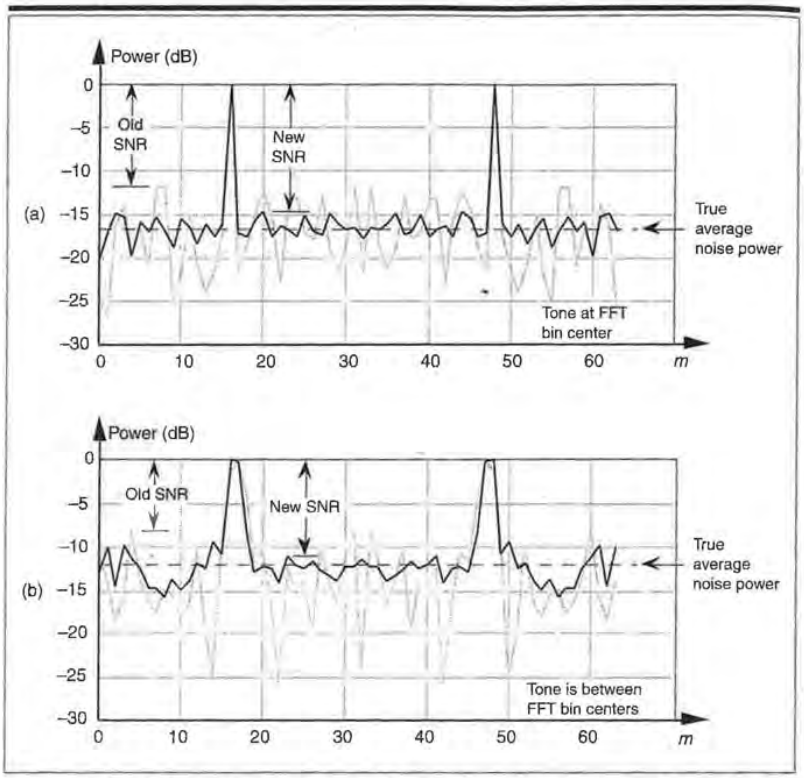


Figure 8-6 Single FFT output magnitudes (shaded) and the average of 10 FFT output magnitudes (solid): (a) tone at bin center; (b) tone between bin centers.

averaging, we haven't raised the power of the tone in the 16th bin, but we have reduced the peaks of the noise-power values. The larger the number of FFTs averaged, the closer the individual noise-power bin values will approach the true average noise power indicated by the dashed horizontal line in Figure 8-6(a).

When the signal tone is not at a bin center, incoherent integration still reduces fluctuations in the FFT's noise-power bins. The shaded curve in Figure 8-6(b) is a single FFT output of random noise added to a tone whose frequency is halfway between the 16th and 17th bins of the 64-point FFT. Likewise, the solid curve in Figure 8-6(b) is the magnitude average of 10 FFTs. The variations in the noise power in the solid curve have again been reduced by the integration of the 10 FFTs. So incoherent integration gain reduces noise-power fluctuations regardless of the fre-

quency location of any signals of interest. As we would expect, the signal peaks are wider, and the true average noise power is larger in Figure 8-6(b) relative to Figure 8-6(a) because leakage raises the average noise-power level and scalloping loss reduces the FFT bin's output power level in Figure 8-6(b). The thing to remember is that incoherent averaging of FFT output magnitudes reduces the *variations* in the background noise power but does not reduce the average background noise power. Equivalent to the incoherent averaging results in Section 8.2, the reduction in the output noise variance[6] of the incoherent average of k FFTs relative to the output noise variance of a single FFT is expressed as

$$\frac{\sigma^2_{k\text{FFTs}}}{\sigma^2_{\text{single FFT}}} = \frac{1}{k} . \tag{8-19}$$

Accordingly, if we average the magnitudes of k separate FFTs, we reduce the noise variance by a factor of k.

In practice, when multiple FFTs are averaged and the FFT inputs are windowed, an overlap in the time-domain sampling process is commonly used. Figure 8-7 illustrates this concept with $5.5Nt_s$ seconds worth of time series data samples, and we wish to average ten separate N-point FFTs where t_s is the sample period ($1/f_s$). Because the FFTs have a 50 percent overlap in the time domain, some of the input noise in the N time samples for the first FFT will also be contained in the second FFT. The question is

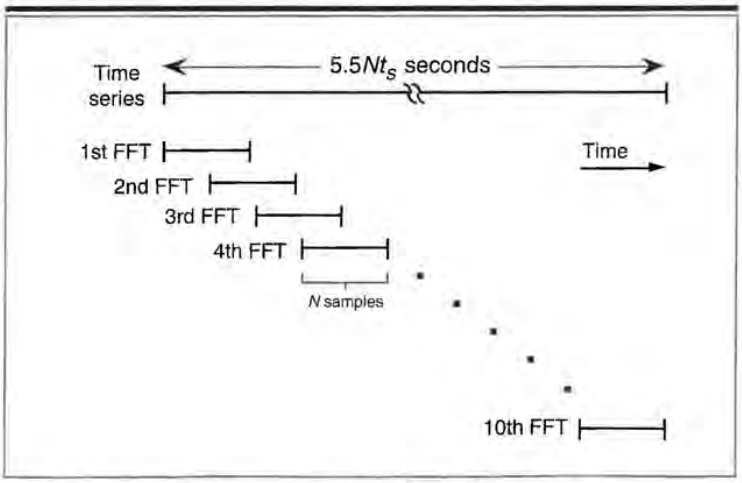


Figure 8-7 Time relationship of multiple FFTs with 50 percent overlap.

"What's the noise variance reduction when some of the noise is common to two FFTs in this averaging scheme?" Well, the answer depends on the window function used on the data before the FFTs are performed. It's been shown that for the most common window functions using an overlap of 50 percent or less, Eq. (8-19) still applies as the level of noise variance reduction[7].

Coherent integration gain is possible when we average the real parts of multiple FFT bin outputs separately from the average of the imaginary parts and then combine the single real average and the single imaginary average into a single complex bin output average value; that is, to coherently average k separate FFTs, the zeroth bin of the complex coherent FFT average, $F_{coh}(0)$, is given by

$$F_{coh}(0) = \frac{F_1(0)_{real} + F_2(0)_{real} + \ldots + F_k(0)_{real}}{k}$$

$$= j\frac{F_1(0)_{imag} + F_2(0)_{imag} + \ldots + F_k(0)_{imag}}{k} . \quad (8\text{-}20)$$

The first bin of the complex coherent FFT average, $F_{coh}(1)$, is

$$F_{coh}(1) = \frac{F_1(1)_{real} + F_2(1)_{real} + \ldots + F_k(1)_{real}}{k}$$

$$+ j\frac{F_1(1)_{imag} + F_2(1)_{imag} + \ldots + F_k(1)_{imag}}{k} , \quad (8\text{-}20')$$

and so on out to the last bin of the complex FFT average, $F_{coh}(N\text{–}1)$, which is

$$F_{coh}(N-1) = \frac{F_1(N-1)_{real} + F_2(N-1)_{real} + \ldots + F_k(N-1)_{real}}{k}$$

$$+ j\frac{F_1(N-1)_{imag} + F_2(N-1)_{imag} + \ldots + F_k(N-1)_{imag}}{k} . \quad (8\text{-}20'')$$

Let's consider why the integration gain afforded by coherent averaging is more useful than the integration gain from incoherent averaging. Where incoherent integration does not reduce the background noise-

power average level, it does reduce the variations in the averaged FFT's background noise power because we're dealing with magnitudes only, and all FFT noise bin values are positive—they're magnitudes. So their averaged noise bin magnitude values will never be zero. On the other hand, when we average FFT complex bin outputs, those complex noise bin values can be positive *or* negative. Those positive and negative values, in the real or imaginary parts of multiple FFT bin outputs, will tend to cancel each other. This means that noise bin averages can approach zero before we take the magnitude, while a signal bin average will approach its true nonzero magnitude. If we say that the coherently averaged FFT signal-to-noise ratio is expressed by

$$SNR_{coh} = 20 \cdot \log_{10}\left(\frac{\text{signal bin magnitude}}{\text{noise bin magnitude}}\right), \tag{8-21}$$

we can see that, should the denominator of Eq. (8-21) approach zero, then SNR_{coh} can be increased. Let's look at an example of coherent integration gain in Figure 8-8(a) where the shaded curve is a single FFT output of random noise added to a tone centered in the 16th bin of a 64-point FFT. The solid curve in Figure 8-8(a) is the coherent integration of ten 64-point FFTs. Notice how the new noise-power average has actually been reduced.[†] That's coherent integration. The larger the number of FFTs averaged, the greater the reduction in the new noise-power average.

When the signal tone is not at bin center, coherent integration still reduces the new noise-power average, but not as much as when the tone is at bin center. The shaded curve in Figure 8-8(b) is a single FFT output of random noise added to a tone whose frequency is halfway between the 16th and 17th bins of the 64-point FFT. Likewise, the solid curve in Figure 8-8(b) is the coherent integration of 10 FFTs. The new noise-power average has, again, been reduced by the integration of the 10 FFTs. Coherent integration provides its maximum gain when the original sample rate f_s is an integral multiple of the tone frequency we're trying to detect. That way the tone will always have the same phase angle at the beginning of the sample interval for each FFT. So, if possible, the f_s sample rate should be

[†] The sharp-eyed reader might say, "Wait a minute. Coherent integration seems to reduce the noise-power average, but it doesn't look like it reduces the noise-power variations." Well, the noise-power variations really are reduced in magnitude—remember, we're using a logarithmic scale. The magnitude of a half-division power variation between –20 dB and –30 dB is smaller than the magnitude of a half-division power variation between –10 dB and –20 dB.

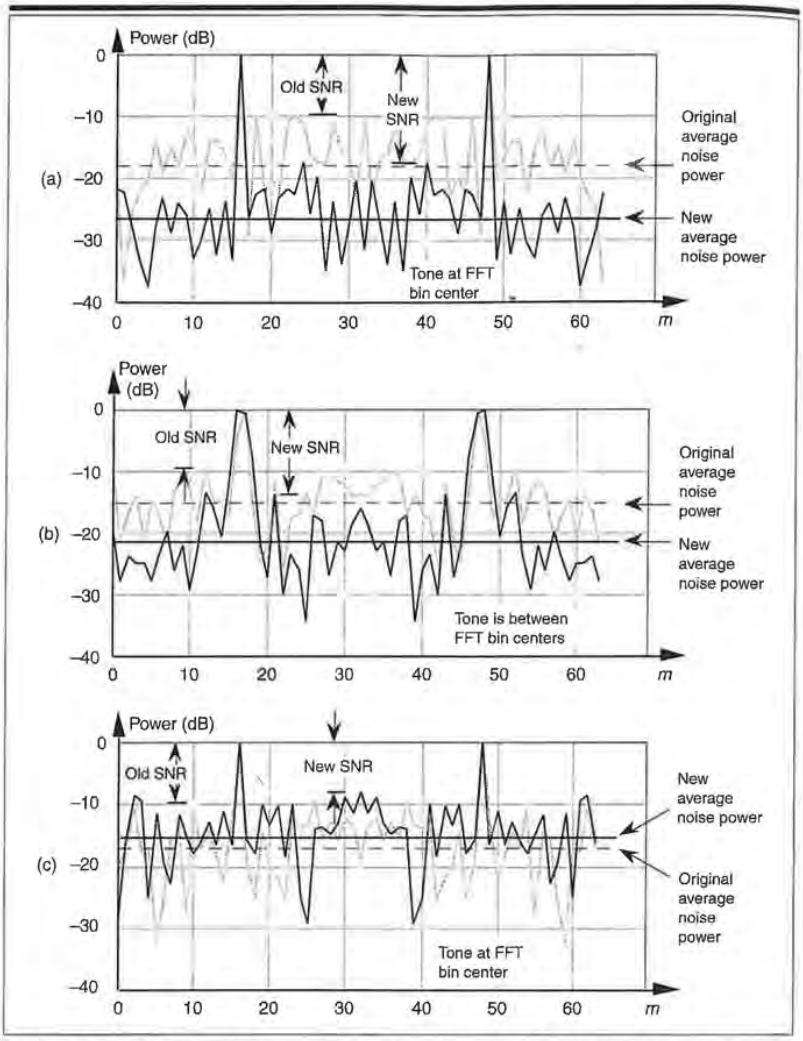


Figure 8-8 Single FFT output magnitudes (shaded) and the coherent integration of 10 FFT outputs (solid): (a) tone at bin center; (b) tone between bin centers; (c) tone at bin center with random phase shift induced prior to the FFT process.

chosen to ensure that the tone of interest, assuming it's not drifting in frequency, will be exactly at a FFT bin center.

Keep two thoughts in mind while using coherent integration of multiple FFTs. First, if the random noise contaminating the time signal does not have an average value of zero, there will be a nonzero DC term in the

averaged FFT's $F_{coh}(0)$ bin. Averaging, of course, will not reduce this DC level in the averaged FFT $F_{coh}(0)$ output. Second, coherent integration can actually reduce the averaged FFT's SNR if the tone being detected has a random phase at the beginning of each FFT sample interval. An example of this situation is as follows: the shaded curve in Figure 8-8(c) is a single FFT output of random noise added to a tone centered in the 16th bin of a 64-point FFT. The solid curve in Figure 8-8(c) is the coherent integration of 10 individual 64-point FFTs. However, for each of the 10 FFTs, a random phase shift was induced in the fixed-frequency test tone prior to the FFT. Notice how the averaged FFT's new average noise power is larger than the original average noise power for the single FFT. When the tone has a random phase, we've actually lost rather than gained SNR because the input tone was no longer phase coherent with the analysis frequency of the FFT bin.

There's a good way to understand why coherent integration behaves the way it does. If we look at the phasors representing the successive outputs of a FFT bin, we can understand the behavior of a single phasor representing the sum of those outputs. But, first, let's refresh our memory for a moment concerning the *vector* addition of two phasors.[†] Because an FFT bin output is a complex quantity with a real and an imaginary part, we can depict a single FFT bin output as a phasor, like phasor A shown in Figure 8-9(a). We can depict a subsequent output from the same FFT bin as phasor B in Figure 8-9(b). There are two ways to coherently add the two FFT bin outputs, i.e., add phasors A and B. As shown in Figure 8-9(c), we can add the two real parts and add the two imaginary parts to get the sum phasor C. A graphical method of summing phasors A and B, shown in Figure 8-9(d), is to position the beginning of phasor B at the end of phasor A. Then the sum phasor C is the new phasor from the beginning of phasor A to the end of phasor B. Notice how the two C phasors in Figures 8-9(c) and 8-9(d) are identical. We'll use the graphical phasor summing technique to help us understand coherent integration of FFT bin outputs.

Now we're ready to look at the phasor outputs of an FFT signal bin to see how a single phasor representing the sum of multiple signal bin phasors behaves. Consider the three phasor combinations in Figure 8-10(a). Each phasor is a separate output of the FFT bin containing the signal tone we're trying to detect with coherent integration. The dark arrows are the phasor components due to the tone, and the small shaded arrows are the phasor components due to random noise in the FFT bin containing the tone. In this case, the original signal sample rate is a multiple of the tone

[†] Following the arguments put forth in Section A.2 of Appendix A, we'll use the term *phasor*, as opposed to the term *vector*, to describe a single, complex DFT output value.

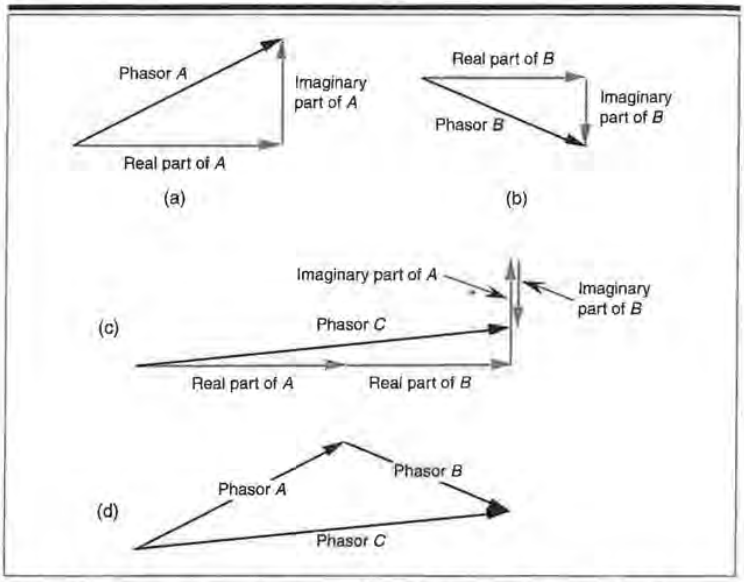


Figure 8-9 Two ways to add phasors *A* and *B*, where phasor $C = A + B$.

frequency, so that the tone phasor phase angle ø is the same at the beginning of the three FFTs, sample intervals. Thus, the three dark tone phasors have a zero phase angle shift relative to one another. The phase angles of the random noise components are random. If we add the three phasor combinations, as a first step in coherent integration, we get the actual phasor sum shown in Figure 8-10(b). The thick shaded vector in Figure 8-10(b) is the ideal phasor sum that would result had there been no noise components in the original three phasor combinations. Because the noise components are reasonably small, the actual phasor sum is not too different from the ideal phasor sum.

Now, had the original signal samples been such that the input tone's phase angles were random at the beginning of the three FFTs, sample intervals, the three phasor combinations in Figure 8-10(c) could result. Summing those three phasor combinations results in the actual phasor sum shown in Figure 8-10(d). Notice that the random tone phases, in Figure 8-10(c), have resulted in an actual phasor sum magnitude (length) that's shorter than the dark lines of the phasor component due to the tone in Figure 8-10(c). What we've done here is degrade, rather than improve,

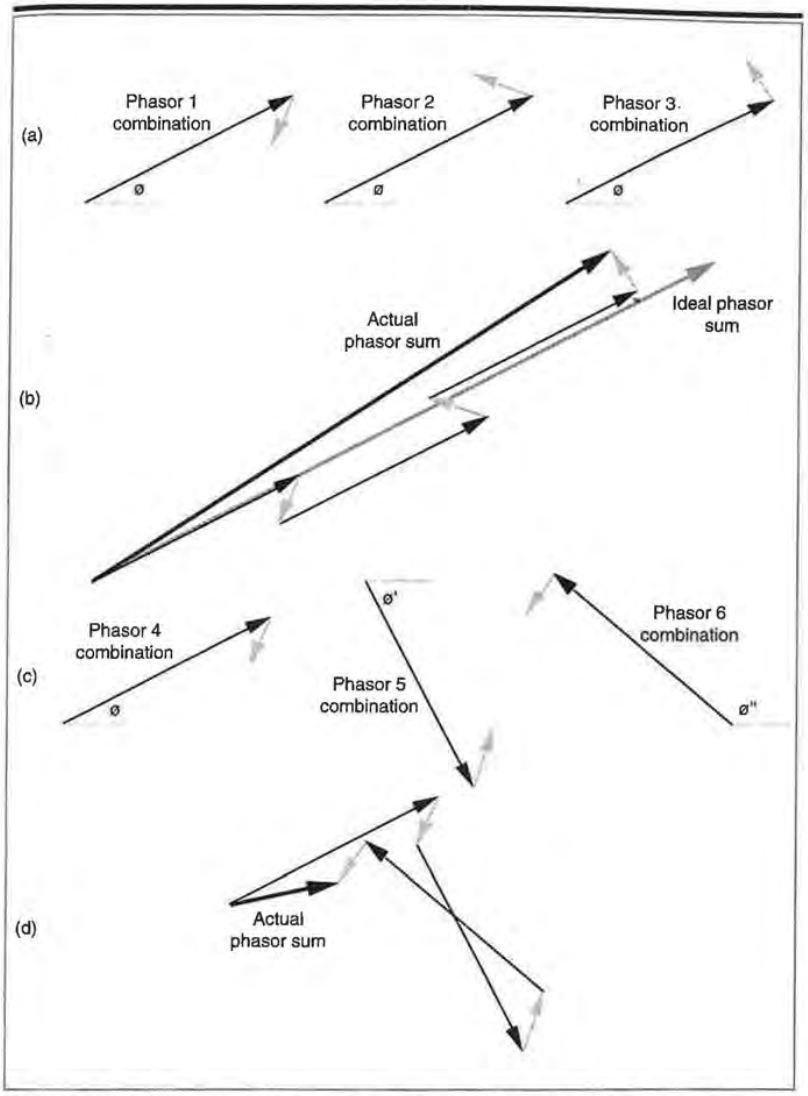


Figure 8-10 FFT bin outputs represented as phasors: (a) three outputs from the same bin when the input tone is at bin center; (b) coherent integration (phasor addition) of the three bin outputs; (c) three outputs from the same bin when input tone has a random phase at the beginning of each FFT; (d) coherent integration of the three bin outputs when input tone has random phase.

the averaged FFT's output SNR when the tone has a random phase at the beginning of each FFT sample interval.

The thought to remember is that, although coherent averaging of FFT outputs is the preferred technique for realizing integration gain, we're rarely lucky enough to work real-world signals with a constant phase at the beginning of every time-domain sample interval. So, in practice, we're usually better off using incoherent averaging. Of course, with either integration technique, the price we pay for improved FFT sensitivity is additional computational complexity and slower system response times because of the additional summation calculations. (A slick way to reduce the computational burden of averaging multiple FFTs is discussed in Section 10.7.)

8.4 Filtering Aspects of Time-Domain Averaging

In Section 5.2 we introduced FIR filters with an averaging example, and that's where we first learned that the process of time-domain averaging performs low-pass filtering. In fact, successive time-domain outputs of an N-point averager are identical to the output of an $(N-1)$-tap FIR filter whose coefficients are all equal to $1/N$, as shown in Figure 8-11.

The question we'll answer here is "What is the frequency magnitude response of a generic N-point averager?" We could evaluate Eq. (6-42), from Section 6.3, that describes the frequency response of a generic N-stage FIR filter. In those expressions, we'd have to set all the $b(0)$ through $b(N)$ coefficient values equal to $1/N$ and calculate $H_{FIR}(\omega)$'s magnitude over the normalized radian frequency range of $0 \le \omega \le \pi$. That range corresponds to an actual frequency range of $0 \le f \le f_s/2$ (where f_s is the equivalent data sample rate in Hz). A simpler approach is to recall, from Section 5.2, that we can calculate the frequency response of an FIR filter by taking the DFT of the filter's coefficients. In doing so, we'd use an M-point FFT

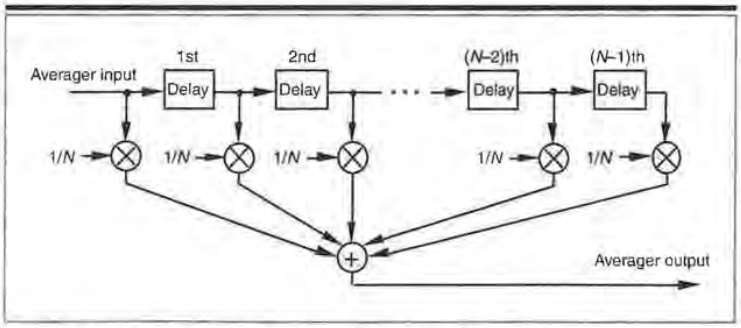


Figure 8-11 An N-point averager depicted as an FIR filter.

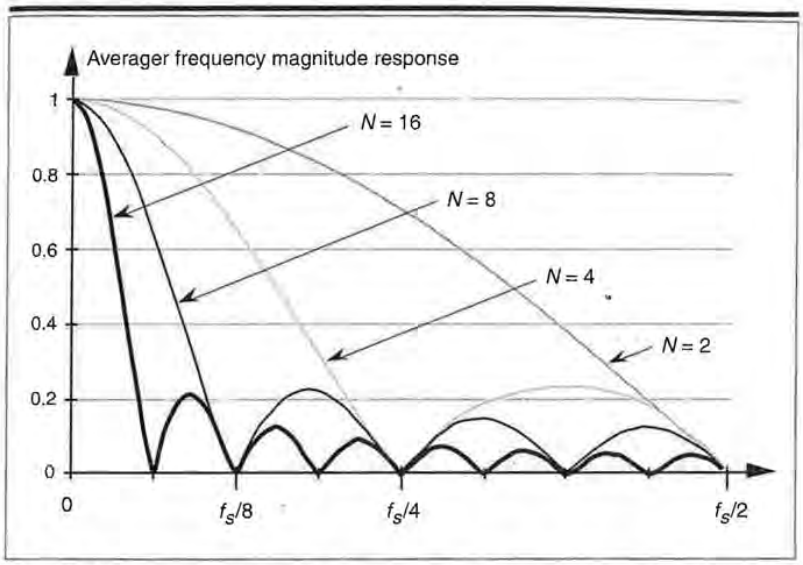


Figure 8-12 *N*-point averager's frequency magnitude response as a function of *N*.

software routine to transform a sequence of N coefficients whose values are all equal to $1/N$. Of course, M should be larger than N so that the $\sin(x)/x$ shape of the frequency response is noticeable. Following through on this by using a 128-point FFT routine, our N-point averager's frequency magnitude responses, for various values of N, are plotted in Figure 8-12. To make these curves more meaningful, the frequency axis is defined in terms of the sample rate f_s in samples/s.

8.5 Exponential Averaging

There is a kind of time-domain averaging used in some power measurement equipment—it's called exponential averaging[8–11]. This technique provides noise reduction by multiplying an input sample by a constant and adding that product to the constant's ones complement multiplied by the most recent averager output. Sounds complicated in words, but the equation for the exponential averager is the simple expression

$$y(n) = \alpha x(n) + (1 - \alpha)y(n - 1), \tag{8-22}$$

where $y(n)$ is the current averager output sample, $y(n–1)$ is the previous averager output sample, and α is the *weighting factor* constant. The

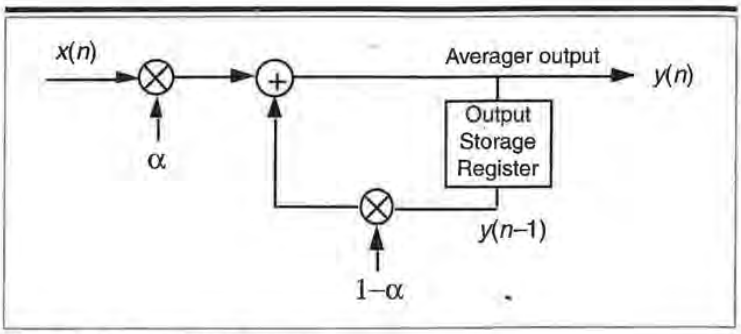


Figure 8-13 Exponential averager.

process described by Eq. (8-22) is implemented as shown in Figure 8-13. The advantage of exponential averaging is that only one storage register is needed to hold the value $y(n-1)$ while waiting for the next input data sample $x(n)$.

The exponential averager's name stems from its time-domain impulse response. Let's assume that the input to the averager is a long string of zeros and we apply a single sample of value 1 at time $t = 0$. Then the input returns again to a string of zero-valued samples. Now, if the weighting factor is $\alpha = 0.4$, the averager's output is shown as the curve in Figure 8-14. When $t = 0$, the input sample is multiplied by α, so the output is 0.4. On the next clock cycle, the input is zero, and the old value of 0.4 is multiplied by $(1 - 0.4)$, or 0.6, to provide an output of 0.24. On the following clock cycle, the input is zero, and the old value of 0.24 is multiplied by 0.6 to provide an output of 0.144. This continues with the averager's output, or impulse response, falling off exponentially because of successive multiplications by 0.6.

A useful feature of the exponential averager is its capability to vary the amount of noise reduction by changing the value of the α weighting factor. If α equals one, input samples are not attenuated, past averager outputs are ignored, and no averaging takes place. In this case, the averager output responds immediately to changes at the input. As α is decreased in value, input samples are attenuated, and past averager outputs begin to affect the present output. These past values represent an exponentially weighted sum of recent inputs, and that summation tends to smooth out noisy signals. The smaller α gets, the more noise reduction is realized. However, with smaller values for α, the slower the averager is in responding to changes in the input. We can demonstrate this behavior by looking

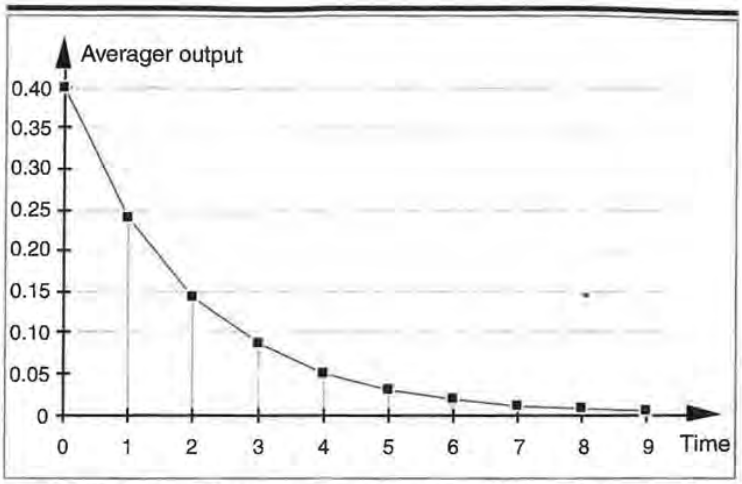


Figure 8-14 Exponential averager impulse response with $\alpha = 0.4$.

at the exponential averager's time-domain step response for various values of α as shown in Figure 8-15.†

So we have a trade-off. The more the noise reduction, the more sluggish the averager will be in responding to changes at the input. We can see in Figure 8-15 that, as α gets smaller, affording better noise reduction, the averager's output takes longer to respond and stabilize. Some test instrumentation manufacturers use a clever scheme to resolve this noise reduction versus response time trade-off. They use a large value for α at the beginning of a measurement so that the averager's output responds immediately with a nonzero value. Then, as the measurement proceeds, the value of α is decreased to reduce the noise fluctuations at the input.

The exponential averager's noise variance reduction as a function of the weighting factor α has been shown to be[9,10].

$$\frac{\text{output noise variance}}{\text{input noise variance}} = SNR_{exp} = \frac{\alpha}{2-\alpha} \quad . \tag{8-23}$$

† The step response is the averager's output when a string of all zeros followed by a string of all ones is applied to the input.

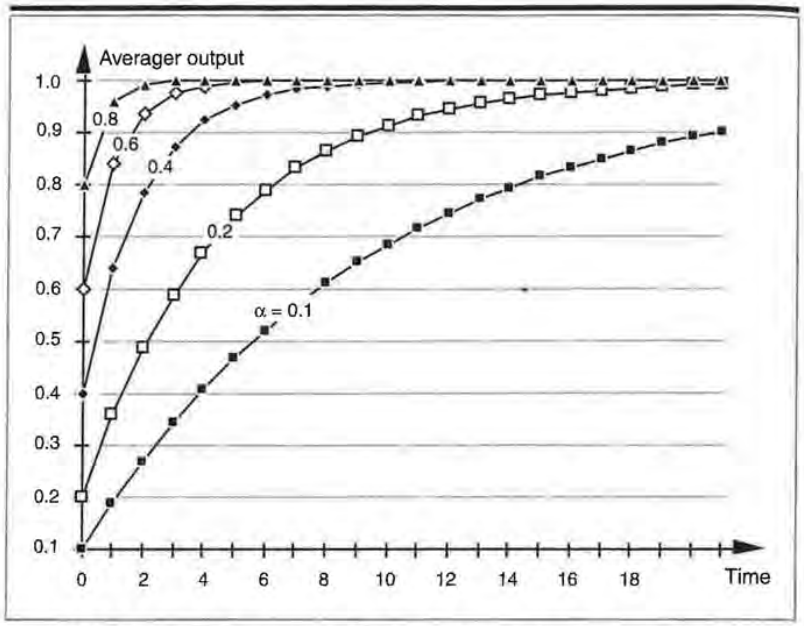


Figure 8-15 Exponential averager output vs. α when a step input is applied at time $t = 0$.

Thus, the exponential averager's noise-power reduction in dB is given by

$$SNR_{\exp_{dB}} = 10 \cdot \log_{10}\left(\frac{\alpha}{2-\alpha}\right). \tag{8-24}$$

Equation (8-24) is plotted in Figure 8-16 to illustrate the trade-off between noise reduction and averager response times.

To demonstrate the exponential averager's noise-power reduction capabilities, Figure 8-17 shows the averager's output with a cosine wave plus noise as an input. The weighting factor α starts out with a value of 1.0 and decreases linearly to a final value of 0.1 at the 180th data input sample. Notice that the noise is reduced as α decreases. However, the cosine wave's peak amplitude also decreases due to the smaller α value.

The reader may recognize the implementation of the exponential averager in Figure 8-13 as a one-tap infinite impulse response (IIR) digital filter[12]. Indeed it is, and, as such, we can determine its frequency response. We do this by remembering the general expression in Chapter 6 for the frequency response of an IIR filter, or Eq. (6-28) repeated here as

$$H_{\exp}(\omega) = \frac{\sum_{k=0}^{N} b(k)\cdot\cos(k\omega) - j\sum_{k=0}^{N} b(k)\cdot\sin(k\omega)}{1 - \sum_{k=1}^{M} a(k)\cdot\cos(k\omega) + j\sum_{k=1}^{M} a(k)\cdot\sin(k\omega)} \quad (8\text{-}25)$$

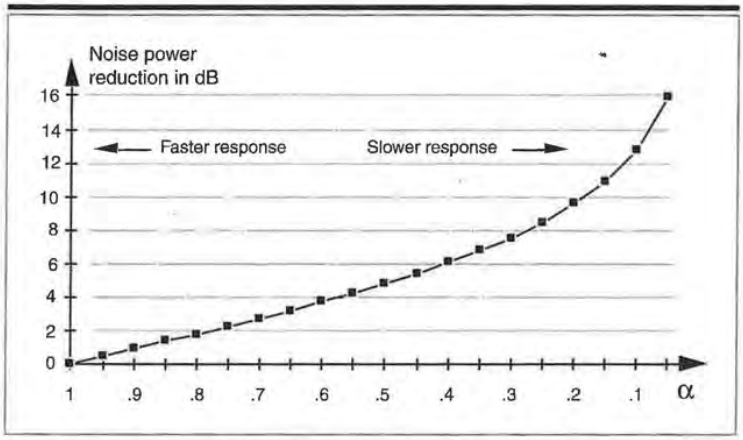


Figure 8-16 Exponential averager noise-power reduction as a function of the weighting factor α.

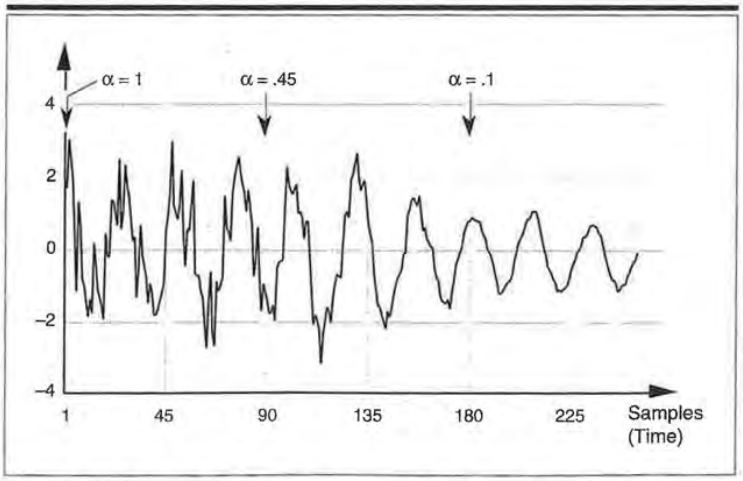


Figure 8-17 Exponential averager output noise reduction as α decreases.

From Figure 6-18, we modify Eq. (8-25) to set $N = 0$, and $M = 1$, so that

$$H_{exp}(\omega) = \frac{b(0)\cdot\cos(0\omega) - jb(0)\cdot\sin(0\omega)}{1 - a(1)\cdot\cos(1\omega) + ja(1)\cdot\sin(1\omega)} \quad . \tag{8-26}$$

Now with $b(0) = \alpha$ and $a(1) = 1 - \alpha$, our exponential averager's frequency response is the complex expression

$$H_{exp}(\omega) = \frac{\alpha}{1 - (1-\alpha)\cdot\cos(\omega) + j(1-\alpha)\cdot\sin(\omega)} \quad . \tag{8-26'}$$

For now, we're interested only in the magnitude response of our filter, so we can express it as

$$\begin{aligned} |H_{exp}(\omega)| &= \left|\frac{\alpha}{1 - (1-\alpha)\cdot\cos(\omega) + j(1-\alpha)\cdot\sin(\omega)}\right| \\ &= \frac{\alpha}{\sqrt{[1-(1-\alpha)\cdot\cos(\omega)]^2 + [(1-\alpha)\cdot\sin(\omega)]^2}} \\ &= \frac{\alpha}{\sqrt{1 - 2\cdot(1-\alpha)\cdot\cos(\omega) + (1-\alpha)^2}} \end{aligned} \tag{8-27}$$

Evaluating Eq. (8-27) over the normalized angular range of $0 \le \omega \le \pi$ (corresponding to a frequency range of $0 \le f \le f_s/2$ Hz), the frequency magnitude responses of our exponential averaging filter for various values of α are shown in Figure 8-18(a) using a normalized decibel scale. Notice that, as α decreases, the exponential averager behaves more and more like a low-pass filter.

We can get some practice manipulating complex numbers by deriving the expression for the phase of an exponential averager. We know that the phase angle, as a function of frequency, is the arctangent of the ratio of the imaginary part of $H_{exp}(\omega)$ over the real part of $H_{exp}(\omega)$, or

$$\varnothing_{exp}(\omega) = \tan^{-1}\left(\frac{\text{imaginary part of } H_{exp}(\omega)}{\text{real part of } H_{exp}(\omega)}\right) \quad . \tag{8-27'}$$

To find those real and imaginary parts, knowing that $H_{exp}(\omega)$ is itself a ratio, we determine what the variables are in Eq. (8-26') corresponding to the form of Eq. (A-20), from Appendix A. Doing that, we get

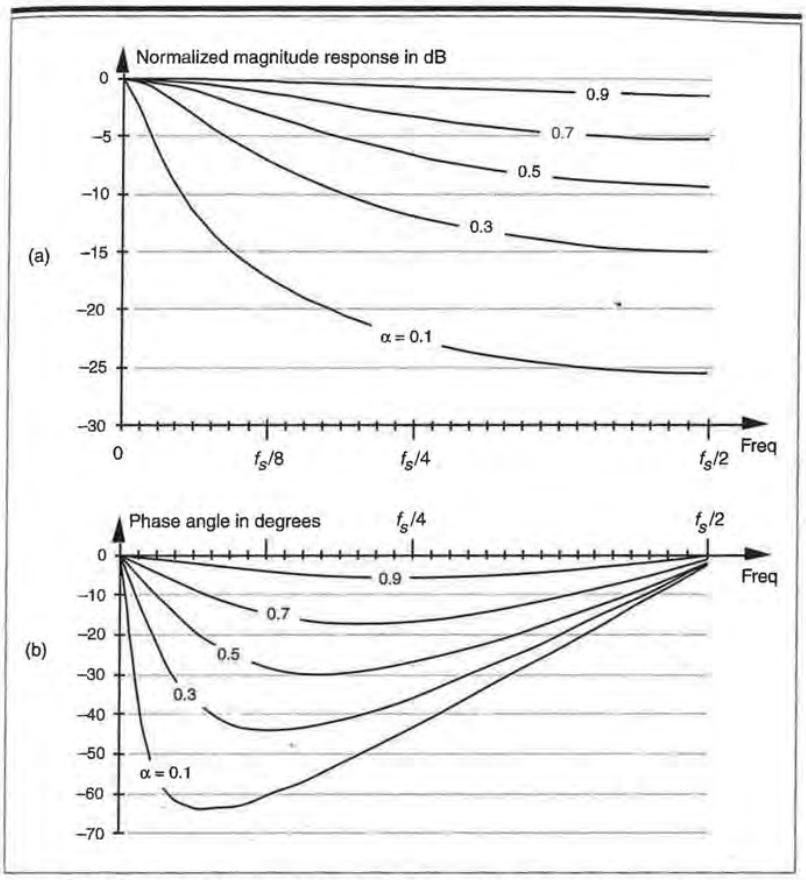


Figure 8-18 Exponential averager frequency response vs. α : (a) normalized magnitude response in dB; (b) phase response in degrees.

$R_1 = \alpha$, $I_1 = 0$, $R_2 = 1 - (1 - \alpha) \cdot \cos(\omega)$, and $I_2 = (1 - \alpha) \cdot \sin(\omega)$.

Substituting those variables in Eq. (A-20), yields

$$H_{exp}(\omega) = \frac{\alpha[1-(1-\alpha)\cdot\cos(\omega)] - j[\alpha(1-\alpha)\cdot\sin(\omega)]}{[1-(1-\alpha)\cdot\cos(\omega)]^2 + [(1-\alpha)\cdot\sin(\omega)]^2} \quad . \tag{8-28}$$

Representing the denominator of this messy Eq. (8-28) with the term *Den*, we use Eq. (8-27) to express the phase angle of $H_{exp}(\omega)$ as

$$\varnothing_{exp}(\omega) = \tan^{-1}\left(\frac{-\alpha(1-\alpha)\cdot\sin(\omega)/Den}{\alpha[1-(1-\alpha)\cdot\cos(\omega)]/Den}\right)$$

$$= \tan^{-1}\left(\frac{-(1-\alpha)\cdot\sin(\omega)}{1-(1-\alpha)\cdot\cos(\omega)}\right). \quad (8\text{-}29)$$

The very nonlinear phase of $\varnothing_{exp}(\omega)$ from Eq. (8-29) is calculated over the normalized angular range of $0 \le \omega \le \pi$, corresponding to a frequency range of $0 \le f \le f_s/2$ Hz, and plotted in Figure 8-18(b).

References

[1] Miller, I., and Freund, J. *Probability and Statistics for Engineers*, 2nd Ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1977, pp. 118.

[2] Beller, J., and Pless, W. "A Modular All-Haul Optical Time-Domain Reflectometer for Characterizing Fiber Links," *Hewlett-Packard Journal*, February 1993.

[3] Spiegel, M. R. *Theory and Problems of Statistics*, Shaum's Outline Series, McGraw-Hill Book Co., New York, 1961, pp. 142.

[4] Papoulis, A. *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill Book Co., New York, 1984, pp. 245.

[5] Davenport, W. B., Jr., and Root, W. L. *Random Signals and Noise*, McGraw-Hill Book Co., New York 1958, pp. 81–84.

[6] Welch, P. D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging over Short, Modified Periodograms," *IEEE Transactions on Audio and Electroacoust.*, Vol. AU-15, No. 2, June 1967

[7] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*, Vol. 66, No. 1, January 1978.

[8] Booster, D. H., et al. "Design of a Precision Optical Low-Coherence Reflectometer," *Hewlett-Packard Journal*, February 1993.

[9] Witte, R. A. "Averaging Techniques Reduce Test Noise, Improve Accuracy," *Microwaves & RF*, February 1988.

[10] Oxaal, J. "Temporal Averaging Techniques Reduce Image Noise," *EDN*, March 17, 1983.

[11] Lymer, A. "Digital-Modulation Scheme Processes RF Broadcast Signals," *Microwaves & RF*, April 1994.

[12] Hayden, D. "Timer Controls DSP-Filter Frequency Resolution," *EDN*, April 13, 1995.

CHAPTER NINE

Digital Data Formats and Their Effects

In digital signal processing, there are many ways to represent numerical data in computing hardware. These representations, known as *data formats*, have a profound effect on the accuracy and ease of implementation of any given signal processing algorithm. The simpler data formats enable uncomplicated hardware designs to be used at the expense of a restricted range of number representation and susceptibility to arithmetic errors. The more elaborate data formats are somewhat difficult to implement in hardware, but they allow us to manipulate very large and very small numbers while providing immunity to many problems associated with digital arithmetic. The data format chosen for any given application can mean the difference between processing success and failure—it's where our algorithmic rubber meets the road.

In this chapter, we'll introduce the most common types of *fixed-point* digital data formats and show why and when they're used. Next, we'll use analog-to-digital (A/D) converter operation to establish the precision and dynamic range afforded by these fixed-point formats along with the inherent errors encountered with their use. Finally, we'll cover the interesting subject of *floating-point* binary formats.

9.1 Fixed-Point Binary Formats

Within digital hardware, numbers are represented by binary digits known as bits—in fact, the term *bit* originated from the words *Binary*

digIT. A single bit can be in only one of two possible states: either a one or a zero.† A six-bit binary number could, for example, take the form 101101, with the leftmost bit known as the *most significant bit* (msb), while the rightmost bit is called the *least significant bit* (lsb). The number of bits in a binary number is known as the word length—hence 101101 has a word length of six. Like the decimal number system so familiar to us, the binary number system assumes a weight associated with each digit in the number. That weight is the base of the system (two for binary numbers and ten for decimal numbers) raised to an integral power. To illustrate this with a simple example, the decimal number 4631 is

$$(\mathbf{4} \cdot 10^3) + (\mathbf{6} \cdot 10^2) + (\mathbf{3} \cdot 10^1) + (\mathbf{1} \cdot 10^0)$$

$$= 4000 + 600 + 30 + 1 = 4631\,. \qquad (9\text{-}1)$$

The factors, 10^3, 10^2, 10^1, and 10^0 are the digit weights in Eq. (9-1). Similarly, the six-bit binary number 101101 is equal to decimal 45 as shown by

$$(\mathbf{1} \cdot 2^5) + (\mathbf{0} \cdot 2^4) + (\mathbf{1} \cdot 2^3) + (\mathbf{1} \cdot 2^2) + (\mathbf{0} \cdot 2^1) + (\mathbf{1} \cdot 2^0)$$

$$= 32 + 8 + 4 + 1 = 45\,. \qquad (9\text{-}2)$$

Using subscripts to signify the base of a number, we can write Eq. (9-2) as $101101_2 = 45_{10}$. Equation (9-2) shows us that, like decimal numbers, binary numbers use the *place value* system where the position of a digit signifies its weight. If we use B to denote a number system's base, the place value representation of the four-digit number $a_3a_2a_1a_0$ is

$$(a_3 \cdot B^3) + (a_2 \cdot B^2) + (a_1 \cdot B^1) + (a_0 \cdot B^0)\,. \qquad (9\text{-}3)$$

In Eq. (9-3), B^n is the weight multiplier for the digit a_n, where $0 \le a_n \le B\text{–}1$. (This place value system of representing numbers is very old—so old, in fact, that it's origin is obscure. However, with its inherent positioning of the decimal or binary point, this number system is so convenient and powerful that its importance has been compared to that of the alphabet[1].)

† Binary numbers are used because early electronic computer pioneers quickly realized that it was much more practical and reliable to use electrical devices (relays, vacuum tubes, transistors, etc.) that had only two states, *on* or *off*. Thus, the on/off state of a device could represent a single binary digit.

9.1.1. Octal Numbers

As the use of minicomputers and microprocessors rapidly expanded in the 1960s, people grew tired of manipulating long strings of ones and zeros on paper and began to use more convenient ways to represent binary numbers. One way to express a binary number is an octal format with its base of eight. Converting from binary to octal is as simple as separating the binary number into three-bit groups starting from the right. For example, the binary number 10101001_2 can be converted to octal format as

$$10101001_2 \rightarrow \quad 10_{|}101_{|}001 = 251_8 .$$

Each of the three groups of bits above are easily converted from their binary formats to a single octal digit because, for three-bit words, the octal and decimal formats are the same. That is, starting with the left group of bits, $10_2 = 2_{10} = 2_8$, $101_2 = 5_{10} = 5_8$, and $001_2 = 1_{10} = 1_8$. The octal format also uses the place value system meaning that $251_8 = (\mathbf{2} \cdot 8^2 + \mathbf{5} \cdot 8^1 + \mathbf{1} \cdot 8^0)$. Octal format enables us to represent the eight-digit 10101001_2 with the three-digit 251_8. Of course, the only valid digits in the octal format are 0 to 7—the digits 8 and 9 have no meaning in octal representation.

9.1.2. Hexadecimal Numbers

Another popular binary format is the hexadecimal number format using 16 as its base. Converting from binary to hexadecimal is done, this time, by separating the binary number into four-bit groups starting from the right. The binary number 10101001_2 is converted to hexadecimal format as

$$10101001_2 \rightarrow \quad 1010_{|}1001 = A9_{16} .$$

If you haven't seen the hexadecimal format used before, don't let the $A9$ digits confuse you. In this format, the characters A, B, C, D, E, and F represent the digits whose decimal values are 10, 11, 12, 13, 14, and 15 respectively. We convert the two groups of bits above to two hexadecimal digits by starting with the left group of bits, $1010_2 = 10_{10} = A_{16}$, and $1001_2 = 9_{10} = 9_{16}$. Hexadecimal format numbers also use the place value system, meaning that $A9_{16} = (\mathbf{A} \cdot 16^1 + 9 \cdot 16^0)$. For convenience, then, we can represent the eight-digit 10101001_2 with the two-digit number $A9_{16}$. Table 9-1 lists the permissible digit representations in the number systems discussed thus far.

Table 9-1 Allowable Digit Representations vs. Number System Base

Binary	*Octal*	*Decimal*	*Hexadecimal*	*Decimal Equivalent*
0	0	0	0	0
1	1	1	1	1
	2	2	2	2
	3	3	3	3
	4	4	4	4
	5	5	5	5
	6	6	6	6
	7	7	7	7
		8	8	8
		9	9	9
			A	10
			B	11
			C	12
			D	13
			E	14
			F	15

9.1.3. Fractional Binary Numbers

Fractions (numbers whose magnitudes are greater than zero and less than one) can also be represented by binary numbers if we use a binary point identical in function to our familiar decimal point. In the binary numbers we've discussed so far, the binary point is assumed to be fixed just to the right of the rightmost digit. Using the symbol ◊ to denote the binary point, the six-bit binary fraction $11_{\lozenge}0101$ is equal to decimal 3.3125 as shown by

$$(\mathbf{1}\cdot 2^{1})+(\mathbf{1}\cdot 2^{0})+(\mathbf{0}\cdot 2^{-1})+(\mathbf{1}\cdot 2^{-2})+(\mathbf{0}\cdot 2^{-3})+(\mathbf{1}\cdot 2^{-4})$$

$$=(\mathbf{1}\cdot 2)+(\mathbf{1}\cdot 1)+(\mathbf{0}\cdot \frac{1}{2})+(\mathbf{1}\cdot \frac{1}{4})+(\mathbf{0}\cdot \frac{1}{8})+(\mathbf{1}\cdot \frac{1}{16})$$

$$=2+1+0+0.25+0+0.0625=3.3125 \quad . \qquad (9\text{-}4)$$

For the example in Eq. (9-4), the binary point is set between the second and third most significant bits. Having a stationary position for the binary point is why binary numbers are often called *fixed-point* binary.

For some binary number formats (like the floating-point formats that we'll cover shortly), the binary point is fixed just to the left of the most significant bit. This forces the number values to be restricted to the range between zero and one. In this format, the largest and smallest values possible for a b-bit fractional word are $1-2^{-b}$ and 2^{-b}, respectively. Taking a six-bit binary fraction, for example, the largest value is $_{0}111111_{2}$, or

$$
\begin{aligned}
&{}_{0}(\mathbf{1}\cdot 2^{-1})+(\mathbf{1}\cdot 2^{-2})+(\mathbf{1}\cdot 2^{-3})+(\mathbf{1}\cdot 2^{-4})+(\mathbf{1}\cdot 2^{-5})+(\mathbf{1}\cdot 2^{-6}) \\
&={}_{0}(\mathbf{1}\cdot\frac{1}{2})+(\mathbf{1}\cdot\frac{1}{4})+(\mathbf{1}\cdot\frac{1}{8})+(\mathbf{1}\cdot\frac{1}{16})+(\mathbf{1}\cdot\frac{1}{32})+(\mathbf{1}\cdot\frac{1}{64}) \\
&=0.5+0.25+0.125+0.0625+0.03125+0.015625=0.984375\ , \qquad (9\text{-}5)
\end{aligned}
$$

which is $1-2^{-6} = 1-{}^{1}/_{64}$ in decimal. The smallest nonzero value is $_{0}000001_{2}$, equaling a decimal ${}^{1}/_{64} = 0.015625_{10}$.

9.1.4. Sign-Magnitude Binary Format

For binary numbers to be at all useful in practice, they must be able to represent negative values. Binary numbers do this by dedicating one of the bits in a binary word to indicate the sign of a number. Let's consider a popular binary format known as sign-magnitude. Here, we assume that a binary word's leftmost bit is a sign bit and the remaining bits represent the magnitude of a number which is always positive. For example, we can say that the four-bit number 0011_2 is $+3_{10}$ and the binary number 1011_2 is equal to -3_{10}, or

magnitude bits ↓↓↓ $0011_2 = 3_{10}$, ↑ sign bit of zero signifies positive

and

magnitude bits ↓↓↓ $1011_2 = -3_{10}$. ↑ sign bit of one signifies negative

Of course, using one of the bits as a sign bit reduces the magnitude of the numbers we can represent. If an unsigned binary number's word length is b bits, the number of different values that can be represented is 2^b. An eight-bit word, for example, can represent $2^8 = 256$ different integral values. With zero being one of the values we have to express, a b-bit unsigned binary word can represent integers from 0 to 2^b-1. The largest value represented by an unsigned eight-bit word is $2^8-1 = 255_{10} = 11111111_2$. In the sign-magnitude binary format a b-bit word can represent only a magnitude of $\pm 2^{b-1}-1$, so the largest positive or negative value we can represent by an eight-bit sign-magnitude word, then, is $\pm 2^{8-1}-1 = \pm 127$.

9.1.5 Two's Complement Format

Another common binary number scheme, known as the *two's complement* format, also uses the leftmost bit as a sign bit. The two's complement format is the most convenient numbering scheme from a hardware design standpoint and has been used for decades. It enables computers to perform both addition and subtraction using the same hardware adder logic. To get the negative version of a positive two's complement number, we merely complement (change a one to a zero and change a zero to a one) each bit and add a one to the complemented word. For example, with 0011_2 representing a decimal 3 in two's complement format, we obtain a negative decimal 3 through the following steps:

```
+3 in two's complement →    0 0 1 1

complement of +3 →          1 1 0 0
add one →                 + 0 0 0 1
-3 in two's complement →    1 1 0 1.
```

In the two's complement format, a b-bit word can represent positive amplitudes as great as $2^{b-1}-1$, and negative amplitudes as large as -2^{b-1}. Table 9-2 shows four-bit word examples of sign-magnitude and two's complement binary formats.

While using two's complement numbers, we have to be careful when adding two numbers of different word lengths. Consider the case where a four-bit number is added to a eight-bit number:

```
+15 in two's complement →    0 0 0 0 1 1 1 1
add +3 in two's complement →           + 0 0 1 1
+18 in two's complement →    0 0 0 1 0 0 1 0.
```

Table 9-2 Examples of Binary Number Formats

Sign-Magnitude	*Two's Complement*	*Offset Binary*	*Decimal Equivalent*
0111	0111	1111	7
0110	0110	1110	6
0101	0101	1101	5
0100	0100	1100	4
0011	0011	1011	3
0010	0010	1010	2
0001	0001	1001	1
0000	0000	1000	0
1001	1111	0111	–1
1010	1110	0110	–2
1011	1101	0101	–3
1100	1100	0100	–4
1101	1011	0011	–5
1110	1010	0010	–6
1111	1001	0001	–7
–	1000	0000	–8

No problem so far. The trouble occurs when our four-bit number is negative. Instead of adding a +3 to the +15, let's try to add a –3 to the +15:

```
+15 in two's complement →  0 0 0 0 1 1 1 1
add a –3 in two's complement →     + 1 1 0 1
+20 in two's complement →  0 0 0 1 0 1 0 0   . ← Wrong answer
```

The above arithmetic error can be avoided by performing what's called a *sign-extend* operation on the four-bit number. This process, typically performed automatically in hardware, extends the sign bit of the four-bit negative number to the left, making it an eight-bit negative number. If we sign-extend the –3 and, then perform the addition, we'll get the correct answer:

```
+15 in two's complement →  0 0 0 0 1 1 1 1
add a sign-extended –3 in two's complement → + 1 1 1 1 1 1 0 1
+12 in two's complement → 1 0 0 0 0 1 1 0 0   . ← That's better
                         ↑
                  overflow bit is ignored
```

9.1.6. Offset Binary Format

Another useful binary number scheme is known as the *offset binary* format. While this format is not as common as two's complement, it still shows up in some hardware devices. Table 9-2 shows offset binary format examples for four-bit words. Offset binary represents negative numbers by subtracting 2^{b-1} from an unsigned binary value. For example, in the second row of Table 9-2, the offset binary number is 1110_2. When this number is treated as an unsigned binary number it's equivalent to 14_{10}. For four-bit words $b = 4$ and $2^{b-1} = 8$, so $14_{10} - 8_{10} = 6_{10}$ which is the decimal equivalent of 1110_2 in offset binary. The difference between the unsigned binary equivalent and the actual decimal equivalent of the offset binary numbers in Table 9-2 is always –8. This kind of offset is sometimes referred to as a *bias* when the offset binary format is used. (It may interest the reader that we can convert back and forth between the two's complement and offset binary formats merely by complementing a word's most significant bit.)

The history, arithmetic, and utility of the many available number formats is a very broad field of study. A thorough and very readable discussion of the subject is given by Knuth in reference [2].

9.2 Binary Number Precision and Dynamic Range

As we implied before, for any binary number format, the number of bits in a data word is a key consideration. The more bits used in the word, the better the resolution of the number, and the larger the maximum value that can be represented.† Assuming that a binary word represents the amplitude of a signal, digital signal processing practitioners find it useful to quantify the dynamic range of various binary number schemes. For a signed integer binary word length of b+1 bits (one sign bit and b magnitude bits), the dynamic range in decibels is defined by

$$\text{dynamic range}_{\text{dB}} = 20 \cdot \log_{10}\left(\frac{\text{largest possible word value}}{\text{smallest possible word value}}\right)$$

$$= 20 \cdot \log_{10}\left(\frac{2^b - 1}{1}\right) = 20 \cdot \log_{10}(2^b - 1) \quad . \tag{9-6}$$

† Some computers use 64-bit words. Now 2^{64} is approximately equal to $1.8 \cdot 10^{19}$—that's a pretty large number. So large, in fact, that if we started incrementing a 64-bit counter once per second at the beginning of the universe (≈20 billion years ago), the most significant four bits of this counter would *still* be all zeros today.

When 2^b is much larger than 1, we can ignore the –1 in Eq. (9-6) and state that

$$\text{dynamic range}_{\text{dB}} = 20 \cdot \log_{10}(2^b)$$

$$= 20 \cdot \log_{10}(2) \cdot b = 6.02 \cdot b \text{ dB}. \quad (9\text{-}6')$$

Equation (9-6'), dimensioned in dB, tells us that the dynamic range of our number system is directly proportional to the word length. Thus, an eight-bit two's complement word, with seven bits available to represent signal magnitude, has a dynamic range of $6.02 \cdot 7 = 42.14$ dB. Most people simplify Eq. (9-6') by using the rule of thumb that the dynamic range is equal to "six dB per bit."

9.3 Effects of Finite Fixed-Point Binary Word Length

The effects of finite binary word lengths touches all aspects of digital signal processing. Using finite word lengths prevents us from representing values with infinite precision, increases the background noise in our spectral estimation techniques, creates nonideal digital filter responses, induces noise in analog-to-digital (A/D) converter outputs, and can (if we're not careful) lead to wildly inaccurate arithmetic results. The smaller the word lengths, the greater these problems will be. Fortunately, these finite, word-length effects are rather well understood. We can predict their consequences and take steps to minimize any unpleasant surprises. The first finite, word-length effect we'll cover is the errors that occur during the A/D conversion process.

9.3.1 A/D Converter Quantization Errors

Practical A/D converters are constrained to have binary output words of finite length. Commercial A/D converters are categorized by their output word lengths which are normally in the range from 8 to 16 bits. A typical A/D converter input analog voltage range is from –1 to +1 volt. If we used such an A/D converter having eight-bit output words, the least significant bit would represent

$$\text{lsb value} = \frac{\text{full voltage range}}{2^{\text{word length}}} = \frac{2 \text{ volts}}{2^8} = 7.81 \text{ millivolts}. \quad (9\text{-}7)$$

What this means is that we can represent continuous (analog) voltages perfectly as long as they're integral multiples of 7.81 millivolts—any

intermediate input voltage will cause the A/D converter to output a *best estimate* digital data value. The inaccuracies in this process are called *quantization errors* because an A/D output least significant bit is an indivisible quantity. We illustrate this situation in Figure 9-1(a) where the continuous waveform is being digitized by an eight-bit A/D converter whose output is in the sign-magnitude format. When we start sampling at time $t = 0$, the continuous waveform happens to have a value of 31.25 millivolts (mv), and our A/D output data word will be exactly correct for sample $x(0)$. At time T when we get the second A/D output word for sample $x(1)$, the continuous voltage is between 0 and –7.81 mv. In this case, the A/D converter outputs a sample value of 10000001 representing –7.81 mv, even though the continuous input was not quite as negative as –7.81mv. The 10000001

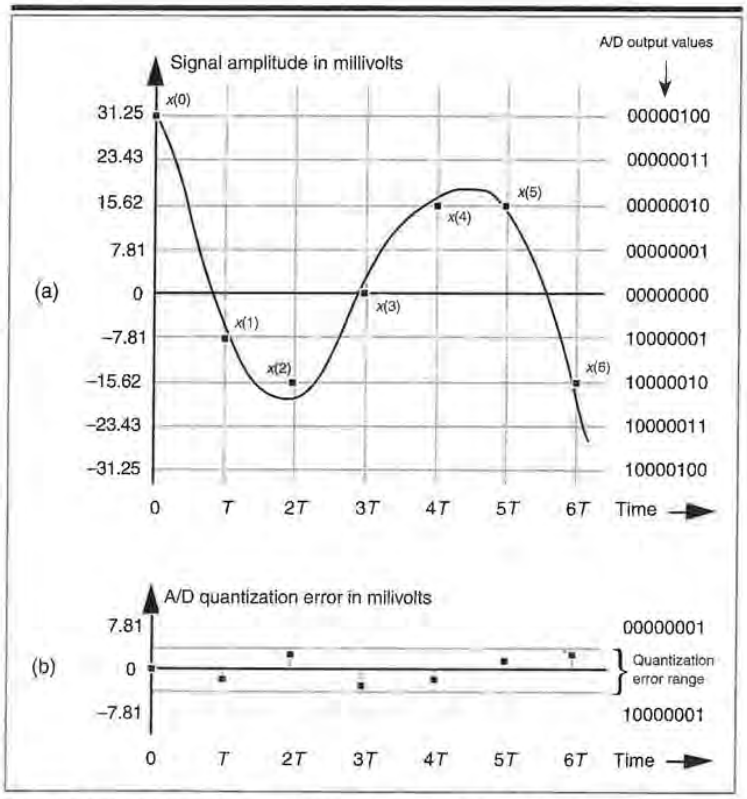


Figure 9-1 Quantization errors: (a) digitized *x(n)* values of a continuous signal; (*b*) quantization error between the actual analog signal values and the digitized signal values.

A/D output word contains some quantization error. Each successive sample contains quantization error because the A/D's digitized output values must lie on a horizontal dashed line in Figure 9-1(a). The difference between the actual continuous input voltage and the A/D converter's representation of the input is shown as the quantization error in Figure 9-1(b). For an ideal A/D converter, the quantization error, a kind of *roundoff* noise, can never be greater than ±1/2 an lsb, or ±3.905 mv.

While Figure 9-1(b) shows A/D quantization noise in the time domain, we can also illustrate this noise in the frequency domain. Figure 9-2(a) depicts a continuous sinewave of one cycle over the sample interval shown as the dotted line and a quantized version of the time-domain samples of that wave as the dots. Notice how the quantized version of the wave is constrained to have only integral values, giving it a *stair step* effect oscillating above and below the true unquantized sinewave. The quantization here is 4 bits, meaning that we have a sign bit and three bits to represent the magnitude of the wave. With three bits, the maximum peak values for the wave are ±7. Figure 9-2(b) shows the discrete Fourier transform (DFT) of a discrete version of the sinewave whose time-domain sample values are not forced to be integers, but have high precision. Notice in this case that the DFT has a nonzero value only at $m = 1$. On the other hand, Figure 9-2(c) shows the spectrum of the 4-bit quantized samples in Figure 9-2(a) where quantization effects have induced noise components across the entire spectral band. If the quantization noise depictions in Figures 9-1(b) and 9-2(c) look random, that's because they are. As it turns out, even though A/D quantization noise is random, we can still quantify its effects in a useful way.

In the field of communications, people often use the notion of output signal-to-noise ratio, or SNR = (signal power)/(noise power), to judge the usefulness of a process or device. We can do likewise and obtain an important expression for the output SNR of an ideal A/D converter, $SNR_{A/D}$, accounting for finite word-length quantization effects. Because quantization noise is random, we can't explicitly represent its power level, but we can use its statistical equivalent of variance to define $SNR_{A/D}$ measured in decibels as

$$SNR_{A/D} = 10 \cdot \log_{10}\left(\frac{\text{input signal variance}}{\text{A/D quantization noise variance}}\right)$$

$$= 10 \cdot \log_{10}\left(\frac{\sigma^2_{\text{signal}}}{\sigma^2_{\text{A/D noise}}}\right). \quad (9\text{-}8)$$

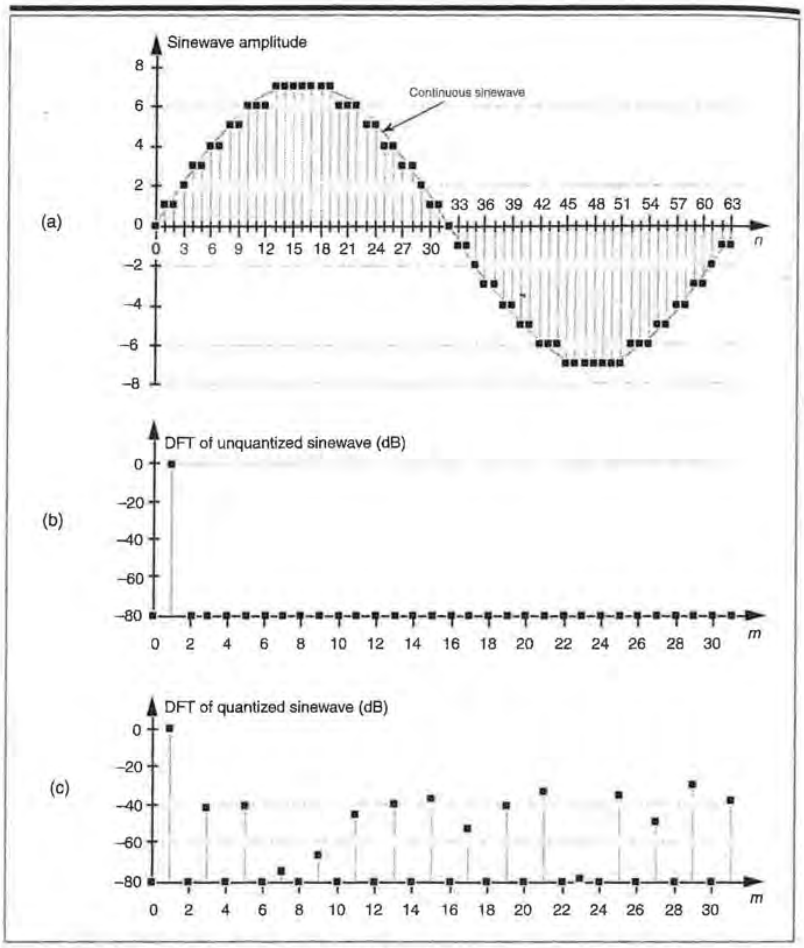


Figure 9-2 Quantization noise effects: (a) input sinewave applied to a 64-point DFT; (b) theoretical DFT magnitude of high-precision sinewave samples; (c) DFT magnitude of a sinewave quantized to 4 bits.

Next, we'll determine an A/D converter's quantization noise variance relative to the converter's maximum input peak voltage V_p. If the full scale ($-V_p$ to $+V_p$ volts) continuous input range of a b-bit A/D converter is $2Vp$, a single quantization level q is that voltage range divided by the number of possible A/D output binary values, or $q = 2V_p/2^b$. (In Figure 9-1, for example, the quantization level q is the lsb value of 7.81 mv.) A depiction of the likelihood of encountering any given quantization error value, called the probability density function $p(e)$ of the quantization error, is shown in Figure 9-3.

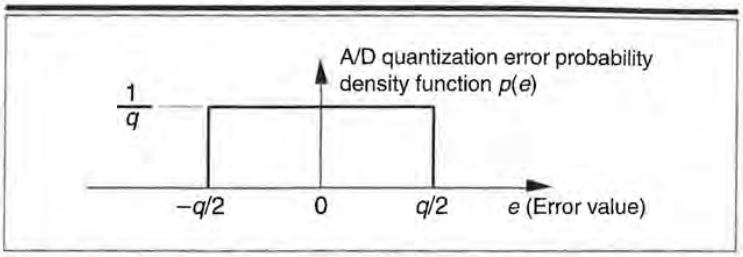


Figure 9-3 Probability density function of A/D conversion roundoff error (noise).

This simple rectangular function has much to tell us. It indicates that there's an equal chance that any error value between $-q/2$ and $+q/2$ can occur. By definition, because probability density functions have an area of unity (i.e., the probability is 100 percent that the error will be somewhere under the curve), the amplitude of the $p(e)$ density function must be the area divided by the width, or $p(e) = 1/q$. From Figure D-4 and Eq. (D-12) in Appendix D, the variance of our uniform $p(e)$ is

$$\sigma^2_{\text{A/D noise}} = \int_{-q/2}^{q/2} e^2 p(e)de = \frac{1}{q} \cdot \int_{-q/2}^{q/2} e^2 de = \frac{q^2}{12} . \tag{9-9}$$

We can now express the A/D noise error variance in terms of A/D parameters by replacing q in Eq. (9-9) with $q = 2V_p/2^b$ to get

$$\sigma^2_{\text{A/D noise}} = \frac{(2V_p)^2}{12 \cdot (2^b)^2} = \frac{V_p^2}{3 \cdot 2^{2b}} . \tag{9-10}$$

OK, we're halfway to our goal—with Eq. (9-10) giving us the denominator of Eq. (9-8), we need the numerator. To arrive at a general result, let's express the input signal in terms of its root mean square (rms), the A/D converter's peak voltage, and a loading factor LF defined as

$$LF = \frac{\text{rms of the input signal}}{V_p} = \frac{\sigma_{\text{signal}}}{V_p} .^{\dagger} \tag{9-11}$$

† Recall from Section D.2 that, although the variance σ^2 is associated with the power of a signal, the standard deviation σ is associated with the rms value of a signal.

With the loading factor defined as the input rms voltage over the A/D converter's peak input voltage, we square and rearrange Eq. (9-11) to show the signal variance σ^2_{signal} as

$$\sigma^2_{signal} = (LF)^2 V_p^2 \ . \tag{9-12}$$

Substituting Eqs. (9-10) and (9-12) in Eq. (9-8),

$$\text{SNR}_{A/D} = 10 \cdot \log_{10}\left(\frac{(LF)^2 V_p^2}{V_p^2/(3 \cdot 2^{2b})}\right) = 10 \cdot \log_{10}[(LF)^2 \cdot (3 \cdot 2^{2b})]$$

$$= 6.02 \cdot b + 4.77 + 20 \cdot \log_{10}(LF) \ . \tag{9-13}$$

Eq. (9-13) gives us the $SNR_{A/D}$ of an ideal b-bit A/D converter in terms of the loading factor and the number of bits b. Figure 9-4 plots Eq. (9-13) for various A/D word lengths as a function of the loading factor. Notice that the loading factor in Figure 9-4 is never greater than –3dB, because the maximum continuous A/D input peak value must not be greater V_p volts. Thus, for a sinusoid input, its rms value must not be greater than $V_p/\sqrt{2}$ volts (3 dB below V_p).

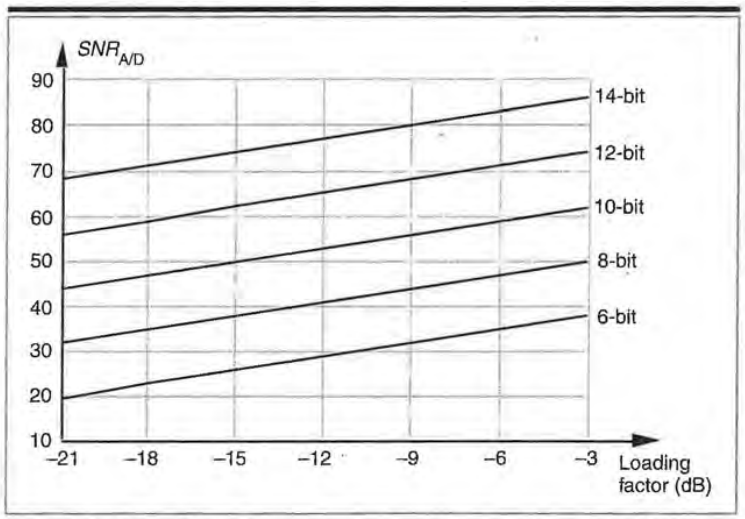


Figure 9-4 $SNR_{A/D}$ of ideal A/D converters as a function of loading factor in dB.

When the input sinewave's peak amplitude is equal to the A/D converter's full-scale voltage V_p, the full-scale LF is

$$LF_{\text{full scale}} = \frac{V_p / \sqrt{2}}{V_p} = \frac{1}{\sqrt{2}} \,. \tag{9-14}$$

Under this condition, the maximum A/D output SNR from Eq. (9-13) is

$$SNR_{\text{A/D-max}} = 6.02 \cdot b + 4.77 + 20 \cdot \log_{10}(1/\sqrt{2})$$

$$= 6.02 \cdot b + 4.77 - 3.01 = 6.02 \cdot b + 1.76 \text{ dB} \,. \tag{9-15}$$

This discussion of SNR relative to A/D converters means three important things to us:

- An ideal A/D converter will have an $SNR_{\text{A/D}}$ defined by Eq. (9-13), so any continuous signal we're trying to digitize with a b-bit A/D converter will never have an $SNR_{\text{A/D}}$ greater than Eq. (9-13) after A/D conversion. For example, let's say we want to digitize a continuous signal whose SNR is 55 dB. Using an ideal eight-bit A/D converter with its full-scale $SNR_{\text{A/D}}$ of $6.02 \cdot 8 + 1.76 = 49.9$ dB from Eq. (9-15), the quantization noise will contaminate the digitized values, and the resultant digital signal's SNR can be no better than 49.9 dB. We'll have lost signal SNR through the A/D conversion process. (A ten-bit A/D, with its ideal $SNR_{\text{A/D}} \approx 62$ dB, could be used to digitize a 55 dB SNR continuous signal to reduce the SNR degradation caused by quantization noise.) Equations (9-13) and (9-15) apply to ideal A/D converters and don't take into account such additional A/D noise sources as aperture jitter error, missing output bit patterns, and other nonlinearities. So actual A/D converters are likely to have SNRs that are lower than that indicated by theoretical Eq. (9-13). To be safe in practice, it's sensible to assume that $SNR_{\text{A/D-max}}$ is 3 to 6 dB lower than that indicated by Eq. (9-15).
- Equation (9-15) is often expressed in the literature, but it can be a little misleading because it's imprudent to force an A/D converter's input to full scale. It's wise to drive an A/D converter to some level below full scale because inadvertent overdriving will lead to signal clipping and will induce distortion in the A/D's output. So Eq. (9-15) is overly optimistic, and, in practice, A/D converter SNRs will be less

than that indicated by Eq. (9-15). The best approximation for an A/D's SNR is to determine the input signal's rms value that will never (or rarely) overdrive the converter input, and plug that value in Eq. (9-11) to get the loading factor value for use in Eq. (9-13).† Again, using an A/D converter with a wider word length will alleviate this problem by increasing the available $SNR_{A/D}$.

- Remember now, real-world continuous signals always have their own inherent continuous SNR, so using an A/D converter, whose $SNR_{A/D}$ is a great deal larger than the continuous signal's SNR serves no purpose. In this case, we'd be using the A/D converter's extra bits to digitize the continuous signal's noise to a greater degree of accuracy.

A word of caution is appropriate here concerning our analysis of A/D converter quantization errors. The derivations of Eqs. (9-13) and (9-15) are based upon three assumptions:

- The cause of A/D quantization errors is a stationary random process; that is, the performance of the A/D converter does not change over time. Given the same continuous input voltage, we always expect an A/D converter to provide exactly the same output binary code.
- The probability density function of the A/D quantization error is uniform. We're assuming that the A/D converter is ideal in its operation and all possible errors between $-q/2$ and $+q/2$ are equally likely. An A/D converter having stuck bits or missing output codes would violate this assumption. High-quality A/D converters being driven by continuous signals that cross many quantization levels will result in our desired uniform quantization noise probability density function.
- The A/D quantization errors are uncorrelated with the continuous input signal. If we were to digitize a single continuous sinewave whose frequency was harmonically related to the A/D sample rate, we'd end up sampling the same input voltage repeatedly and the quantization error sequence would not be random. The quantization

† By the way, some folks use the term *crest factor* to describe how hard an A/D converter's input is being driven. The crest factor is the reciprocal of the loading factor, or $CF = V_p/$(rms of the input signal).

error would be predictable and repetitive, and our quantization noise variance derivation would be invalid. In practice, complicated continuous signals such as music or speech, with their rich spectral content, avoid this problem.

To conclude our discussion of A/D converters, let's consider one last topic. In the literature the reader is likely to encounter the expression

$$b_{\text{eff}} = \frac{SNR - 1.76}{6.02} \tag{9-16}$$

Equation (9-16) is used by test equipment manufacturers to specify the sensitivity of test instruments using a b_{eff} parameter known as the number of *effective bits*, or effective number of bits (ENOB) [3–8]. Equation (9-16) is merely Eq. (9-15) solved for b. Test equipment manufacturers measure the actual SNR of their product indicating its ability to capture continuous input signals relative to the instrument's inherent noise characteristics. Given this true SNR, they use Eq. (9-16) to determine the b_{eff} value for advertisement in their product literature. The larger b_{eff}, the greater the continuous voltage that can be accurately digitized relative to the equipment's intrinsic quantization noise.

9.3.2 Data Overflow

The next finite, word-length effect we'll consider is called *overflow*. Overflow is what happens when the result of an arithmetic operation has too many bits, or digits, to be represented in the hardware registers designed to contain that result. We can demonstrate this situation to ourselves rather easily using a simple four-function, eight-digit pocket calculator. The sum of a decimal 9.9999999 plus 1.0 is 10.9999999, but on an eight-digit calculator the sum is 10.999999 as

```
  9.9999999
+ 1.0000000
 10.9999999.
           ↑
  this digit gets discarded
```

The hardware registers, which contain the arithmetic result, and drive the calculator's display, can hold only eight decimal digits; so the least significant (of course) digit is discarded. Although the above error is less

than one part in ten million, overflow effects can be striking when we work with large numbers. If we use our calculator to add 99,999,999 plus 1, instead of getting the correct result of 100 million, we'll get a result of 1. Now that's an authentic overflow error!

Let's illustrate overflow effects with examples more closely related to our discussion of binary number formats. First, adding two unsigned binary numbers is as straightforward as adding two decimal numbers. The sum of 42 plus 39 is 81, or

```
                          1 1 1   ← carry bits
+42 in unsigned binary →  1 0 1 0 1 0
+39 in unsigned binary → +1 0 0 1 1 1
+81 in unsigned binary →  1 0 1 0 0 0 1 .
```

In this case, two 6-bit binary numbers required 7 bits to represent the results. The general rule is *the sum of* m *individual* b*-bit binary numbers can require as many as* $[b + log_2(m)]$ *bits to represent the results.* So, for example, a 24-bit result register (accumulator) is needed to accumulate the sum of sixteen 20-bit binary numbers, or $20 + \log_2(16) = 24$. The sum of 256 eight-bit words requires an accumulator whose word length is $[8 + \log_2(256)]$, or 16 bits, to ensure that no overflow errors occur.

In the preceding example, if our accumulator word length was 6 bits, an overflow error occurs as

```
                          1 1 1   ← carry bits
+42 in unsigned binary →  1 0 1 0 1 0
+39 in unsigned binary → +1 0 0 1 1 1
+17 in unsigned binary →  1 0 1 0 0 0 1 .   ← Overflow error
                          ↑
an overflow out of the sign bit is ignored, causing an overflow error
```

Here, the most significant bit of the result overflowed the 6-bit accumulator, and an error occurred.

With regard to overflow errors, the two's complement binary format has two interesting characteristics. First, under certain conditions, overflow during the summation of two numbers causes no error. Second, with multiple summations, intermediate overflow errors cause no problems if the final magnitude of the sum of the *b*-bit two's complement numbers is less than 2^{b-1}. Let's illustrate these properties by considering the four-bit two's complement format in Figure 9-5, whose binary values are taken from Table 9-2.

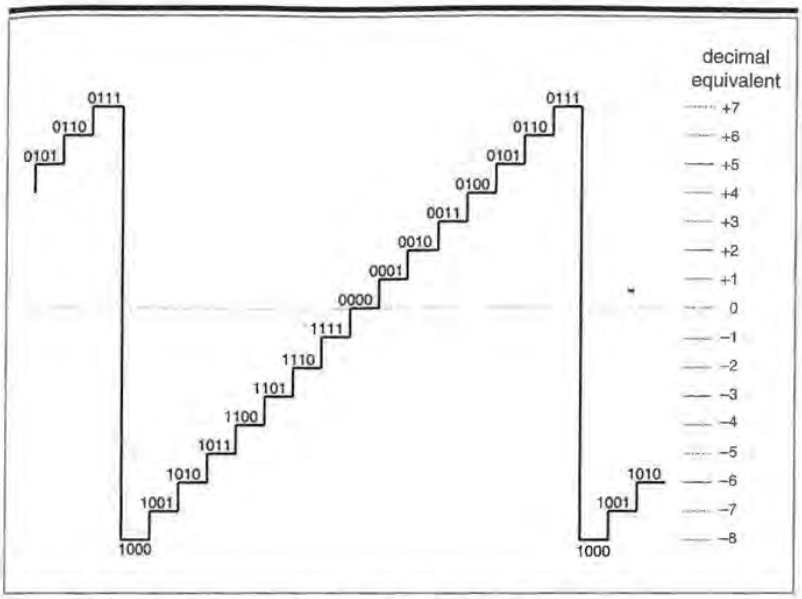


Figure 9-5 Four-bit two's complement binary numbers.

The first property of two's complement overflow, which sometimes causes no errors, can be shown by the following examples:

```
                                  0 1 0    ← carry bits
–5 in two's complement  →         1 0 1 1
+2 in two's complement  →       + 0 0 1 0
–3 in two's complement  →         0 1 1 0 1   ← valid negative result
                                  ↑
                  zero overflow out of the sign bit
```

```
                                  1 1 0    ← carry bits
–2 in two's complement  →         1 1 1 0
+6 in two's complement  →       + 0 1 1 0
+4 in two's complement  →         1 0 1 0 0   ← valid positive result
                                  ↑
                  overflow out of the sign bit ignored, no harm done
```

Then again, the following examples show how two's complement overflow sometimes does cause errors:

```
                                  0 0 0        ← carry bits
–7 in two's complement  →         1 0 0 1
–6 in two's complement  →       + 1 0 1 0
+3 in two's complement  →         1 0 0 1 1    ← invalid positive result
                                  ↑
overflow out of the sign bit ignored, causing overflow error
```

```
                                  1 1 1        ← carry bits
+7 in two's complement  →          0 1 1 1
+7 in two's complement  →        + 0 1 1 1
–2 in two's complement  →          0 1 1 1 0  ← invalid negative result
                                   ↑
zero overflow out of the sign bit
```

The rule with two's complement addition is *if the carry bit into the sign bit is the same as the overflow bit out of the sign bit, the overflow bit can be ignored causing no errors; if the carry bit into the sign bit is different from the overflow bit out of the sign bit, the result is invalid.* An even more interesting property of two's complement numbers is that a series of b-bit word summations can be performed where intermediate sums are invalid, but the final sum will be correct if its magnitude is less than 2^{b-1}. We show this by the following example. If we add a +6 to a +7, and then add a –7, we'll encounter an intermediate overflow error but our final sum will be correct as

```
+7 in two's complement  →          0 1 1 1
+6 in two's complement  →        + 0 1 1 0
–3 in two's complement  →          1 1 0 1    ← overflow error here
–7 in two's complement  →        + 1 0 0 1
+6 in two's complement  →          1 0 1 1 0  ← valid positive result
                                   ↑
overflow ignored, with no harm done
```

The magnitude of the sum of the three four-bit numbers was less than 2^{4-1} (<8), so our result was valid. If we add a +6 to a +7, and next add a –5, we'll encounter an intermediate overflow error, and our final sum will also be in error because its magnitude is not less than 8.

```
+7 in two's complement  →          0 1 1 1
+6 in two's complement  →        + 0 1 1 0
–3 in two's complement  →          1 1 0 1    ← overflow error here
–5 in two's complement  →        + 1 0 1 1
–8 in two's complement  →          1 1 0 0 0  ← invalid negative result
```

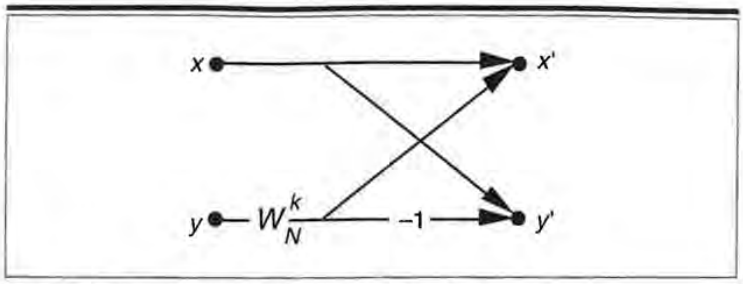


Figure 9-6 Single decimation-in-time FFT butterfly.

Another situation where overflow problems are conspicuous is during the calculation of the fast Fourier transform (FFT). It's difficult at first to imagine that multiplying complex numbers by sines and cosines can lead to excessive data word growth—particularly, because sines and cosines are never greater than unity. Well, we can show how FFT data word growth occurs by considering a decimation-in-time FFT butterfly from Figure 4-14(c) repeated here as Figure 9-6, and grinding through a little algebra. The expression for the x' output of this FFT butterfly, from Eq. (4-26), is

$$x' = x + W_N^k \cdot y \ . \tag{9-17}$$

Breaking up the butterfly's x and y inputs into their real and imaginary parts and remembering that $W_N^k = e^{-j2\pi k/N}$, we can express Eq. (9-17) as

$$x' = x_{\text{real}} + jx_{\text{imag}} + (e^{-j2\pi k/N}) \cdot (y_{\text{real}} + jy_{\text{imag}}) \ . \tag{9-18}$$

If we let α be the twiddle factor angle of $2\pi k/N$, and recall that $e^{-j\alpha} = \cos(\alpha) - j\sin(\alpha)$, we can simplify Eq. (9-18) as

$$x' = x_{\text{real}} + jx_{\text{imag}} + [\cos(\alpha) - j\sin(\alpha)] \cdot (y_{\text{real}} + jy_{\text{imag}})$$

$$= x_{\text{real}} + \cos(\alpha)y_{\text{real}} + \sin(\alpha)y_{\text{imag}} + j(x_{\text{imag}} + \cos(\alpha)y_{\text{imag}} - \sin(\alpha)y_{\text{real}}) \ . \tag{9-19}$$

If we look, for example, at just the real part of the x' output, x'_{real}, it comprises the three terms

$$x'_{\text{real}} = x_{\text{real}} + \cos(\alpha)y_{\text{real}} + \sin(\alpha)y_{\text{imag}} \tag{9-20}$$

If x_{real}, y_{real}, and y_{imag} are of unity value when they enter the butterfly and the twiddle factor angle $\alpha = 2\pi k/N$ happens to be $\pi/4 = 45°$, then, x'_{real} can be greater than 2 as

$$x'_{real} = 1 + \cos(45°) \cdot 1 + \sin(45°) \cdot 1$$

$$= 1 + 0.707 + 0.707 = 2.414 \quad . \tag{9-21}$$

So we see that the real part of a complex number can more than double in magnitude in a single stage of an FFT. The imaginary part of a complex number is equally likely to more than double in magnitude in a single FFT stage. Without mitigating this word growth problem, overflow errors could render an FFT algorithm useless.

OK, overflow problems are handled in one of two ways: by truncation or rounding—each inducing its own individual kind of quantization errors as we shall see.

9.3.3 Truncation

Truncation is the process where a data value is represented by the largest quantization level that is less than or equal to that data value. If we're quantizing to integral values, for example, the real value 1.2 would be quantized to 1. An example of truncation to integral values is shown in Figure 9-7(a), where all values of x in the range of $0 \le x < 1$ are set equal to 0, values of x in the range of $1 \le x < 2$ are set equal to 1, x values in the range of $2 \le x < 3$ are set equal to 2, and so on.

As we did with A/D converter quantization errors, we can call upon the concept of probability density functions to characterize the errors induced by truncation. The probability density function of truncation errors, in terms of the quantization level, is shown in Figure 9-7(b). In Figure 9-7(a) the quantization level q is 1, so, in this case, we can have truncation errors as great as –1. Drawing upon our results from Eqs. (D-11) and (D-12) in Appendix D, the mean and variance of our uniform truncation probability density function are expressed as

$$\mu_{truncation} = \frac{-q}{2} \tag{9-22}$$

and

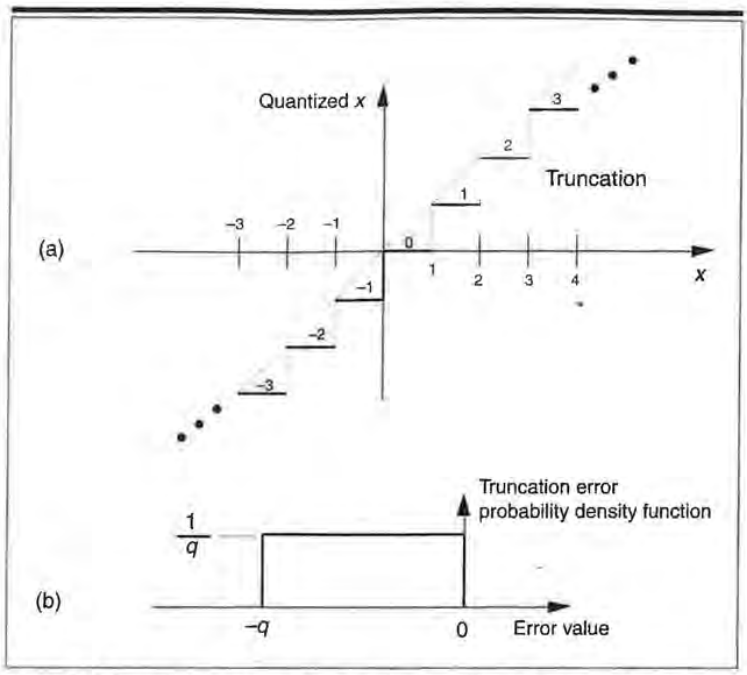


Figure 9-7 Truncation: (a) quantization nonlinearities; (b) error probability density function.

$$\sigma^2_{\text{truncation}} = \frac{q^2}{12} \quad . \tag{9-23}$$

In a sense, truncation error is the price we pay for the privilege of using integer binary arithmetic. One aspect of this is the error introduced when we use truncation to implement division by some integral power of 2. We often speak of a quick way of dividing a binary value by 2^T is to shift that binary word T bits to the right; that is, we're truncating the data value (not the data word) by lopping off the rightmost T bits after the shift. For example, let's say we have the value 31 represented by the five-bit binary number 11111_2, and we want to divide it by 16 through shifting the bits $T = 4$ places to the right and ignoring (truncating) those shifted bits. After the right shift and truncation, we'd have a binary quotient of $31/16 = 00001_2$. Well, we see the significance of the problem because our quick division gave us an answer of one instead of the

correct 31/16 = 1.9375. Our division-by-truncation error here is almost 50 percent of the correct answer. Had our original dividend been a 63 represented by the six-bit binary number 111111_2, dividing it by 16 through a four-bit shift would give us an answer of binary 000011_2, or decimal three. The correct answer, of course, is 63/16 = 3.9375. In this case the percentage error is 0.9375/3.9375, or about 23.8 percent. So, the larger the dividend, the lower the truncation error.

If we study these kinds of errors, we'll find that the resulting truncation error depends on three things: the number of value bits shifted and truncated, the values of the truncated bits (were those dropped bits ones or zeros), and the magnitude of the binary number left over after truncation. Although a complete analysis of these truncation errors is beyond the scope of this book, we can quantify the maximum error that can occur with our division-by-truncation scheme when using binary integer arithmetic. The worst case scenario is when all of the T bits to be truncated are ones. For binary integral numbers the value of T bits, all ones, to the right of the binary point is $1 - 2^{T}$. If the resulting quotient N is small, the significance of those truncated ones aggravates the error. We can normalize the maximum division error by comparing it to the correct quotient using a percentage. So the maximum percentage error of the binary quotient N after truncation for a T-bit right shift, when the truncated bits are all ones, is

$$\%\text{truncation error}_{max} = 100 \cdot \frac{\text{Correct quotient} - \text{Quotient after truncation}}{\text{Correct quotient}}$$

$$= 100 \cdot \frac{\text{Truncation error}}{\text{Correct quotient}} = 100 \cdot \frac{1 - 2^{-T}}{N + (1 - 2^{-T})} . \qquad (9\text{-}24)$$

Let's plug a few numbers into Eq. (9-24) to understand its significance. In the first example above, 31/16, where $T = 4$ and the resulting binary quotient was $N = 1$, the percentage error is

$$100 \cdot (1 - 0.0625)/(1 + 1 - 0.0625) = 100 \cdot (0.9375/1.9375) = 48.4\%.$$

Plotting Eq. (9-24) for three different shift values as functions of the quotient, N, after truncation results in Figure 9-8.

So, to minimize this type of division error, we need to keep the resulting binary quotient N, after the division, as large as possible. Remember, Eq. (9-24) was a worst case condition where all of the truncated bits

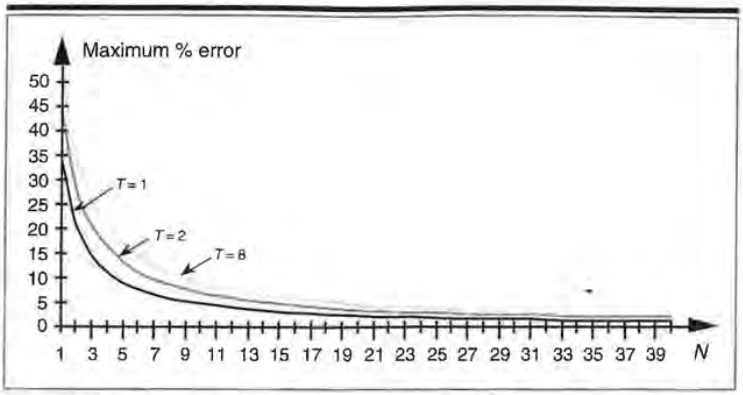


Figure 9-8 Maximum error when data shifting and truncation are used to implement division by 2^T. $T = 1$ is division by 2; $T = 2$ is division by 4; and $T = 8$ is division by 256.

were ones. On the other hand, if all of the truncated bits happened to be zeros, the error will be zero. In practice, the errors will be somewhere between these two extremes. A practical example of how division-by-truncation can cause serious numerical errors is given in reference [9].

9.3.4 Data Rounding

Rounding is an alternate process of dealing with overflow errors where a data value is represented by, or rounded off to, its nearest quantization level. If we're quantizing to integral values, the number 1.2 would be quantized to 1, and the number 1.6 would be quantized to 2. This is shown in Figure 9-9(a) where all values of x in the range of $-0.5 \le x < 0.5$ are set equal to 0, values of x in the range of $0.5 \le x < 1.5$ are set equal to 1, x values in the range of $1.5 \le x < 2.5$ are set equal to 2, etc. The probability density function of the error induced by rounding, in terms of the quantization level, is shown in Figure 9-9(b). In Figure 9-9(a), the quantization level q is 1, so, in this case, we can have truncation error magnitudes no greater than $q/2$, or $1/2$. Again, using our Eqs. (D-11) and (D-12) results from Appendix D, the mean and variance of our uniform roundoff probability density function are expressed as

$$\mu_{\text{roundoff}} = 0\,, \tag{9-25}$$

and

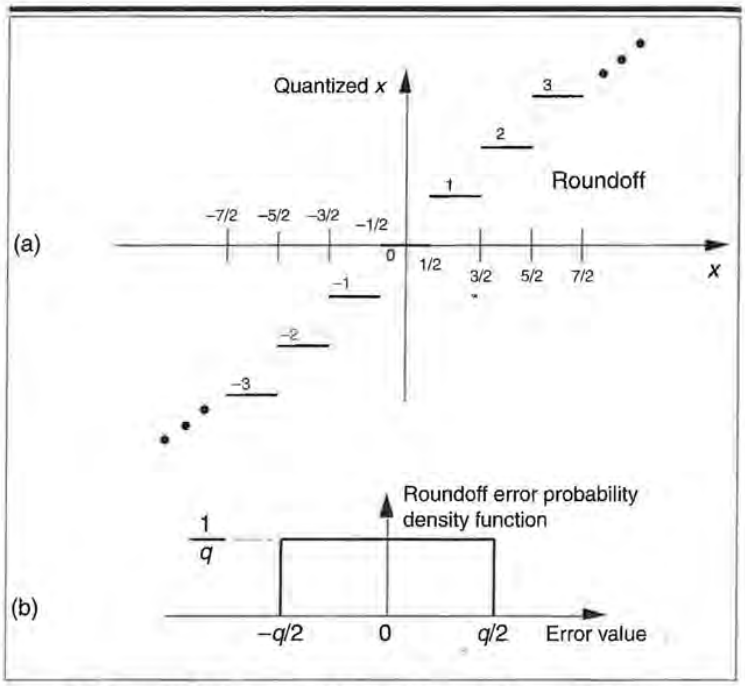


Figure 9-9 Roundoff: (a) quantization nonlinearities; (b) error probability density function.

$$\sigma^2_{\text{roundoff}} = \frac{q^2}{12} \quad . \tag{9-26}$$

Because the mean (average) and maximum error possibly induced by roundoff is less than that of truncation, rounding is generally preferred, in practice, to minimize overflow errors.

In digital signal processing, statistical analysis of quantization error effects is exceedingly complicated. Analytical results depend on the types of quantization errors, the magnitude of the data being represented, the numerical format used, and which of the many FFT or digital filter structures we happen to use. Be that as it may, digital signal processing experts have developed simplified error models whose analysis has proved useful. Although discussion of these analysis techniques and their results is beyond the scope of this introductory text, many references are available for the energetic reader[10–18]. (Reference [11] has an extensive reference list of its own on the topic of quantization error analysis.)

Again, the overflow problems using fixed-point binary formats—that we try to alleviate with truncation or roundoff—arise because so many digital signal processing algorithms comprise large numbers of additions or multiplications. This obstacle, particularly in hardware implementations of digital filters and the FFT, is avoided by hardware designers through the use of floating-point binary formats.

9.4 Floating-Point Binary Formats

Floating-point binary formats allow us to overcome most of the limitations of precision and dynamic range mandated by fixed-point binary formats, particularly in reducing the ill effects of overflow [19]. Floating-point formats segment a data word into two parts: a mantissa m and an exponent e. Using these parts, the value of a binary floating-point number n is evaluated as

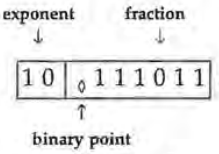
$$n = m \cdot 2^e, \tag{9-27}$$

that is, the number's value is the product of the mantissa and 2 raised to the power of the exponent. (*Mantissa* is a somewhat unfortunate choice of terms because it has a meaning here very different from that in the mathematics of logarithms—*mantissa* originally meant the decimal fraction of a logarithm.[†] However, due to its abundance in the literature we'll continue using the term *mantissa* here.) Of course, both the mantissa and the exponent in Eq. (9-27) can be either positive or negative numbers.

Let's assume that a b-bit floating-point number will use b_e bits for the fixed-point signed exponent and b_m bits for the fixed-point signed mantissa. The greater the number of b_e bits used, the larger the dynamic range of the number. The more bits used for b_m, the better the resolution, or precision, of the number. Early computer simulations conducted by the developers of b-bit floating-point formats indicated that the best trade-off occurred with $b_e \approx b/4$ and $b_m \approx 3b/4$. We'll see that, for typical 32-bit floating-point formats used today, $b_e \approx 8$ bits and $b_m \approx 24$ bits. To take advantage of a mantissa's full dynamic range, most implementations of floating-point numbers treat the mantissa as a fractional fixed-

[†] For example, the common logarithm (log to the base 10) of 256 is 2.4082. The 2 to the left of the decimal point is called the characteristic of the logarithm and the 4082 digits are called the mantissa. The 2 in 2.4082 does not mean that we multiply .4082 by 10^2. The 2 means that we take the antilog of .4082 to get 2.56 and multiply that by 10^2 to get 256.

point binary number, shift the mantissa bits to the right or left, so that its most significant bit is a one, and adjust the exponent accordingly. This convention is called *normalization*. When normalized, the mantissa bits are typically called the *fraction* of the floating-point number, instead of the mantissa. For example, the decimal value 3.6875_{10} can be represented by the fractional binary number 11.1011_2. If we use a two-bit exponent with a six-bit fraction floating-point word, we can just as well represent 11.1011_2 by shifting it to the right two places and setting the exponent to two as



$$11.1011_2 = \quad . \tag{9-28}$$

The floating-point word above can be evaluated to retrieve our decimal number again as

$$[0(1\cdot 2^{-1})+(1\cdot 2^{-2})+(1\cdot 2^{-3})+(0\cdot 2^{-4})+(1\cdot 2^{-5})+(1\cdot 2^{-6})]\cdot 2^2$$

$$=[0(1\cdot\frac{1}{2})+(1\cdot\frac{1}{4})+(1\cdot\frac{1}{8})+(0\cdot\frac{1}{16})+(1\cdot\frac{1}{32})+(1\cdot\frac{1}{64})]\cdot 2^2$$

$$=[0.5+0.25+0.125+0.0625+0.03125+0.015625]\cdot 2^2$$

$$=0.921875\cdot 4=3.6875\ . \tag{9-29}$$

After some experience using floating-point normalization, users soon realized that always having a one in the most significant bit of the fraction was wasteful. That redundant one was taking up a single bit position in all data words and serving no purpose. So practical implementations of floating-point formats discard that one, assume its existence, and increase the useful number of fraction bits by one. This is why the term *hidden bit* is used to describe some floating-point formats. While increasing the fraction's precision, this scheme uses less memory because the hidden bit is merely accounted for in the hardware arithmetic logic. Using a hidden bit, the fraction in Eq. (9-28)'s floating point number is shifted to the left one place and would now be

$$11.1011_2 = \boxed{10 \mid \lozenge 110110}. \qquad (9\text{-}30)$$

exponent ↓ fraction ↓

↑ binary point

Recall that the exponent and mantissa bits were fixed-point signed binary numbers, and we've discussed several formats for representing signed binary numbers, i.e., sign magnitude, two's complement, and offset binary. As it turns out, all three signed binary formats are used in industry-standard floating-point formats. The most common floating-point formats, all using 32-bit words, are listed in Table 9-3.

Table 9-3 Floating-Point Number Formats

IEEE Standard P754 Format

Bit	31	30	29	28	27	26	25	24	23	22	21	20	...	2	1	0
	S	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	...	2^{-21}	2^{-22}	2^{-23}
	Sign (s)	← Exponent (e) →								← Fraction (f) →						

IBM Format

Bit	31	30	29	28	27	26	25	24	23	22	21	20	...	2	1	0
	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	...	2^{-22}	2^{-23}	2^{-24}
	Sign (s)	← Exponent (e) →								← Fraction (f) →						

DEC (Digital Equipment Corp.) Format

Bit	31	30	29	28	27	26	25	24	23	22	21	20	...	2	1	0
	S	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-2}	2^{-3}	2^{-4}	...	2^{-22}	2^{-23}	2^{-24}
	Sign (s)	← Exponent (e) →								← Fraction (f) →						

MIL–STD 1750A Format

Bit	31	30	29	...	11	10	9	8	7	6	5	4	3	2	1	0
	2^0	2^{-1}	2^{-2}	...	2^{-20}	2^{-21}	2^{-22}	2^{-23}	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	← Fraction (f) →								← Exponent (e) →							

The IEEE P754 floating-point format is the most popular because so many manufacturers of floating-point integrated circuits comply with this standard [8, 20–22]. Its exponent e is offset binary (biased exponent), and its fraction is a sign-magnitude binary number with a hidden bit that's assumed to be 2^0. The decimal value of a normalized IEEE P754 floating-point number is evaluated as

$$\text{value}_{\text{IEEE}} = (-1)^s \cdot \underset{\text{hidden bit}}{\overset{\uparrow}{1_\Diamond}} f \cdot 2^{e-127}. \tag{9-31}$$

The IBM floating point format differs somewhat from the other floating-point formats because it uses a base of 16 rather than 2. Its exponent is offset binary, and its fraction is sign magnitude with no hidden bit. The decimal value of a normalized IBM floating-point number is evaluated as

$$\text{value}_{\text{IBM}} = (-1)^s \cdot 0_\Diamond f \cdot 16^{e-64}. \tag{9-32}$$

The DEC floating-point format uses an offset binary exponent, and its fraction is sign magnitude with a hidden bit that's assumed to be 2^{-1}. The decimal value of a normalized DEC floating-point number is evaluated as

$$\text{value}_{\text{DEC}} = (-1)^s \cdot \underset{\text{hidden bit}}{\overset{\uparrow}{0_\Diamond 1}} f \cdot 2^{e-128}. \tag{9-33}$$

MIL-STD 1750A is a United States Military Airborne floating-point standard. Its exponent e is a two's complement binary number residing in the least significant eight bits. MIL-STD 1750A's fraction is also a two's complement number (with no hidden bit), and that's why no sign bit is specifically indicated in Table 9-3. The decimal value of a MIL-STD 1750A floating-point number is evaluated as

$$\text{value}_{1750\text{A}} = f \cdot 2^e. \tag{9-34}$$

Notice how the floating-point formats in Table 9-3 all have word lengths of 32 bits—this was not accidental. Using 32-bit words makes these formats easier to handle using 8-, 16-, and 32-bit hardware processors. That fact not withstanding and given the advantages afforded by floating-point number formats, these formats do require a significant

amount of logical comparisons and branching to correctly perform arithmetic operations. Reference [23] provides useful flow charts showing what procedural steps must be taken when floating-point numbers are added and multiplied.

9.4.1 Floating-Point Dynamic Range

Attempting to determine the dynamic range of an arbitrary floating-point number format is a challenging exercise. We start by repeating the expression for a number system's dynamic range from Eq. (9-6) as

$$\text{dynamic range}_{dB} = 20 \cdot \log_{10}\left(\frac{\text{largest possible word value}}{\text{smallest possible word value}}\right). \quad (9\text{-}35)$$

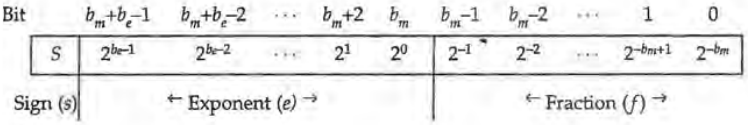
When we attempt to determine the largest and smallest possible values for a floating-point number format, we quickly see that they depend on such factors as

- the position of the binary point
- whether a hidden bit is used or not (If used, its position relative to the binary point is important.)
- the base value of the floating-point number format
- the signed binary format used for the exponent and the fraction (For example, recall from Table 9-2 that the binary two's complement format can represent larger negative numbers than the sign-magnitude format.)
- how unnormalized fractions are handled, if at all. (Unnormalized, also called gradual underflow, means a nonzero number that's less than the minimum normalized format but can still be represented when the exponent and hidden bit are both zero.)
- how exponents are handled when they're either all ones or all zeros. (For example, the IEEE P754 format treats a number having an all ones exponent and a nonzero fraction as an invalid number, whereas the DEC format handles a number having a sign = 1 and a zero exponent as a special instruction instead of a valid number.)

Trying to develop a dynamic range expression that accounts for all the possible combinations of the above factors is impractical. What we can do

is derive a rule of thumb expression for dynamic range that's often used in practice[8,22,24].

Let's assume the following for our derivation: the exponent is a b_e-bit offset binary number, the fraction is a normalized sign-magnitude number having a sign bit and b_m magnitude bits, and a hidden bit is used just left of the binary point. Our hypothetical floating-point word takes the following form:



First we'll determine what the largest value can be for our floating-point word. The largest fraction is a one in the hidden bit, and the remaining b_m fraction bits are all ones. This would make fraction $f = [1 + (1 - 2^{-b_m})]$. The first 1 in this expression is the hidden bit to the left of the binary point, and the value in parenthesis is all b_m bits equal to ones to the right of the binary point. The greatest positive value we can have for the b_e-bit offset binary exponent is $2^{(2^{b_e-1}-1)}$. So the largest value that can be represented with the floating-point number is the largest fraction raised to the largest positive exponent or

$$\text{largest possible word value} = [1+(1-2^{-b_m})]\cdot 2^{(2^{b_e-1}-1)} \quad . \tag{9-36}$$

The smallest value we can represent with our floating-point word is a one in the hidden bit times two raised to the exponent's most negative value, $2^{-(2^{b_e-1})}$, or

$$\text{smallest possible word value} = 1\cdot 2^{-(2^{b_e-1})} \quad . \tag{9-37}$$

Plugging Eqs. (9-36) and (9-37) into Eq. (9-35),

$$\text{dynamic range}_{\text{dB}} = 20\cdot\log_{10}\left(\frac{[1+(1-2^{-b_m})]\cdot 2^{(2^{b_e-1}-1)}}{1\cdot 2^{-(2^{b_e-1})}}\right) \quad . \tag{9-38}$$

Now here's where the thumb comes in—when b_m is large, say over seven, the 2^{-b_m} value approaches zero; that is, as b_m increases, the all ones fraction

$(1 - 2^{-b_m})$ value in the numerator approaches 1. Assuming this, Eq. (9-38) becomes

$$\text{dynamic range}_{\text{dB}} \approx 20 \cdot \log_{10}\left(\frac{[1+1]\cdot 2^{(2^{b_e-1}-1)}}{1\cdot 2^{-(2^{b_e-1})}}\right)$$

$$= 20 \cdot \log_{10}\left(\frac{2\cdot 2^{(2^{b_e-1}-1)}}{2^{-(2^{b_e-1})}}\right) = 20 \cdot \log_{10}\left(\frac{2^{(2^{b_e-1})}}{2^{-(2^{b_e-1})}}\right)$$

$$= 20 \cdot \log_{10}(2\cdot 2^{(2^{b_e-1})}) = 20 \cdot \log_{10}(2^{(2^{b_e})}) = 6.02 \cdot 2^{b_e}. \quad (9\text{-}39)$$

Using Eq. (9-39) we can estimate, for example, the dynamic range of the single-precision IEEE P754 standard floating-point format with its eight-bit exponent:

$$\text{dynamic range}_{\text{IEEE P754}} = 6.02 \cdot 2^8 = 1529 \text{ dB}. \quad (9\text{-}40)$$

Although we've introduced the major features of the most common floating-point formats, there are still more details to learn about floating-point numbers. For the interested reader the references given in this section provide a good place to start.

9.5 Block Floating-Point Binary Format

A marriage of fixed-point and floating-point binary formats is known as *block floating point*. This scheme is used, particularly in dedicated FFT integrated circuits, when large arrays, or blocks, of associated data are to be manipulated mathematically. Block floating-point schemes begin by examining all the words in a block of data, normalizing the largest-valued word's fraction, and establishing the correct exponent. This normalization takes advantage of the fraction's full dynamic range. Next, the fractions of the remaining data words are shifted appropriately, so that they can use the exponent of the largest word. In this way, all of the data words use the same exponent value to conserve hardware memory.

In FFT implementations, the arithmetic is performed treating the block normalized data values as fixed-point binary. However, when an addition causes an overflow condition, all of the data words are shifted one bit to the right (division by two), and the exponent is incremented by

one. As the reader may have guessed, block floating-point formats have increased dynamic range and avoid the overflow problems inherent in fixed-point formats but do not reach the performance level of true floating-point formats[8,25,26].

References

[1] Neugebauer, O. "The History of Ancient Astronomy," *Journal of Near Eastern Studies*, Vol. 4, 1945, pp. 12.

[2] Knuth, D. E. *The Art of Computer Programming: Seminumerical Methods*, Vol. 2, Section 4.1, Addison-Wesley Publishing, Reading, Massachusetts, 1981, pp. 179.

[3] Kester, W. "Peripheral Circuits Can Make or Break Sampling-ADC Systems," *EDN Magazine*, October 1, 1992.

[4] Grove, M. "Measuring Frequency Response and Effective Bits Using Digital Signal Processing Techniques," *Hewlett-Packard Journal*, February 1992.

[5] Tektronix. "Effective Bits Testing Evaluates Dynamic Range Performance of Digitizing Instruments," *Tektronix Application Note*, No. 45W-7527, December 1989.

[6] Ushani, R. "Subranging ADCs Operate at High Speed with High Resolution," *EDN Magazine*, April 11, 1991.

[7] Demler, M. "Time-domain Techniques Enhance Testing of High-speed ADCs," *EDN Magazine*, March 30, 1992.

[8] Hilton, H. "A 10-MHz Analog-to-Digital Converter with 110-dB Linearity," *Hewlett-Packard Journal*, October 1993.

[9] Lyons, R. G. "Providing Software Flexibility for Optical Processor Noise Analysis," *Computer Design*, July 1978, pp. 95.

[10] Knuth, D. E. *The Art of Computer Programming: Seminumerical Methods*, Vol. 2, Section 4.2, Addison-Wesley Publishing, Reading, Massachusetts, 1981, pp. 198.

[11] Rabiner, L. R., and Gold, B. *Theory and Application of Digital Signal Processing*, Chapter 5, Prentice-Hall, Englewood Cliffs, New Jersey, 1975, pp. 353.

[12] Jackson, L. B. "An Analysis of Limit Cycles Due to Multiplicative Rounding in Recursive Digital Filters," *Proc. 7th Allerton Conf. Circuit System Theory*, 1969, pp. 69–78.

[13] Kan, E. P. F, and Aggarwal, J. K. "Error Analysis of Digital Filters Employing Floating Point Arithmetic," *IEEE Trans. Circuit Theory*, Vol. CT-18, November 1971, pp. 678–86.

[14] Crochiere, R. E. "Digital Ladder Structures and Coefficient Sensitivity," *IEEE Trans. Audio Electroacoustics*, Vol. AU-20, October 1972, pp. 240–46.

[15] Jackson, L. B. "On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters," *Bell System Technical Journal*, Vol. 49, February 1970, pp. 159–84.

[16] Roberts, R. A., and Mullis, C. T. *Digital Signal Processing*, Addison-Wesley Publishing, Reading, Massachusetts, 1987, pp. 277.

[17] Jackson, L. B. "Roundoff Noise Analysis for Fixed-Point Digital Filters Realized in Cascade or Parallel Form," *IEEE Trans. Audio Electroacoustics*, Vol. AU-18, June 1970, pp. 107–22.

[18] Oppenheim, A. V., and Schafer, R. W. *Discrete-Time Signal Processing*, Sections 6.8 and 9.8, Prentice-Hall, Englewood Cliffs, New Jersey, 1989, pp. 335.

[19] Larimer, J., and D. Chen. "Fixed or Floating? A Pointed Question in DSPs," *EDN Magazine*, August 3, 1995.

[20] Ashton, C. "Floating Point Math Handles Iterative and Recursive Algorithms," *EDN Magazine*, January 9, 1986.

[21] Windsor, B., and Wilson, J. "Arithmetic Duo Excels in Computing Floating Point Products," *Electronic Design*, May 17, 1984.

[22] Windsor, W. A. "IEEE Floating Point Chips Implement DSP Architectures," *Computer Design*, January 1985.

[23] Texas Instruments Inc., *Digital Signal Processing Applications with the TMS320 Family: Theory, Algorithms, and Implementations*, SPRA012A, Texas Instruments, Dallas, TX, 1986.

[24] Strauss, W. I. "Integer or Floating Point? Making the Choice," *Computer Design Magazine*, April 1, 1990, pp. 85.

[25] Oppenheim and Weinstein. "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform," *Proc. IEEE*, August 1972, pp. 957–76.

[26] Woods, R. E. "Transform-based Processing: "How Much Precision Is Needed," *ESD: The Electronic System Design Magazine*, February 1987.

CHAPTER TEN

Digital Signal Processing Tricks

As we study the literature of digital signal processing, we'll encounter some creative techniques that *professionals* use to make their algorithms more efficient. These techniques are straightforward examples of the philosophy "don't work hard, work smart" and studying them will give us a deeper understanding of the underlying mathematical subtleties of digital signal processing. In this chapter, we present a collection of these clever tricks of the trade and explore several of them in detail, because doing so reinforces some of the lessons we've learned in previous chapters.

10.1 Frequency Translation without Multiplication

Frequency translation is often called for in digital signal processing algorithms. A filtering scheme called *transmultiplexing* (using the FFT to efficiently implement a bank of bandpass filters) requires spectral shifting by half the sample rate, or $f_s/2$[1]. Inverting bandpass sampled spectra and converting low-pass FIR filters to highpass filters both call for frequency translation by half the sample rate. Conventional quadrature bandpass sampling uses spectral translation by one quarter of the sample rate, or $f_s/4$, to reduce unnecessary computations[2,3]. There are a couple of tricks used to perform discrete frequency translation, or *mixing*, by $f_s/2$ and $f_s/4$ without actually having to perform any multiplications. Let's take a look at these mixing schemes in detail.

First, we'll consider a slick technique for frequency translating an input sequence by $f_s/2$ by merely multiplying that sequence by $(-1)^n$, or $(-1)^0$, $(-1)^1$, $(-1)^2$, $(-1)^3$, etc. Better yet, this requires only changing the sign of every other input sample value because $(-1)^n = 1, -1, 1, -1$, etc. This process may seem a bit mysterious at first, but it can be explained in a straightforward way if we review Figure 10-1. The figure shows us that

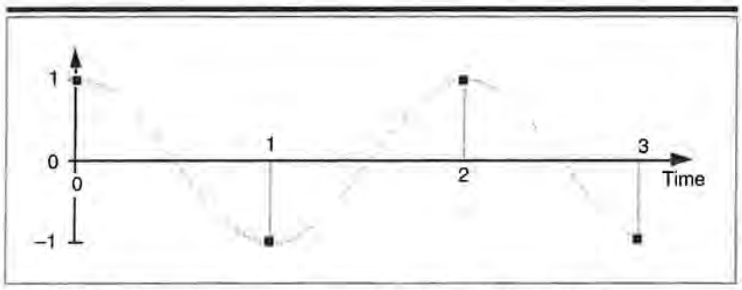


Figure 10-1 Mixing sequence comprising $(-1)^n$: 1, -1, 1, -1, etc.

multiplying a time-domain signal sequence by the $(-1)^n$ mixing sequence is equivalent to multiplying the signal sequence by a sampled cosinusoid where the mixing sequence values are shown as the dots in Figure 10-1. Because the mixing sequence's cosine repeats every two sample values, its frequency is $f_s/2$. Let's look at this situation in detail, not only to understand mixing sequences, but to illustrate the DFT equation's analysis capabilities, to reiterate the nature of complex signals, and to reconfirm the important equivalence of shifting in the time domain and phase shifting in the frequency domain.

We can verify the $(-1)^n$ mixing sequence's frequency translation of $f_s/2$ by taking the DFT of the mixing sequence expressed as $F_{1,-1,1,-1\ldots}(m)$ where

$$F_{1,-1,1,-1,\ldots}(m) = \sum_{n=0}^{N-1}(1,-1,1,-1,\ldots)e^{-j2\pi nm/N} \quad . \tag{10-1}$$

Using a 4-point DFT, we expand the sum in Eq. (10-1), with $N = 4$, to

$$F_{1,-1,1,-1}(m) = e^{-j2\pi 0m/4} - e^{-j2\pi 1m/4} + e^{-j2\pi 2m/4} - e^{-j2\pi 3m/4} \, . \tag{10-2}$$

Notice that the mixing sequence is embedded in the signs of the terms of Eq. (10-2) that we evaluate from $m = 0$ to $m = 3$ to get

$$m = 0: \quad F_{1,-1,1,-1}(0) = e^{-j0} - e^{-j0} + e^{-j0} - e^{-j0} = 1 - 1 + 1 - 1 = 0 \, , \tag{10-3}$$

$$m = 1: \quad F_{1,-1,1,-1}(1) = e^{-j0} - e^{-j2\pi/4} + e^{-j4\pi/4} - e^{-j6\pi/4} = 1 + j1 - 1 - j1 = 0 \, , \tag{10-4}$$

$$m = 2: \quad F_{1,-1,1,-1}(2) = e^{-j0} - e^{-j4\pi/4} + e^{-j8\pi/4} - e^{-j12\pi/4} = 1 + 1 + 1 + 1 = 4 \, , \tag{10-5}$$

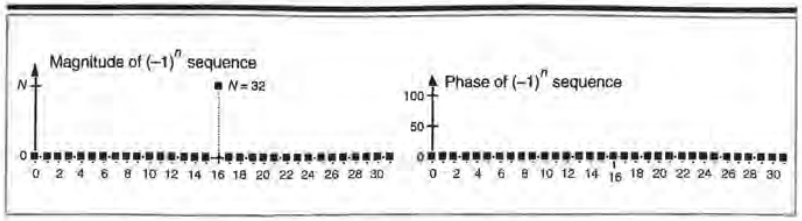


Figure 10-2 Frequency-domain magnitude and phase of an N-point $(-1)^n$ sequence.

and

$$m = 3: \quad F_{1,-1,1,-1}(3) = e^{-j0} - e^{-j6\pi/4} + e^{-j12\pi/4} - e^{-j18\pi/4} = 1 - j1 - 1 + j1 = 0 \,. \quad (10\text{-}6)$$

See how the 1, –1, 1, –1 mixing sequence has a nonzero frequency component only when $m = 2$ corresponding to a frequency of $mf_s/N = 2f_s/4 = f_s/2$. So, in the frequency domain the four-sample 1, –1, 1, –1 mixing sequence is an $f_s/2$ sinusoid with a magnitude of 4 and a phase angle of 0°. Had our mixing sequence contained eight separate values, the results of an 8-point DFT would have been all zeros with the exception of the $m = 4$ frequency sample with its magnitude of 8 and phase angle of 0°. In fact, the DFT of N $(1)^n$ has a magnitude of N at a frequency $f_s/2$. When $N = 32$, for example, the magnitude and phase of a 32-point $(-1)^n$ sequence is shown in Figure 10-2.

Let's demonstrate this $(-1)^n$ mixing with an example. Consider the 32 discrete samples of the sum of 3 sinusoids comprising a real time-domain sequence $x(n)$ shown in Figure 10-3(a) where

$$x(n) = \cos\left(\frac{2\pi 10n}{32}\right) + 0.5 \cdot \cos\left(\frac{2\pi 11n}{32} - \frac{\pi}{4}\right) + 0.25 \cdot \cos\left(\frac{2\pi 12n}{32} - \frac{3\pi}{8}\right) \,. \quad (10\text{-}7)$$

The frequencies of the three sinusoids are 10/32 Hz, 11/32 Hz, and 12/32 Hz, and they each have an integral number of cycles (10, 11, and 12) over 32 samples of $x(n)$. To show the magnitude and phase shifting effect of using the 1, –1, 1, –1 mixing sequence, we've added arbitrary phase shifts of $-\pi/4$ (–45°) and $-3\pi/8$ (–67.5°) to the second and third tones. Using a 32-point DFT results in the magnitude and phase of $X(m)$ shown in Figure 10-3(b).

Let's notice something about Figure 10-3(b) before we proceed. The magnitude of the $m = 10$ frequency sample $|X(10)|$ is 16. Remember why this is so? Recall Eq. (3-17) from Chapter 3 where we learned that, if a real input signal contains a sinewave component of peak amplitude A_o with

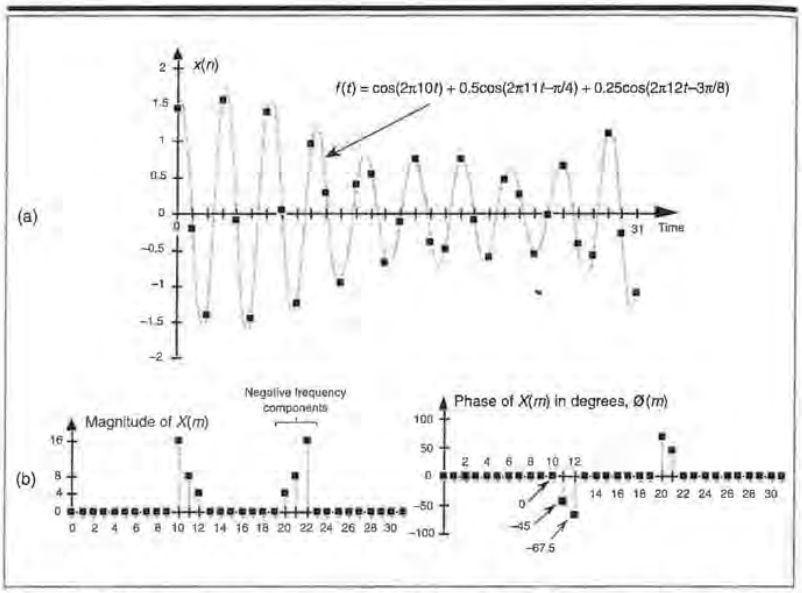


Figure 10-3 Discrete signal sequence $x(n)$: (a) time-domain representation of $x(n)$; (b) frequency-domain magnitude and phase of $X(m)$.

an integral number of cycles over N input samples, the output magnitude of the DFT for that particular sinewave is $A_0N/2$. In our case, then, $|X(10)| = 1 \cdot 32/2 = 16$, $|X(11)| = 8$, and $|X(12)| = 4$.

If we multiply $x(n)$, sample by sample, by the $(-1)^n$ mixing sequence, our new time-domain sequence $x_{1,-1}(n)$ is shown in Figure 10–4(a), and the DFT of the frequency translated $x_{1,-1}$ is provided in Figure 10-4(b). (Remember now, we didn't really perform any explicit multiplications—the whole idea here is to avoid multiplications—we merely changed the sign of alternating $x(n)$ samples to get $x_{1,-1}(n)$.) Notice that the magnitude and phase of $X_{1,-1}(m)$ are the magnitude and phase of $X(m)$ shifted by $f_s/2$, or 16 sample shifts, from Figure 10-3(b) to Figure 10-4(b). The negative frequency components of $X(m)$ are shifted downward in frequency, and the positive frequency components of $X(m)$ are shifted upward in frequency resulting in $X_{1,-1}(m)$. It's a good idea for the reader to be energetic and prove that the magnitude of $X_{1,-1}(m)$ is the convolution of the $(-1)^n$ sequence's spectral magnitude in Figure 10-2 and the magnitude of $X(m)$ in Figure 10-3(b). Another way to look at the $X_{1,-1}(m)$ magnitudes in Figure 10-4(b) is to see that multiplication by the $(-1)^n$ mixing sequence flips the positive frequency band of $X(m)$ from zero to $+f_s/2$ Hz about $f_s/4$ Hz and flips the negative frequency band of $X(m)$ from $-f_s/2$ to zero Hz, about

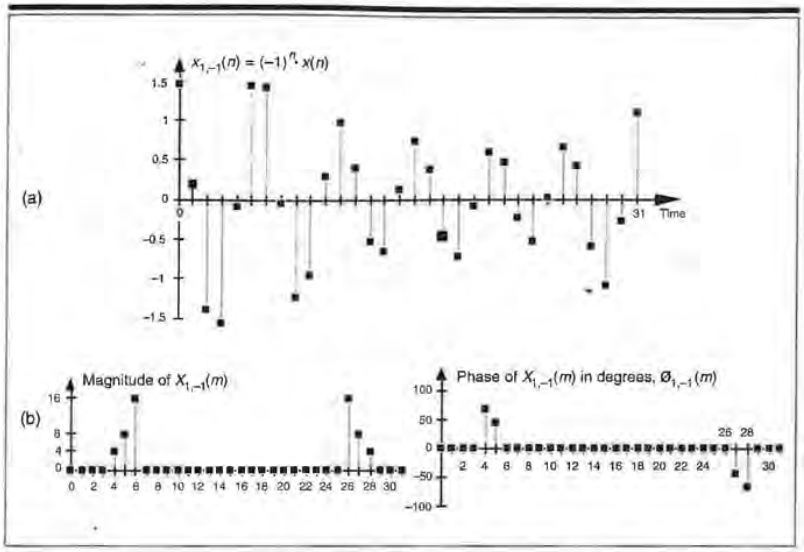


Figure 10-4 Frequency translation by $f_s/2$: (a) mixed sequence $x_{1,-1}(n) = (-1)^n \cdot x(n)$; (b) magnitude and phase of frequency-translated $X_{1,-1}(m)$.

$-f_s/4$ Hz. This process can be used to invert spectra when bandpass sampling is used, as described in Section 2.4.

Another useful mixing sequence is 1,–1,–1,1, etc. It's used to translate spectra by $f_s/4$ in quadrature sampling schemes and is illustrated in Figure 10-5(a). In digital quadrature mixing, we can multiply an input data sequence $x(n)$ by the cosine mixing sequence 1,–1,–1,1 to get the in-phase component of $x(n)$—what we'll call $i(n)$. To get the quadrature-phase product sequence $q(n)$, we multiply the original input data sequence by the sine mixing sequence of 1,1,–1,–1. This sine mixing sequence is out of phase with the cosine mixing sequence by 90°, as shown in Figure 10-5(b).

If we multiply the 1,–1,–1,1 cosine mixing sequence and an input sequence $x(n)$, we'll find that the $i(n)$ product has a DFT magnitude that's related to the input's DFT magnitude $X(m)$ by

$$|I(m)|_{1,-1,-1,1} = \frac{|X(m)|}{\sqrt{2}} . \tag{10-8}$$

To see why, let's explore the cosine mixing sequence 1,–1,–1,1 in the frequency domain. We know that the DFT of the cosine mixing sequence, represented by $F_{1,-1,-1,1}(m)$, is expressed by

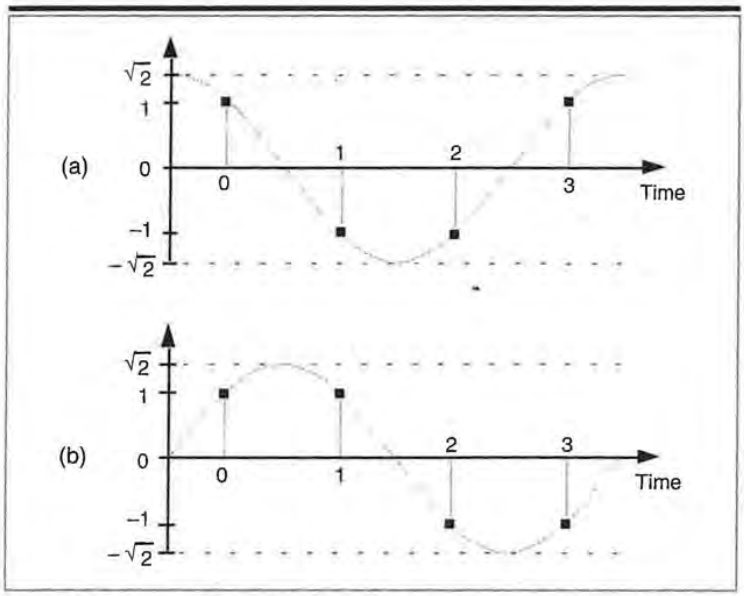


Figure 10-5 Quadrature mixing sequences for downconversion by $f_s/4$: (a) cosine mixing sequence using 1,-1,-1,1,...; (b) sine mixing sequence using 1,1,-1,-1,....

$$F_{1,-1,-1,1}(m) = \sum_{n=0}^{N-1}(1,-1,-1,1,\ \ldots)e^{-j2\pi nm/N} \quad . \tag{10-9}$$

Because a 4-point DFT is sufficient to evaluate Eq. (10-9), with $N = 4$,

$$F_{1,-1,-1,1}(m) = e^{-j2\pi 0m/4} - e^{-j2\pi 1m/4} - e^{-j2\pi 2m/4} + e^{-j2\pi 3m/4} \quad . \tag{10-10}$$

Notice, again, that the cosine mixing sequence is embedded in the signs of the terms of Eq. (10-10). Evaluating Eq. (10-10) for $m = 1$, corresponding to a frequency of $1 \cdot f_s/N$, or $f_s/4$, we find

$$m = 1: \quad F_{1,-1,-1,1}(1) = e^{-j0} - e^{-j\pi/2} - e^{-j\pi} + e^{-j3\pi/2}$$

$$= 1 + j1 + 1 + j1 = 2 + j2 = \frac{4}{\sqrt{2}}\angle 45°. \tag{10-11}$$

So, in the frequency domain, the cosine mixing sequence has an $f_s/4$ magnitude of $4/\sqrt{2}$ at a phase angle of 45°. Similarly, evaluating Eq. (10-10) for $m = 3$, corresponding to a frequency of $-f_s/4$, we find

$$m = 3: \quad F_{1,-1,-1,1}(3) = e^{-j0} - e^{-j3\pi/2} - e^{-j3\pi} + e^{-j9\pi/2}$$

$$= 1 - j1 + 1 - j1 = 2 - j2 = \frac{4}{\sqrt{2}} \angle -45°. \tag{10-12}$$

The energetic reader should evaluate Eq. (10-10) for $m = 0$ and $m = 2$, to confirm that the 1,–1,–1,1 sequence's DFT coefficients are zero for the frequencies of 0 and $f_s/2$.

Because the 4-point DFT magnitude of an all positive ones mixing sequence (1, 1, 1, 1) is 4†, we see that the frequency-domain scale factor for the 1,–1,–1,1 cosine mixing sequence is expressed as

$$I(m)_{1,-1,-1,1} \text{ scale factor} = \frac{\text{cosine sequence DFT magnitude}}{\text{all ones sequence DFT magnitude}}$$

$$= \frac{4/\sqrt{2}}{4} = \frac{1}{\sqrt{2}}, \tag{10-13}$$

which confirms the relationship in Eq. (10-8) and Figure 10-5(a). Likewise, the DFT scale factor for the quadrature-phase mixing sequence (1,1,–1,–1) is

$$Q(m)_{1,1,-1,-1} \text{ scale factor} = \frac{1}{\sqrt{2}},$$

thus

$$|Q(m)|_{1,1,-1,-1} = \frac{|X(m)|}{\sqrt{2}}. \tag{10-14}$$

So what this all means is that an input signal's spectral magnitude, after frequency translation, will be reduced by a factor of $\sqrt{2}$. There's really no harm done, however—when the in-phase and quadrature-phase components are combined to find the magnitude of a complex frequency

† We can show this by letting $K = N = 4$ in Eq. (3-44) for a four-sample all ones sequence in Chapter 3.

sample $X(m)$, the $\sqrt{2}$ scale factor is eliminated, and there's no overall magnitude loss because

$$|\text{scale factor}| = \sqrt{(I(m)\text{ scale factor})^2 + (Q(m)\text{ scale factor})^2}$$

$$= \sqrt{(1/\sqrt{2})^2 + (1/\sqrt{2})^2} = \sqrt{(1/2)+(1/2)} = 1. \qquad (10\text{-}15)$$

We can demonstrate this quadrature mixing process using the $x(n)$ sequence from Eq. (10-7) whose spectrum is shown in Figure 10-3(b). If we multiply that 32-sample $x(n)$ by 32 samples of the quadrature mixing sequences 1,–1,–1,1 and 1,1,–1,–1, whose DFT magnitudes are shown in Figure 10-6(a) and (b), the new quadrature sequences will have the

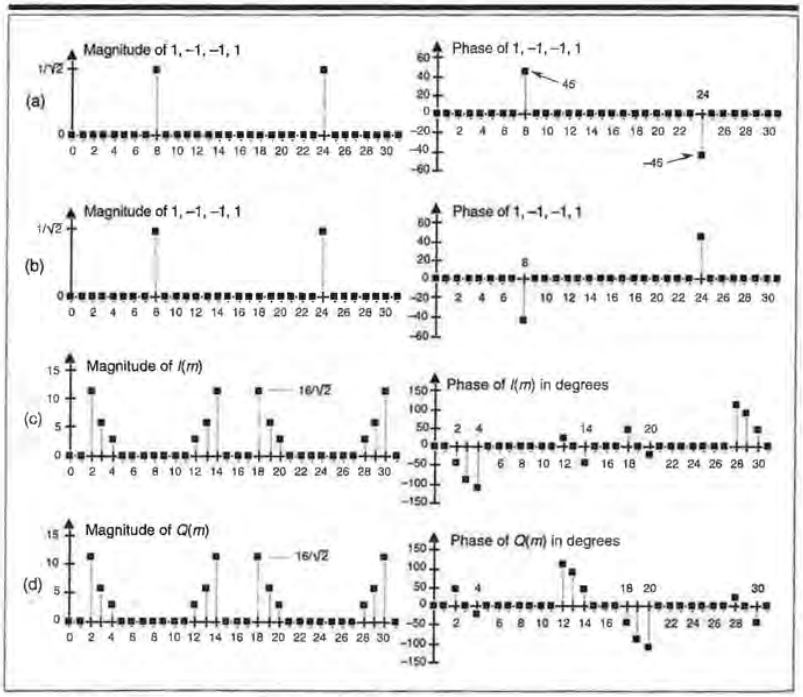


Figure 10-6 Frequency translation by $f_s/4$: (a) normalized magnitude and phase of cosine 1,–1,–1,1 sequence; (b) normalized magnitude and phase of sine 1,1,–1,–1 sequence; (c) magnitude and phase of frequency-translated, in-phase $I(m)$; (d) magnitude and phase of frequency-translated, quadrature-phase $Q(m)$.

frequency-translated $I(m)$ and $Q(m)$ spectra shown in Figure 10-6(c) and (d). (Remember now, we don't actually perform any multiplications; we merely change the sign of appropriate $x(n)$ samples to get the $i(n)$ and $q(n)$ sequences.)

There's a lot to learn from Figure 10-6. First, the positive frequency components of $X(m)$ from Figure 10-3(b) are indeed shifted downward by $f_s/4$ in Figure 10-6(c). Because our total discrete frequency span (f_s Hz) is divided into 32 samples, $f_s/4$ is equal to eight. So, for example, the $X(10)$ component in Figure 10-3(b) corresponds to the $I(10-8) = I(2)$ component in Figure 10-6(c). Likewise, $X(11)$ corresponds to $I(11-8) = I(3)$, and so on. Notice, however, that the positive and negative components of $X(m)$ have each been repeated twice in the frequency span in Figure 10-6(c). This effect is inherent in the process of mixing any discrete time-domain signal with a sinusoid of frequency $f_s/4$. Verifying this gives us a good opportunity to pull convolution out of our toolbox and use it to see why the $I(m)$ spectral replication period is reduced by a factor of 2 from that of $X(m)$. Recall, from the convolution theorem, that the DFT of the time-domain product of $x(n)$ and the 1,–1,–1,1 mixing sequence $I(m)$ is the convolution of their individual DFTs, or $I(m)$ is equal to the convolution of $X(m)$ and the 1,–1,–1,1 mixing sequence's magnitude spectrum in Figure 10-6(a). If, for convenience, we denote the 1,–1,–1,1 cosine mixing sequence's magnitude spectrum as $S_c(m)$, we can say that $I(m) = X(m)*S_c(m)$ where the "$*$" symbol denotes convolution.

Let's look at that particular convolution to make sure we get the $I(m)$ spectrum in Figure 10-6(c). Redrawing $X(m)$ from Figure 10-3(b) to show its positive and negative frequency replications gives us Figure 10-7(a). We also redraw $S_c(m)$ from Figure 10-6(a) showing its positive and negative frequency components in Figure 10-7(b). Before we perform the convolution's shift and multiply, we realize that we don't have to flip $S_c(m)$ about the zero frequency axis because, due to its symmetry, that would have no effect. So now our convolution comprises the shifting of $S_c(m)$ in Figure 10-7(b), relative to the stationary $X(m)$, and taking the product of that shifted sequence and the $X(m)$ spectrum in Figure 10-7(a) to arrive at $I(m)$. No shift of $S_c(m)$ corresponds to the $m = 0$ sample of $I(m)$. The sums of the products for this zero shift is zero, so $I(0) = 0$. If we shift $S_c(m)$ to the right by two samples, we'd have an overlap of $S_c(8)$ and $X(10)$, and that product gives us $I(2)$. One more $S_c(m)$ shift to the right results in an overlap of $S_c(8)$ and $X(11)$, and that product gives us $I(3)$, and so on. So shifting $S_c(m)$ to the right and summing the products of $S_c(m)$ and $X(m)$ results in $I(1)$ to $I(14)$. If we return $S_c(m)$ to its unshifted position in Figure 10-7(b), and then shift it to the left two samples in the negative frequency

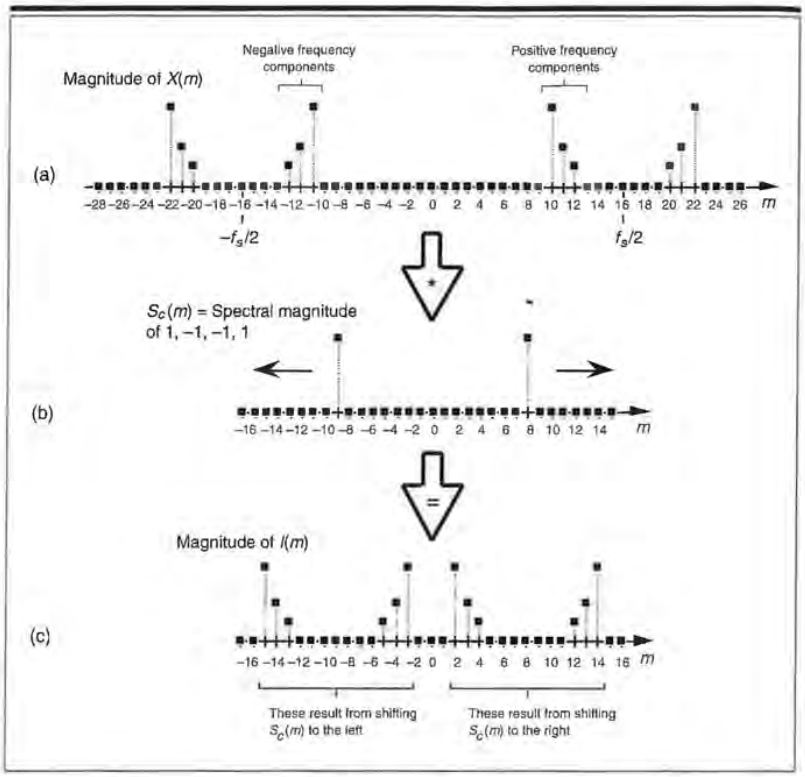


Figure 10-7 Frequency-domain convolution resulting in $I(m)$: (a) magnitude of $X(m)$; (b) spectral magnitude of the cosine's 1,−1,−1,1 time-domain sequence, $S_c(m)$; this is the sequence we'll shift to the left and right to perform the convolution; (c) convolution result: the magnitude of frequency-translated, in-phase $I(m)$.

direction, we'd have an overlap of $S_c(-8)$ and $X(-10)$, and that product gives us $I(-2)$. One more $S_c(m)$ shift to the left results in an overlap of $S_c(-8)$ and $X(-11)$, and that product gives us $I(-3)$, and so on. Continuing to shift $S_c(m)$ to the left determines the remaining negative frequency components $I(-4)$ to $I(-14)$. Figure 10-7(c) shows which $I(m)$ samples resulted from the left and right shifting of $S_c(m)$. By using the convolution theorem, we can see, now, that the magnitudes in Figure 10-7(c) and Figure 10-6(c) really are the spectral magnitudes of the in-phase component $I(m)$ with its reduced spectral replication period.

The upshot of all of this is that we can change the signs of appropriate $x(n)$ samples to shift $x(n)$'s spectrum by one quarter of the sample rate

without having to perform any explicit multiplications. Moreover, if we change the signs of appropriate $x(n)$ samples in accordance with the mixing sequences in Figure 10-5, we can get the in-phase $i(n)$ and quadrature-phase $q(n)$ components of the original $x(n)$. One important effect of this digital mixing by $f_s/4$ is that the spectral replication periods of $I(m)$ and $Q(m)$ are half the replication period of the original $X(m)$.† So we must be aware of the potential frequency aliasing problems that may occur with this frequency-translation method if the signal bandwidth is too wide relative to the sample rate, as discussed in Section 7.3.

Before we leave this particular frequency-translation scheme, let's review two more issues, magnitude and phase. Notice that the untranslated $X(10)$ magnitude is equal to 16 in Figure 10-3(b), and that the translated $I(2)$ and $Q(2)$ magnitudes are $16/\sqrt{2} = 11.314$ in Figure 10-6. This validates Eq. (10-8) and Eq. (10-14). If we use those quadrature components $I(2)$ and $Q(2)$ to determine the magnitude of the corresponding frequency-translated, complex spectral component from the square root of the sum of the squares relationship, we'd find that the magnitude of the peak spectral component is

$$\text{peak component magnitude} = \sqrt{(16/\sqrt{2})^2 + (16/\sqrt{2})^2} = \sqrt{256} = 16, \quad (10\text{-}16)$$

verifying Eq. (10-15). So combining the quadrature components $I(m)$ and $Q(m)$ does not result in any loss in spectral amplitude due to the frequency translation. Finally, in performing the above convolution process, the phase angle samples of $X(m)$ in Figure 10-3(b) and the phase samples of the 1,–1,–1,1 sequence in Figure 10-6(a) add algebraically. So the resultant $I(m)$ phase angle samples in Figure 10-6(c) result from either adding or subtracting 45° from the phase samples of $X(m)$ in Figure 10-3(b).

Another easily implemented mixing sequence used for $f_s/4$ frequency translations to obtain $I(m)$ is the 1, 0, –1, 0, etc., cosine sequence shown in Figure 10-8(a). This mixing sequence's quadrature companion 0, 1, 0, –1, Figure 10-8(b), is used to produce $Q(m)$. To determine the spectra of these sequences, let's, again, use a 4-point DFT to state that

$$F_{1,0,-1,0}(m) = \sum_{n=0}^{N-1} (1, 0, -1, 0, \ldots) e^{-j2\pi nm/N} . \quad (10\text{-}17)$$

† Recall that we saw this reduction in spectral replication period in the quadrature sampling results shown in Figures 7-2(g) and 7-3(d).

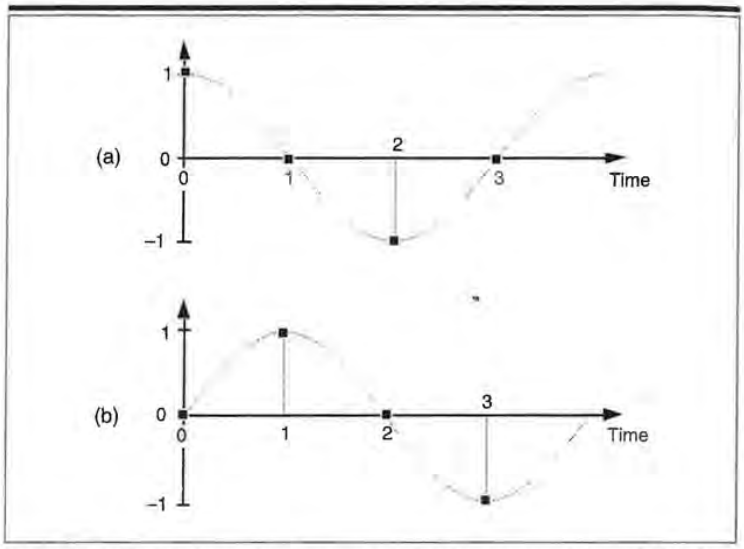


Figure 10-8 Quadrature mixing sequences for downconversion by $f_s/4$: (a) cosine mixing sequence using 1, 0, –1, 0, . . .; (b) sine mixing sequence using 0, 1, 0, –1, . . .

When $N = 4$,

$$F_{1,0,-1,0}(m) = e^{-j2\pi 0m/4} - e^{-j2\pi 2m/4} \quad (10\text{-}18)$$

Again, the cosine mixing sequence is embedded in the signs of the terms of Eq. (10-18), and there are only two terms for our 4-point DFT. We evaluate Eq. (10-18) for $m = 1$, corresponding to a frequency of $f_s/4$, to find that

$$F_{1,0,-1,0}(1) = e^{-j0} - e^{-j\pi} = 1+1 = 2\angle 0° \quad . \quad (10\text{-}19)$$

Evaluating Eq. (10-18) for $m = 3$, corresponding to a frequency of $-f_s/4$, shows that

$$F_{1,0,-1,0}(3) = e^{-j0} - e^{-j3\pi} = 1+1 = 2\angle 0° \quad . \quad (10\text{-}20)$$

Using the approach in Eq. (10-13), we can show that the scaling factor for the 1, 0, –1, 0 cosine mixing sequence is given as

$$I(m)_{1,0,-1,0} \text{ scale factor} = \frac{2}{4} = \frac{1}{2} .$$

So

$$|I(m)|_{1,0,-1,0} = \frac{|X(m)|}{2} . \quad (10\text{-}21)$$

Likewise, if we went through the same exercise as above, we'd find that the scaling factor for the 0, 1, 0, –1 sine mixing sequence is given by

$$Q(m)_{0,1,0,-1} \text{ scale factor} = \frac{1}{2} .$$

So

$$|Q(m)|_{0,1,0,-1} = \frac{|X(m)|}{2} . \quad (10\text{-}22)$$

So these mixing sequences induce a loss in the frequency-translated signal amplitude by a factor of 2.

By way of example, let's show this scale factor loss again by frequency translating the $x(n)$ sequence from Eq. (10-7), whose spectrum is shown in Figure 10-3(b). If we multiply that 32-sample $x(n)$ by 32 samples of the quadrature mixing sequences 1, 0, –1, 0 and 0, 1, 0, –1, whose DFT magnitudes are shown in Figure 10-9(a) and (b), the resulting quadrature sequences will have the frequency-translated $I(m)$ and $Q(m)$ spectra shown in Figure 10-9(c) and (d).

Notice that the untranslated $X(10)$ magnitude is equal to 16 in Figure 10-3(b) and that the translated $I(2)$ and $Q(2)$ magnitudes are $16/2 = 8$ in Figure 10-6. This validates Eq. (10-21) and Eq. (10-22). If we use those quadrature components $I(2)$ and $Q(2)$ to determine the magnitude of the corresponding frequency-translated, complex spectral component from the square root of the sum of the squares relationship, we'd find that the magnitude of the peak spectral component is

$$\text{peak component magnitude} = \sqrt{(16/2)^2 + (16/2)^2} = \sqrt{(16)^2/2} = \frac{16}{\sqrt{2}} . \quad (10\text{-}23)$$

When the in-phase and quadrature-phase components are combined to get the magnitude of a complex value, a resultant $\sqrt{2}$ scale factor, for the 1, 0, –1, 0 and 0, 1, 0, –1 sequences, is not eliminated. An overall 3 dB loss remains because we eliminated some of the signal power when we multiplied half of the data samples by zero.

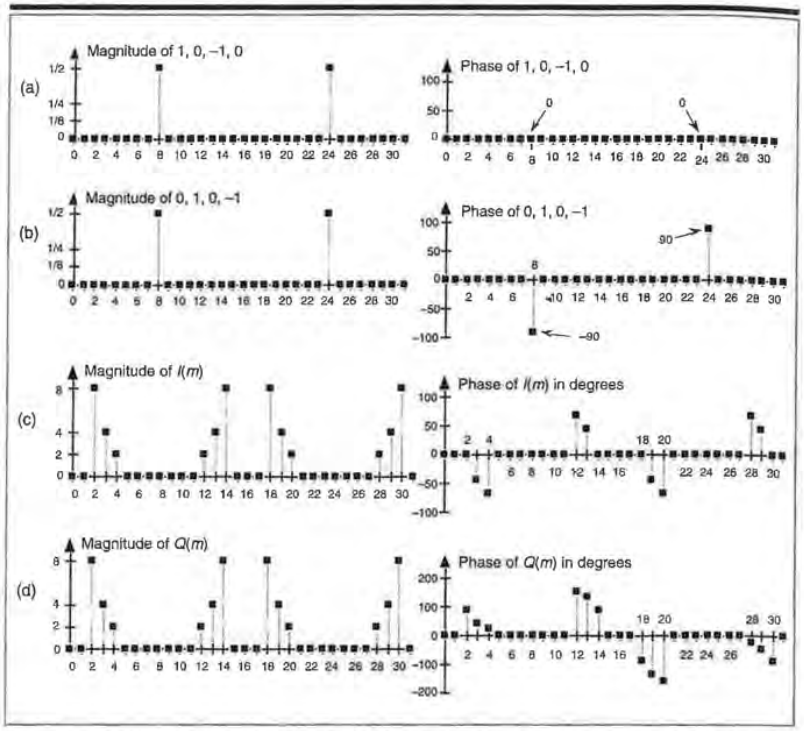


Figure 10-9 Frequency translation by $f_s/4$: (a) normalized magnitude and phase of cosine 1, 0, –1, 0 sequence; (b) normalized magnitude and phase of sine 0, 1, 0, –1 sequence; (c) magnitude and phase of frequency-translated in-phase $I(m)$; (d) magnitude and phase of frequency-translated quadrature-phase $Q(m)$.

The question is "Why would the sequences 1, 0, –1, 0 and 0, 1, 0, –1 ever be used if they induce a signal amplitude loss in $i(n)$ and $q(n)$?" The answer is that the alternating zero-valued samples reduce the amount of follow-on processing performed on $i(n)$ and $q(n)$. Let's say, for example, that an application requires both $i(n)$ and $q(n)$ to be low-pass filtered. When alternating samples of $i(n)$ and $q(n)$ are zeros, the digital filters have only half as many multiplications to perform because multiplications by zero are unnecessary.

Another way to look at this situation is that $i(n)$ and $q(n)$, in a sense, have been decimated by a factor of 2, and the necessary follow-on processing rates (operations/second) are also being reduced by a factor of 2. If $i(n)$ and $q(n)$ are, indeed, applied to two separate FIR digital filters, we can be clever and embed the mixing sequences' plus and minus ones and zeros into the

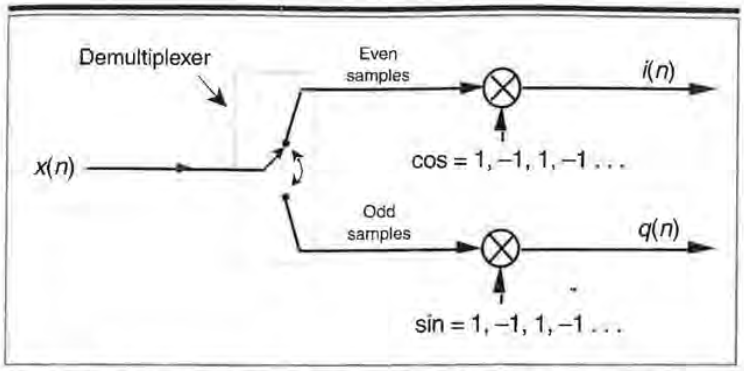


Figure 10-10 Quadrature downconversion by $f_s/4$ using a demultiplexer (demux) and the sequence 1, -1, 1, -1, ...

filters' coefficient values and avoid actually performing any multiplications. Because some coefficients are zero, they need not be used at all, and the number of actual multipliers used can be reduced. In that way, we'll have performed quadrature mixing and FIR filtering in the same process with a simpler filter. This technique also forces us to be aware of the potential frequency aliasing problems that may occur if the input signal is not sufficiently bandwidth limited relative to the original sample rate.

Figure 10-10 illustrates an interesting hybrid technique using the $f_s/2$ mixing sequence (1, –1, 1, –1) to perform quadrature mixing and downconversion by $f_s/4$. This scheme uses a demultiplexing process of routing alternate input samples to one of the two mixer paths[3–6]. Although both digital mixers use the same mixing sequence, this process is equivalent to multiplying the input by the two quadrature mixing sequences shown in Figure 10-8(a) and 10-8(b) with their frequency-domain magnitudes indicated in Figure 10-9(a) and 10-9(b). That's because alternate samples are routed to the two mixers. Although this scheme can be used for the quadrature sampling and demodulation described in Section 7.2, interpolation filters must be used to remove the inherent half sample time delay between $i(n)$ and $q(n)$ caused by using the single mixing sequence of 1,–1,1,–1.

Table 10-1 summarizes the effect of multiplying time-domain signal samples by various digital mixing sequences of ones, zeros, and minus ones.

A "yes" in the last column in Table 10-1 indicates that alternating samples of $i(n)$ and $q(n)$ can be discarded with no adverse impact on their spectra, allowing a follow-on processing data rate that's half the original $x(n)$ data rate.

Table 10-1 Digital Mixing Sequences

In-phase sequence	*Quadrature sequence*	*Frequency translation by*	*Scale factor*	*Final signal power loss*	*Decimation can occur*
1, –1, 1, –1, . . .	–	$f_s/2$	1	0 dB	no
1, –1, –1, 1, . . .	1, 1, -1, –1, . . .	$f_s/4$	$1/\sqrt{2}$	0 dB	yes
1, 0, –1, 0, . . .	0, 1, 0, –1, . . .	$f_s/4$	1/2	3 dB	yes
1, –1, 1, –1, . . . (with demux)	–1, 1, –1, 1, . . . (with demux)	$f_s/4$	1/2	3 dB	no

10.2 High-Speed Vector-Magnitude Approximation

The quadrature processing techniques employed in spectrum analysis, computer graphics, and digital communications routinely require high-speed determination of the magnitude of a complex vector V given its real and imaginary parts; i.e., the in-phase part I and the quadrature-phase part Q[4]. This magnitude calculation requires a square root operation because the magnitude of V is

$$|V| = \sqrt{I^2 + Q^2} \quad . \tag{10-24}$$

Assuming that the sum $I^2 + Q^2$ is available, the problem is to efficiently perform the square root operation.

There are several ways to obtain square roots, but the optimum technique depends on the capabilities of the available hardware and software. For example, when performing a square root using a high-level software language, we employ whatever software square root function is available. Although accurate, software routines can be very slow. In contrast, if a system must accomplish a square root operation in 50 nanoseconds, high-speed magnitude approximations are required[7,8]. Let's look at a neat magnitude approximation scheme that's particularly efficient.

10.2.1 αMax+βMin Algorithm

There is a technique called the αMax+βMin (read as "alpha max plus beta min") algorithm for calculating the magnitude of a complex vector.† It's a

† A "Max+βMin" algorithm had been in use, but in 1988 this author suggested expanding it to the αMax+βMin form, where α could be a value other than unity[9].

linear approximation to the vector-magnitude problem that requires determining which orthogonal vector, I or Q, has the greater absolute value. If the maximum absolute value of I or Q is designated by Max and the minimum absolute value of either I or Q is Min, an approximation of $|V|$, using the αMax+βMin algorithm, is expressed as

$$|V| \approx \alpha Max + \beta Min . \quad (10\text{-}25)$$

There are several pairs for the α and β constants that provide varying degrees of vector-magnitude approximation accuracy to within 0.1dB[7,10]. The αMax+βMin algorithms in reference [10] determine a vector magnitude at whatever speed it takes a system to perform a magnitude comparison, two multiplications, and one addition. But, as a minimum, those algorithms require a 16-bit multiplier to achieve reasonably accurate results. However, if hardware multipliers are not available, all is not lost. By restricting the α and β constants to reciprocals of integral powers of 2, Eq. (10-25) lends itself well to implementation in binary integral arithmetic. A prevailing application of the αMax+βMin algorithm uses α= 1.0 and β = 0.5[11,12]. The 0.5 multiplication operation is performed by shifting the minimum quadrature vector magnitude, Min, to the right by 1 bit. We can gauge the accuracy of any vector magnitude estimation by plotting its error as a function of vector phase angle. Let's do that. The αMax+βMin estimate for a complex vector of unity magnitude, using

$$|V| \approx Max + \frac{Min}{2} , \quad (10\text{-}26)$$

over the vector angular range of 0 to 90°, is shown as the solid curve in Figure 10-11. (The curves in Figure 10-11, of course, repeat every 90°.)

An ideal estimation curve for a unity magnitude vector would have an average value of one and an error standard deviation (σ_e) of zero; that is, having $\sigma_e = 0$ means that the ideal curve is flat—because the curve's value is one for all vector angles and its average error is zero. We'll use this ideal estimation curve as a yardstick to measure the merit of various αMax+βMin algorithms. Let's make sure we know what the solid curve in Figure 10-11 is telling us. It indicates that a unity magnitude vector oriented at an angle of approximately 26° will be estimated by Eq. (10-26) to have a magnitude of 1.118 instead of the correct magnitude of one. The error then, at 26°, is 11.8 percent, or 0.97 dB. Analyzing the entire solid curve in Figure 10-11 results in $\sigma_e = 0.032$ and an average error, over the 0 to 90° range, of 8.6 percent (0.71 dB).

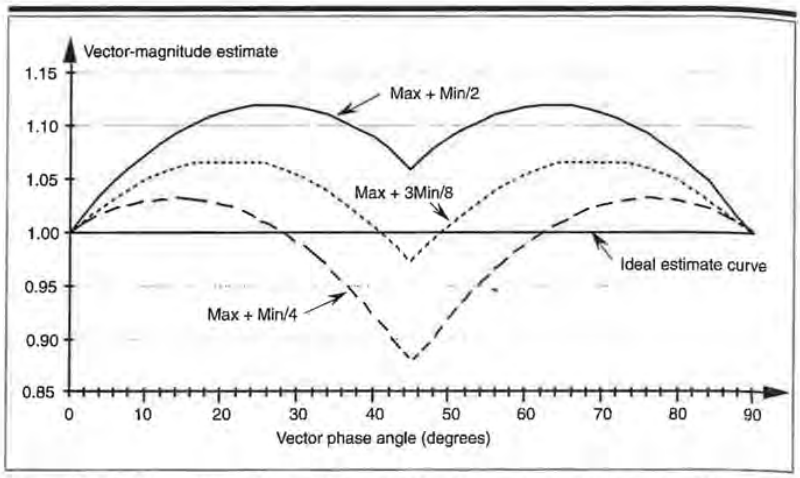


Figure 10-11 Normalized αMax+βMin estimates for α = 1, β = 1/2, and β = 1/4.

To reduce the average error introduced by Eq. (10-26), it is equally convenient to use a β value of 0.25, such as

$$|V| \approx \text{Max} + \frac{\text{Min}}{4} . \tag{10-27}$$

Equation (10-27), whose β multiplication is realized by shifting the digital value Min 2 bits to the right, results in the normalized magnitude approximation shown as the dashed curve in Figure 10-11. Although the largest error of 11.6 percent at 45° is similar in magnitude to that realized from Eq. (10-26), Eq. (10-27) has reduced the average error to −0.64 percent (−0.06 dB) and produced a slightly larger standard deviation of $\sigma_e = 0.041$. Though not as convenient to implement as Eqs. (10-26) and (10-27), a β value of 3/8 has been used to provide even more accurate vector magnitude estimates[13]. Using

$$|V| \approx \text{Max} + \frac{3 \cdot \text{Min}}{8} \tag{10-27'}$$

provides the normalized magnitude approximation shown as the dotted curve in Figure 10-11. Equation (10-27') results in magnitude estimates, whose largest error is only 6.8 percent, and a reduced standard deviation of $\sigma_e = 0.026$.

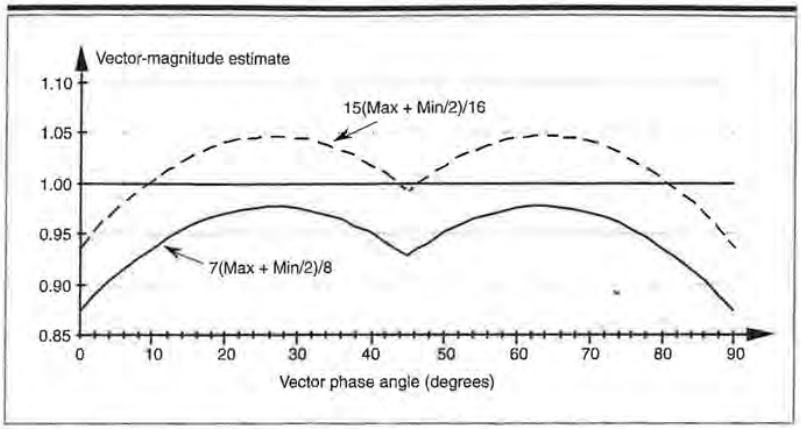


Figure 10-12 αMax+βMin estimates for α = 7/8, β = 7/16 and α = 15/16, β = 15/32.

Although the values for α and β in Figure 10-11 yield rather accurate vector-magnitude estimates, there are other values for α and β that deserve our attention because they result in smaller error standard deviations. Consider α = 7/8 and β = 7/16 where

$$|V| \approx \frac{7}{8}\text{Max} + \frac{7}{16}\text{Min} = \frac{7}{8}\left(\text{Max} + \frac{\text{Min}}{2}\right) . \qquad (10\text{-}28)$$

Equation (10-28), whose normalized results are shown as the solid curve in Figure 10-12, provides an average error of –5.01 percent and $\sigma_e = 0.028$. The 7/8ths factor applied to Eq. (10-26) produces both a smaller σ_e and a reduced average error—it lowers and flattens out the error curve from Eq. (10-26).

A further improvement can be obtained with α = 15/16 and β = 15/32 where

$$|V| \approx \frac{15}{16}\text{Max} + \frac{15}{32}\text{Min} = \frac{15}{16}\left(\text{Max} + \frac{\text{Min}}{2}\right) . \qquad (10\text{-}29)$$

Equation (10-29), whose normalized results are shown as the dashed curve in Figure 10-12, provides an average error of 1.79 percent and $\sigma_e = 0.030$. At the expense of a slightly larger σ_e, Eq. (10-29) provides an average error that is reduced below that provided by Eq. (10-28).

Although Eq. (10-29) appears to require two multiplications and one addition, its digital hardware implementation can be straightforward, as

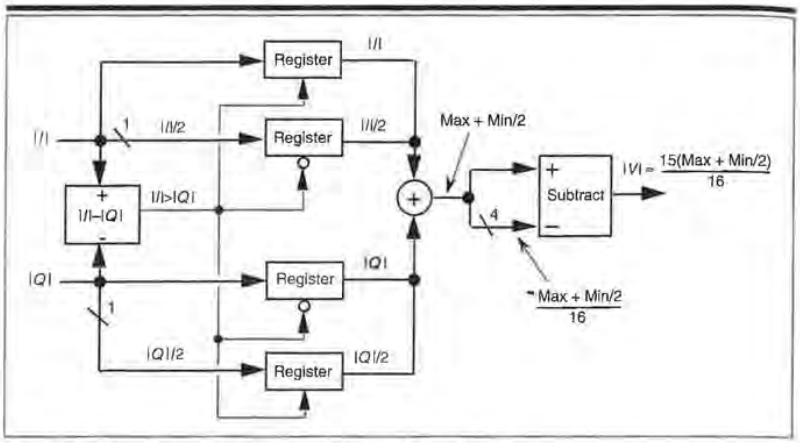


Figure 10-13 Hardware implementation of Eq. (10-29).

shown in Figure 10-13. The diagonal lines, \1 for example, denote a hardwired shift of 1 bit to the right to implement a divide-by-two operation by truncation. Likewise, the \4 symbol indicates a right shift by 4 bits to realize a divide-by-16 operation. The $|I|>|Q|$ control line is TRUE when the magnitude of I is greater than the magnitude of Q, so that Max = $|I|$ and Min = $|Q|$. This condition enables the registers to apply the values $|I|$ and $|Q|/2$ to the adder. When $|I| > |Q|$ is FALSE, the registers apply the values $|Q|$ and $|I|/2$ to the adder. Notice that the output of the adder, Max + Min/2, is the result of Eq. (10-26). Equation (10-29) is implemented via the subtraction of (Max + Min/2)/16 from Max + Min/2.

In Figure 10-13, all implied multiplications from Eq. (10-29) are performed by hardwired bit shifting, and the total execution time is limited only by the delay times associated with the hardware components.

10.2.2 Overflow Errors

In Figures 10-11 and 10-12, notice that we have a potential overflow problem with the results of Eqs. (10-26), (10-27), and (10-29) because the estimates can exceed the correct normalized vector-magnitude values; i.e., some magnitude estimates are greater than one. This means that, although the correct magnitude value may be within the system's full-scale word width, the algorithm result may exceed the work width of the system and cause overflow errors. With αMax+βMin algorithms, the user must be certain that no true vector magnitude exceeds the value that will produce an estimated magnitude greater than the maximum allowable word width. For example,

when using Eq. (10-26), we must ensure that no true vector magnitude exceeds 89.4 percent (1/1.118) of the maximum allowable work width.

10.2.3 Truncation Errors

The penalty we pay for the convenience of having α and β as powers of two is the error induced by the division-by-truncation process; and, thus far, we haven't taken that error into account. The error curves in Figure 10-11 and Figure 10-12 were obtained using a software simulation with its floating-point accuracy and are useful in evaluating different α and β values. However, the true error introduced by the αMax+βMin algorithm will be somewhat different from that shown in Figures 10-11 and 10-12 due to division errors when truncation is used with finite word widths.[†] For αMax+βMin schemes, the truncation errors are a function of the data's word width, the algorithm used, the values of both $|I|$ and $|Q|$, and the vector's phase angle. (These errors due to truncation compound the errors already inherent in our αMax+βMin algorithms.) Thus, a complete analysis of the truncation errors is beyond the scope of this book. What we can do, however, is illustrate a few truncation error examples.

Figure 10-14 shows the percent truncation errors using Eq. (10-29) for vector magnitudes of 4 to 512. Two vector phase angles were chosen to

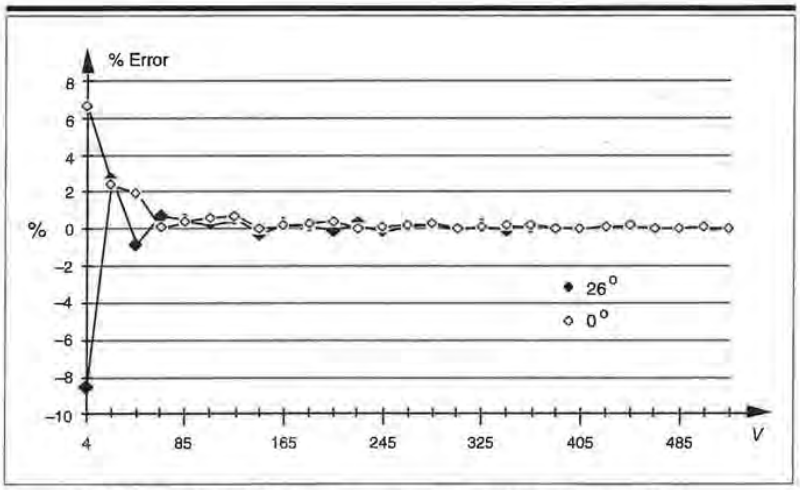


Figure 10-14 Equation (10-29) truncation error vs. vector magnitude $|V|$.

[†] Errors associated with division-by-truncation are covered in more detail in Section 9.3.

Table 10-2 α Max+βMin Algorithm Comparisons

Algorithm |V| ≈	*Largest error (%)*	*Largest error (dB)*	*Average error (%)*	*Average error (dB)*	*Standard deviation* σ_e	*Max* |V| *(% F.S.)*
Max + Min/2	11.8%	0.97 dB	8.6%	0.71 dB	0.032	89.4%
Max + Min/4	–11.6%	–1.07 dB	–0.64%	–0.06 dB	0.041	97.0%
Max + 3Min/8	6.8%	0.57 dB	3.97%	0.34 dB	0.026	93.6%
7(Max + Min/2)/8	–12.5%	–1.16 dB	–4.99%	–0.45 dB	0.028	100%
15(Max + Min/2)/16	–6.25%	–0.56 dB	1.79%	0.15 dB	0.030	95.4%

illustrate these truncation errors. The first is 26° because this is the phase angle where the most positive algorithm error occurs, and the second is 0° because this is the phase angle that introduces the greatest negative algorithm error. Notice that, at small vector magnitudes, the truncation errors are as great as 9 percent, but for an eight-bit system (maximum vector magnitude = 255) the truncation error is less than 1 percent. As the system word width increases, the truncation errors approach 0 percent. This means that truncation errors add very little to the inherent αMax+βMin algorithm errors.

The relative performance of the various algorithms is summarized in Table 10-2. The last column in Table 10-2 illustrates the maximum allowable true vector magnitude as a function of the system's full-scale (F.S.) word width to avoid overflow errors.

So, the αMax+βMin algorithm enables high-speed, vector-magnitude computation without the need for math coprocessor or hardware multiplier chips. Of course with the recent availability of high-speed, floating-point multiplier integrated circuits—with their ability to multiply or divide by nonintegral numbers in one or two clock cycles—α and β may not always need to be restricted to reciprocals of integral powers of two. It's also worth mentioning that this algorithm can be nicely implemented in a single hardware integrated circuit (for example, a field programmable gate array) affording high-speed operation.

10.3 Data Windowing Tricks

There are two useful schemes associated with using window functions on input data applied to a DFT or an FFT. The first technique is an efficient implementation of the Hanning (raised cosine) and Hamming windows

to reduce leakage in the FFT. The second scheme is related to minimizing the amplitude loss associated with using windows.

10.3.1 Windowing in the Frequency Domain

There's a clever technique for minimizing the calculations necessary to implement FFT input data windowing to reduce spectral leakage. There are times when we need the FFT of unwindowed time-domain data, and at the same time, we also want the FFT of that same time data with a window function applied. In this situation, we don't have to perform two separate FFTs. We can perform the FFT of the unwindowed data, and then we can perform frequency-domain windowing on that FFT result to reduce leakage. Let's see how.

Recall from Section 3.9 that the expressions for the Hanning and the Hamming windows were $w_{Han}(n) = 0.5-0.5\cos(2\pi n/N)$, and $w_{Ham}(n) = 0.54 - 0.46\cos(2\pi n/N)$, respectively. They both have the general cosine function form of

$$w(n) = \alpha - \beta\cos(2\pi n/N), \tag{10-30}$$

for $n = 0, 1, 2, \ldots, N-1$. Looking at the frequency response of the general cosine window function, using the definition of the DFT, the transform of Eq. (10-30) is expressed by

$$W(m) = \sum_{n=0}^{N-1}[\alpha - \beta\cos(2\pi n/N)]e^{-j2\pi nm/N}. \tag{10-31}$$

Because $\cos(2\pi n/N) = \dfrac{e^{j2\pi n/N}}{2} + \dfrac{e^{-j2\pi n/N}}{2}$, Eq. (10-31) can be rewritten as

$$W(m) = \sum_{n=0}^{N-1}\alpha e^{-j2\pi nm/N} - \frac{\beta}{2}\sum_{n=0}^{N-1}e^{j2\pi n/N}e^{-j2\pi nm/N} - \frac{\beta}{2}\sum_{n=0}^{N-1}e^{-j2\pi n/N}e^{-j2\pi nm/N}$$

$$= \sum_{n=0}^{N-1}\alpha e^{-j2\pi nm/N} - \frac{\beta}{2}\sum_{n=0}^{N-1}e^{-j2\pi n(m-1)/N} - \frac{\beta}{2}\sum_{n=0}^{N-1}e^{-j2\pi n(m+1)/N}. \tag{10-32}$$

Equation (10-32) looks pretty complicated, but, using the derivation from Section 3.13 for expressions like those summations, we find that Eq. (10-32) merely results in the superposition of three $\sin(x)/x$ functions in the frequency domain. Their amplitudes are shown in Figure 10-15.

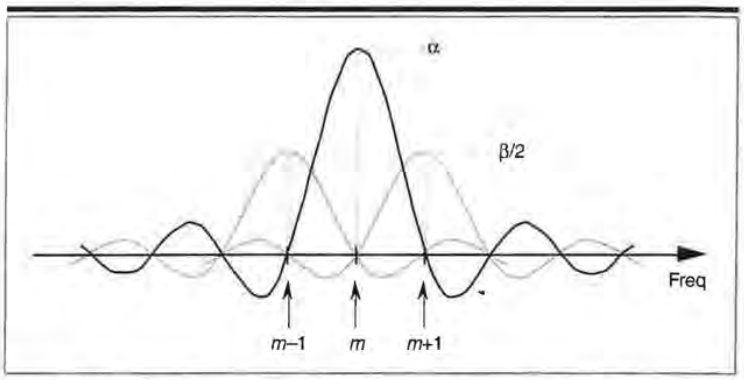


Figure 10-15 General cosine window frequency-response amplitude.

Notice that the two translated $\sin(x)/x$ functions have sidelobes with phase opposite from that of the center $\sin(x)/x$ function. This means that α times the mth bin output, minus $\beta/2$ times the $(m-1)$th bin output, minus $\beta/2$ times the $(m+1)$th bin output will minimize the sidelobes of the mth bin. This frequency-domain convolution process is equivalent to multiplying the input time data sequence by the N-valued window function $w(n)$ in Eq. (10-30)[14,15].

For example, let's say the output of the mth FFT bin is $X(m) = a_m + jb_m$, and the outputs of its two neighboring bins are $X(m-1) = a_{-1} + jb_{-1}$ and $X(m+1) = a_{+1} + jb_{+1}$. Then frequency-domain windowing for the mth bin of the unwindowed $X(m)$ is as follows:

$$X_{\text{windowed}}(m) = \alpha X(m) - \frac{\beta}{2}X(m-1) - \frac{\beta}{2}X(m+1)$$

$$= \alpha(a_m + jb_m) - \frac{\beta}{2}(a_{-1} + jb_{-1}) - \frac{\beta}{2}(a_{+1} + jb_{+1})$$

$$= \alpha a_m - \frac{\beta}{2}(a_{-1} + a_{+1}) + j[\alpha b_m - \frac{\beta}{2}(b_{-1} + b_{+1})] \quad . \tag{10-33}$$

To get a windowed N-point FFT, then, we can apply Eq. (10-33), requiring $4N$ additions and $3N$ multiplications, to the unwindowed FFT result and avoid having to perform the N multiplications of time-domain windowing and a second FFT with its $N\log_2 N$ additions and $2N\log_2 N$ multiplications.

The neat situation here is the α and β values for the Hanning window. They're both 0.5, and the products in Eq. (10-33) can be obtained in hardware with binary shifts by a single bit for α and two shifts for $\beta/2$. Thus, no multiplications are necessary to implement the Hanning frequency-domain windowing scheme. The issues that we need to consider are the window function best for the application and the efficiency of available hardware in performing the frequency-domain multiplications.

Along with the Hanning and Hamming windows, reference [15] describes a family of windows known as Blackman and Blackman-Harris windows that are also very useful for performing frequency-domain windowing. (Be aware that reference [15] has two typographical errors in the 4-Term (–74 dB) window coefficients column on its page 65. Reference [16] specifies that those coefficients should be 0.40217, 0.49703, 0.09892, and 0.00188.) Let's finish our discussion of frequency-domain windowing by saying that this scheme can be efficient because we don't have to window the entire set of FFT data. Frequency-domain windowing need be performed only on those FFT bins that are of interest to us.

10.3.2 Minimizing Window-Processing Loss

In Section 3.9, we stated that nonrectangular window functions reduce the overall signal levels applied to the FFT. Recalling Figure 3-16(a), we see that the peak response of the Hanning window function, for example, is half that obtained with the rectangular window because the input signal is attenuated at the beginning and end edges of the window sample interval, as shown in Figure 10-16(a). In terms of signal power, this attenuation results in a 6 dB loss. Going beyond the signal-power loss, window edge effects can be a problem when we're trying to detect short-duration signals that may occur right when the window function is at its edges. Well, some early digital signal processing practitioners tried to get around this problem by using dual window functions.

The first step in the dual window process is windowing the input data with a Hanning window function and taking the FFT of the windowed data. Then the same input data sequence is windowed against the inverse of the Hanning, and another FFT is performed. (The inverse of the Hanning window is depicted in Figure 10-16(b).) The two FFT results are then averaged. Using the dual window functions shown in Figure 10-16 enables signal energy attenuated by one window to be multiplied by the full gain of the other window. This technique seemed like a reasonable idea at the time, but, depending on the original signal, there could be

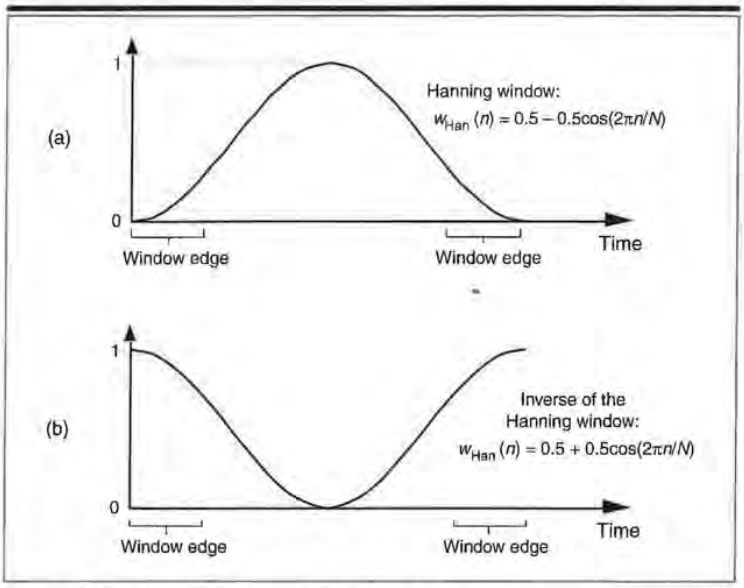


Figure 10-16 Dual windows used to reduce windowed-signal loss.

excessive leakage from the inverse window in Figure 10-16(b). Remember, the purpose of windowing was to ensure that the first and last data sequence samples, applied to an FFT, had the same value. The Hanning window guaranteed this, but the inverse window could not. Although this dual window technique made its way into the literature, it quickly fell out of favor. The most common technique used today to minimize signal loss due to window edge effects is known as *overlapped windows*.

The use of overlapped windows is depicted in Figure 10-17. It's a straightforward technique where a single *good* window function is applied multiple times to an input data sequence. Figure 10-17 shows an N-point window function applied to the input time series data four times resulting in four separate N-point data sequences. Next, four separate N-point FFTs are performed, and their outputs averaged. Notice that any input sample value that's fully attenuated by one window will be multiplied by the full gain of the following window. Thus, all input samples will contribute to the final averaged FFT results, and the window function keeps leakage to a minimum. (Of course, the user has to decide which particular window function is best for the application.) Figure 10-17 shows a window overlap of 50 percent where each input data sample contributes to the results of two FFTs. It's not uncommon to see an overlap of

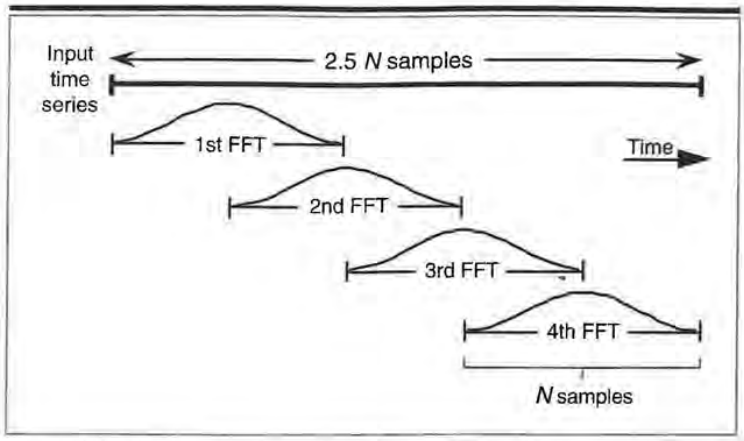


Figure 10-17 Windows overlapped by 50 percent to reduce windowed-signal loss.

75 percent being used where each input data sample would contribute to the results of the three individual FFTs. Of course the 50 percent and 75 percent overlap techniques increase the amount of total signal processing required, but, depending on the application, the improved signal sensitivity may justify the extra number crunching.

10.4 Fast Multiplication of Complex Numbers

The multiplication of two complex numbers is one of the most common functions performed in digital signal processing. It's mandatory in all discrete and fast Fourier transformation algorithms, necessary for graphics transformations, and used in processing digital communications signals. Be it in hardware or software, it's always to our benefit to streamline the processing necessary to perform a complex multiplication whenever we can. If the available hardware can perform three additions faster than a single multiplication, there's a way to speed up a complex multiplication operation[17].

The multiplication of two complex numbers $a + jb$ and $c + jd$, results in the complex product

$$R + jI = (a + jb) \cdot (c + jd) = (ac - bd) + j(bc + ad) . \qquad (10\text{-}34)$$

We can see that Eq. (10-34) required four multiplications and two additions. (From a computational standpoint, we'll assume that a subtraction

is equivalent to an addition.) Instead of using Eq. (10-34), we can calculate the following intermediate values:

$$k_1 = a(c + d) ,$$
$$k_2 = d(a + b) ,$$

and

$$k_3 = c(b - a) . \tag{10-35}$$

Then we perform the following operations to get the final R and I:

$$R = k_1 - k_2 ,$$

and

$$I = k_1 + k_3 . \tag{10-36}$$

The reader is invited to plug the k values from Eq. (10-35) into Eq. (10-36) to verify that the expressions in Eq. (10-36) are equivalent to Eq. (10-34). The intermediate values in Eq. (10-35) required three additions and three multiplications, whereas the results in Eq. (10-36) required two more additions. So we traded one of the multiplications required in Eq. (10-34) for three addition operations needed by Eq. (10-35) and Eq. (10-36). If our hardware uses fewer clock cycles to perform three additions than a single multiplication, we may well gain overall processing speed by using Eq. (10-35) and Eq. (10-36) for complex multiplication, instead of Eq. (10-34).

10.5 Efficiently Performing the FFT of Real Sequences

Upon recognizing its linearity property and understanding the odd and even symmetries of the transform's output, the early investigators of the fast Fourier transform (FFT) realized that two separate, real N-point input data sequences could be transformed using a single N-point complex FFT. They also developed a technique using a single N-point complex FFT to transform a $2N$-point real input sequence. Let's see how these two techniques work.

10.5.1 Performing Two *N*-Point Real FFTs

The standard FFT algorithms were developed to accept complex inputs; that is, the FFT's normal input $x(n)$ sequence is assumed to comprise real and imaginary parts, such as

$$\begin{aligned} x(0) &= x_r(0) + jx_i(0)\,, \\ x(1) &= x_r(1) + jx_i(1)\,, \\ x(2) &= x_r(2) + jx_i(2)\,, \\ &\cdots \\ &\cdots \\ x(N-1) &= x_r(N-1) + jx_i(N-1)\,. \end{aligned} \tag{10-37}$$

In typical signal processing schemes, FFT input data sequences are usually real. The most common example of this is the FFT input samples coming from an A/D converter that provides real integer values of some continuous (analog) signal. In this case the FFT's imaginary $x_i(n)$'s inputs are all zero. So initial FFT computations performed on the $x_i(n)$ inputs represent wasted operations. Early FFT pioneers recognized this inefficiency, studied the problem, and developed a technique where two independent N-point, *real* input data sequences could be transformed by a single N-point complex FFT. We call this scheme the Two N-Point Real FFTs algorithm. The derivation of this technique is straightforward and described in the literature[18–20]. If two N-point, real input sequences are $a(n)$ and $b(n)$, they'll have discrete Fourier transforms represented by $X_a(m)$ and $X_b(m)$. If we treat the $a(n)$ sequence as the real part of an FFT input and the $b(n)$ sequence as the imaginary part of the FFT input, then

$$\begin{aligned} x(0) &= a(0) + jb(0)\,, \\ x(1) &= a(1) + jb(1)\,, \\ x(2) &= a(2) + jb(2)\,, \\ &\cdots \\ &\cdots \\ x(N-1) &= a(N-1) + jb(N-1)\,. \end{aligned} \tag{10-38}$$

Applying the $x(n)$ values from Eq. (10-38) to the standard DFT,

$$X(m) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi nm/N}\,, \tag{10-39}$$

we'll get an DFT output $X(m)$ where m goes from 0 to N–1. (We're assuming, of course, that the DFT is implemented by way of an FFT algorithm.) Using the superscript * symbol to represent the complex conjugate, we can extract the two desired FFT outputs $X_a(m)$ and $X_b(m)$ from $X(m)$ by using the following:

$$X_a(m) = \frac{X^*(N-m) + X(m)}{2}, \tag{10-40}$$

and

$$X_b(m) = \frac{j[X^*(N-m) - X(m)]}{2}. \tag{10-41}$$

Let's break Eqs. (10-40) and (10-41) into their real and imaginary parts to get expressions for $X_a(m)$ and $X_b(m)$ that are easier to understand and implement. Using the notation showing $X(m)$'s real and imaginary parts, where $X(m) = X_r(m) + jX_i(m)$, we can rewrite Eq. (10-40) as

$$X_a(m) = \frac{X_r(N-m) + X_r(m) + j[X_i(m) - X_i(N-m)]}{2} \tag{10-42}$$

where $m = 1, 2, 3, \ldots, N-1$. What about the first $X_a(m)$, when $m = 0$? Well, this is where we run into a bind if we actually try to implement Eq. (10-40) directly. Letting $m = 0$ in Eq. (10-40), we quickly realize that the first term in the numerator, $X^*(N-0) = X^*(N)$, isn't available because the $X(N)$ sample does not exist in the output of an N-point FFT! We resolve this problem by remembering that $X(m)$ is periodic with a period N, so $X(N) = X(0)$.† When $m = 0$, Eq. (10-40) becomes

$$X_a(0) = \frac{X_r(0) - jX_i(0) + X_r(0) + jX_i(0)}{2} = X_r(0). \tag{10-43}$$

Next, simplifying Eq. (10-41),

$$X_b(m) = \frac{j[X_r(N-m) - jX_i(N-m) - X_r(m) - jX_i(m)]}{2}$$

$$= \frac{X_i(N-m) + X_i(m) + j[X_r(N-m) - X_r(m)]}{2} \tag{10-44}$$

where, again, $m = 1, 2, 3, \ldots, N-1$. By the same argument used for Eq. (10-43), when $m = 0$, $X_b(0)$ in Eq. (10-44) becomes

† This fact is illustrated in Section 3.8 during the discussion of spectral leakage in DFTs.

$$X_b(0) = \frac{X_i(0) + X_i(0) + j[X_r(0) - X_r(0)]}{2} = X_i(0) \ . \qquad (10\text{-}45)$$

This discussion brings up a good point for beginners to keep in mind. In the literature Eqs. (10-40) and (10-41) are often presented without any discussion of the $m = 0$ problem. So, whenever you're grinding through an algebraic derivation or have some equations tossed out at you, be a little skeptical. Try the equations out on an example—see if they're true. (After all, both authors and book typesetters are human and sometimes make mistakes. We had an old saying in Ohio for this situation: "Trust everybody, but cut the cards.") Following this advice, let's prove that this Two N-Point Real FFTs algorithm really does work by applying the 8-point data sequences from Chapter 3's DFT Examples to Eqs. (10-42) through (10-45). Taking the 8-point input data sequence from Section 3.1's DFT Example 1 and denoting it $a(n)$,

$$\begin{aligned} a(0) &= 0.3535, \quad a(1) = 0.3535, \\ a(2) &= 0.6464, \quad a(3) = 1.0607, \\ a(4) &= 0.3535, \quad a(5) = -1.0607, \\ a(6) &= -1.3535, \quad a(7) = -0.3535 \ . \end{aligned} \qquad (10\text{-}46)$$

Taking the 8-point input data sequence from Section 3.6's DFT Example 2 and calling it $b(n)$,

$$\begin{aligned} b(0) &= 1.0607, \quad b(1) = 0.3535, \\ b(2) &= -1.0607, \quad b(3) = -1.3535, \\ b(4) &= -0.3535, \quad b(5) = 0.3535, \\ b(6) &= 0.3535, \quad b(7) = 0.6464 \ . \end{aligned} \qquad (10\text{-}47)$$

Combining the sequences in Eqs. (10-46) and (10-47) into a single complex sequence $x(n)$,

$$\begin{aligned} x(n) = \ & \underset{\downarrow}{a(n)} \quad \underset{\downarrow}{b(n)} \\ & 0.3535 \quad + j\,1.0607 \\ & + 0.3535 \quad + j\,0.3535 \\ & + 0.6464 \quad - j\,1.0607 \\ & + 1.0607 \quad - j\,1.3535 \\ & + 0.3535 \quad - j\,0.3535 \\ & - 1.0607 \quad + j\,0.3535 \\ & - 1.3535 \quad + j\,0.3535 \\ & - 0.3535 \quad + j\,0.6464 \ . \end{aligned} \qquad (10\text{-}48)$$

Now, taking the 8-point FFT of the complex sequence in Eq. (10-48) we get

$$\begin{array}{rll} & X_r(m) & X_i(m) \\ & \downarrow & \downarrow \\ X(m) = & 0.0000 & +j\,0.0000 \quad \leftarrow m = 0 \text{ term} \\ & -2.8283 & -j\,1.1717 \quad \leftarrow m = 1 \text{ term} \\ & +2.8282 & +j\,2.8282 \quad \leftarrow m = 2 \text{ term} \\ & +0.0000 & +j\,0.0000 \quad \leftarrow m = 3 \text{ term} \\ & +0.0000 & +j\,0.0000 \quad \leftarrow m = 4 \text{ term} \\ & +0.0000 & +j\,0.0000 \quad \leftarrow m = 5 \text{ term} \\ & +0.0000 & +j\,0.0000 \quad \leftarrow m = 6 \text{ term} \\ & +2.8283 & +j\,6.8282 \quad \leftarrow m = 7 \text{ term} . \end{array} \qquad (10\text{-}49)$$

So from Eq. (10-43),

$$X_a(0) = X_r(0) = 0 .$$

To get the rest of $X_a(m)$, we have to plug the FFT output's $X(m)$ and $X(N-m)$ values into Eq. (10-42).† Doing so,

$$X_a(1) = \frac{X_r(7) + X_r(1) + j[X_i(1) - X_i(7)]}{2} = \frac{2.8283 - 2.8283 + j[-1.1717 - 6.8282]}{2}$$

$$= \frac{0 - j7.9999}{2} = 0 - j4.0 = 4\angle -90^\circ ,$$

$$X_a(2) = \frac{X_r(6) + X_r(2) + j[X_i(2) - X_i(6)]}{2} = \frac{0.0 + 2.8282 + j[2.8282 - 0.0]}{2}$$

$$= \frac{2.8282 + j2.8282}{2} = 1.414 + j1.414 = 2\angle 45^\circ ,$$

$$X_a(3) = \frac{X_r(5) + X_r(3) + j[X_i(3) - X_i(5)]}{2} = \frac{0.0 + 0.0 + j[0.0 - 0.0]}{2} = 0\angle 0^\circ ,$$

$$X_a(4) = \frac{X_r(4) + X_r(4) + j[X_i(4) - X_i(4)]}{2} = \frac{0.0 + 0.0 + j[0.0 - 0.0]}{2} = 0\angle 0^\circ ,$$

† Remember, when the FFT's input is complex, the FFT outputs may not be conjugate symmetric; that is, we can't assume that $F(m)$ is equal to $F^*(N-m)$ when the FFT input sequence's real and imaginary parts are both nonzero.

$$X_a(5) = \frac{X_r(3) + X_r(5) + j[X_i(5) - X_i(3)]}{2} = \frac{0.0 + 0.0 + j[0.0 - 0.0]}{2} = 0\angle 0°\ ,$$

$$X_a(6) = \frac{X_r(2) + X_r(6) + j[X_i(6) - X_i(2)]}{2} = \frac{2.8282 + 0.0 + j[0.0 - 2.8282]}{2}$$

$$= \frac{2.8282 - j2.8282}{2} = 1.414 - j1.414 = 2\angle -45°\ , \text{ and}$$

$$X_a(7) = \frac{X_r(1) + X_r(7) + j[X_i(7) - X_i(1)]}{2} = \frac{-2.8282 + 2.8282 + j[6.8282 + 1.1717]}{2}$$

$$= \frac{0.0 + j7.9999}{2} = 0 + j4.0 = 4\angle 90°\ .$$

So Eq. (10-42) really does extract $X_a(m)$ from the $X(m)$ sequence in Eq. (10-49). We can see that we need not solve Eq. (10-42) when m is greater than 4 (or $N/2$) because $X_a(m)$ will always be conjugate symmetric. Because $X_a(7) = X_a(1)$, $X_a(6) = X_a(2)$, etc., only the first $N/2$ elements in $X_a(m)$ are independent and need be calculated.

OK, let's keep going and use Eqs. (10-44) and (10-45) to extract $X_b(m)$ from the FFT output. From Eq. (10-45)

$$X_b(0) = X_i(0) = 0\ .$$

Plugging the FFT's output values into Eq. (10-44) to get the next four $X_b(m)$s, we have

$$X_b(1) = \frac{X_i(7) + X_i(1) + j[X_r(7) - X_r(1)]}{2} = \frac{6.8282 - 1.1717 + j[2.8283 + 2.8283]}{2}$$

$$= \frac{5.656 + j5.656}{2} = 2.828 + j2.828 = 4\angle 45°\ ,$$

$$X_b(2) = \frac{X_i(6) + X_i(2) + j[X_r(6) - X_r(2)]}{2} = \frac{0.0 + 2.8282 + j[0.0 - 2.8282]}{2}$$

$$= \frac{2.8282 - j2.8282}{2} = 1.414 - j1.414 = 2\angle -45°\ , \text{ and}$$

$$X_b(3) = \frac{X_i(5) + X_i(3) + j[X_r(5) - X_r(3)]}{2} = \frac{0.0 + 0.0 + j[0.0 - 0.0]}{2} = 0\angle 0^\circ \text{ , and}$$

$$X_b(4) = \frac{X_i(4) + X_i(4) + j[X_r(4) - X_r(4)]}{2} = \frac{0.0 + 0.0 + j[0.0 - 0.0]}{2} = 0\angle 0^\circ \text{ .}$$

The question arises "With the additional processing required by Eqs. (10-42) and (10-44) after the initial FFT, how much computational saving (or loss) is to be had by this Two N-Point Real FFTs algorithm?" We can estimate the efficiency of this algorithm by considering the number of arithmetic operations required relative to two separate N-point radix-2 FFTs. First, we estimate the number of arithmetic operations in two separate N-point complex FFTs.

From Section 4.2, we know that a standard radix-2 N-point complex FFT comprises $(N/2) \cdot \log_2 N$ butterfly operations. If we use the optimized butterfly structure, each butterfly requires one complex multiplication and two complex additions. Now, one complex multiplication requires two real additions and four real multiplications, and one complex addition requires two real additions.† So a single FFT butterfly operation comprises four real multiplications and six real additions. This means that a single N-point complex FFT requires $(4N/2) \cdot \log_2 N$ real multiplications, and $(6N/2) \cdot \log_2 N$ real additions. Finally, we can say that two separate N-point complex radix-2 FFTs require

(two N-point complex FFTs →) $4N \cdot \log_2 N$ real multiplications, and (10-50)

$6N \cdot \log_2 N$ real additions. (10-50')

Next, we need to determine the computational workload of the Two N-Point Real FFTs algorithm. If we add up the number of real multiplications and real additions required by the algorithm's N-point complex FFT, plus those required by Eq. (10-42) to get $X_a(m)$, and those required by Eq. (10-44) to get $X_b(m)$, the Two N-Point Real FFTs algorithm requires

(two N-Point Real FFTs algorithm →) $2N \cdot \log_2 N + N$ real multiplications, and (10-51)

$3N \cdot \log_2 N + 2N$ real additions. (10-51')

† The complex addition $(a+jb) + (c+jd) = (a+c) + j(b+d)$ requires two real additions. A complex multiplication $(a+jb) \cdot (c+jd) = ac-bd + j(ad+bc)$ requires two real additions and four real multiplications.

Equations (10-51) and (10-51') assume that we're calculating only the first $N/2$ independent elements of $X_a(m)$ and $X_b(m)$. The single N term in Eq. (10-51) accounts for the $N/2$ divide by 2 operations in Eq. (10-42) and the $N/2$ divide by 2 operations in Eq. (10-44).

OK, now we can find out how efficient the Two N-Point Real FFTs algorithm is compared to two separate complex N-point radix-2 FFTs. This comparison, however, depends on the hardware used for the calculations. If our arithmetic hardware takes many more clock cycles to perform a multiplication than an addition, then the difference between multiplications in Eqs. (10-50) and (10-51) is the most important comparison. In this case, the percentage gain in computational saving of the Two N-Point Real FFTs algorithm relative to two separate N-point complex FFTs is the difference in their necessary multiplications over the number of multiplications needed for two separate N-point complex FFTs, or

$$\frac{4N \cdot \log_2 N - (2N \cdot \log_2 N + N)}{4N \cdot \log_2 N} \cdot 100\% = \frac{2 \cdot \log_2 N - 1}{4 \cdot \log_2 N} \cdot 100\% \quad . \quad (10\text{-}52)$$

The computational (multiplications only) saving from Eq. (10-52) is plotted as the top curve of Figure 10-18. In terms of multiplications, for $N \geq 32$,

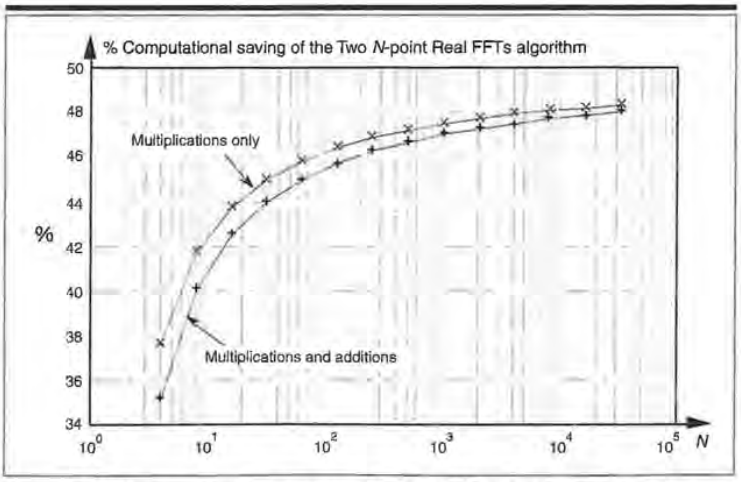


Figure 10-18 Computational saving of the Two N-Point Real FFTs algorithm over that of two separate N-point complex FFTs. The top curve indicates the saving when only multiplications are considered. The bottom curve is the saving when both additions and multiplications are used in the comparison.

the Two N-Point Real FFTs algorithm saves us over 45 percent in computational workload compared to two separate N-point complex FFTs.

For hardware using high-speed multiplier integrated circuits, multiplication and addition can take roughly equivalent clock cycles. This makes addition operations just as important and time consuming as multiplications. Thus the difference between those combined arithmetic operations in Eqs. (10-50) plus (10-50') and Eqs. (10-51) plus (10-51') is the appropriate comparison. In this case, the percentage gain in computational saving of our algorithm over two FFTs is their total arithmetic operational difference over the total arithmetic operations in two separate N-point complex FFTs, or

$$\frac{(4N \cdot \log_2 N + 6N \cdot \log_2 N) - (2N \cdot \log_2 N + N + 3N \cdot \log_2 N + 2N)}{4N \cdot \log_2 N + 6N \cdot \log_2 N} \cdot 100\%$$

$$= \frac{5 \cdot \log_2 N - 3}{10 \cdot \log_2 N} \cdot 100\% \quad . \tag{10-53}$$

The full computational (multiplications and additions) saving from Eq. (10-53) is plotted as the bottom curve of Figure 10-18. OK, that concludes our discussion and illustration of how a single N-point complex FFT can be used to transform two separate N-point real input data sequences.

10.5.2 Performing a 2N-Point Real FFT

Similar to the scheme above where two separate N-point real data sequences are transformed using a single N-point FFT, a technique exists where a $2N$-point real sequence can be transformed with a single complex N-point FFT. This $2N$-Point Real FFT algorithm, whose derivation is also described in the literature, requires that the $2N$-sample real input sequence be separated into two parts[20,21]. Not broken in two, but unzipped—separating the even and odd sequence samples. The N even-indexed input samples are loaded into the real part of a complex N-point input sequence $x(n)$. Likewise, the input's N odd-indexed samples are loaded into $x(n)$'s imaginary parts. To illustrate this process, let's say we have a $2N$-sample real input data sequence $a(n)$ where $0 \le n \le 2N-1$. We want $a(n)$'s $2N$-point transform $X_a(m)$. Loading $a(n)$'s odd/even sequence values appropriately into an N-point complex FFT's input sequence, $x(n)$,

$$\begin{aligned} x(0) &= a(0) + ja(1)\,, \\ x(1) &= a(2) + ja(3)\,, \\ x(2) &= a(4) + ja(5)\,, \\ &\cdots \\ &\cdots \\ x(N-1) &= a(2N-2) + ja(2N-1)\,. \end{aligned} \tag{10-54}$$

Applying the N complex values in Eq. (10-54) to an N-point complex FFT, we'll get an FFT output $X(m) = X_r(m) + jX_i(m)$, where m goes from 0 to $N-1$. To extract the desired $2N$-Point Real FFT algorithm output $X_a(m) = X_{a,\text{real}}(m) + jX_{a,\text{imag}}(m)$ from $X(m)$, let's define the following relationships

$$X_r^+(m) = \frac{X_r(m) + X_r(N-m)}{2}\,, \tag{10-55}$$

$$X_r^-(m) = \frac{X_r(m) - X_r(N-m)}{2}\,, \tag{10-56}$$

$$X_i^+(m) = \frac{X_i(m) + X_i(N-m)}{2}\,, \text{ and} \tag{10-57}$$

$$X_i^-(m) = \frac{X_i(m) - X_i(N-m)}{2}\,. \tag{10-58}$$

The values resulting from Eqs. (10-55) through (10-58) are, then, used as factors in the following expressions to obtain the real and imaginary parts of our final $X_a(m)$:

$$X_{a,\text{real}}(m) = X_r^+(m) + \cos(\frac{\pi m}{N}) \cdot X_i^+(m) - \sin(\frac{\pi m}{N}) \cdot X_r^-(m), \tag{10-59}$$

and

$$X_{a,\text{imag}}(m) = X_i^-(m) - \sin(\frac{\pi m}{N}) \cdot X_i^+(m) - \cos(\frac{\pi m}{N}) \cdot X_r^-(m)\,. \tag{10-60}$$

Remember now, the original $a(n)$ input index n goes from 0 to $2N-1$, and our N-point FFT output index m goes from 0 to $N-1$. We apply $2N$ real input time-domain samples to this algorithm and get back N complex frequency-domain samples representing the first half of the equivalent $2N$-point complex FFT, $X_a(0)$ through $X_a(N-1)$. Because this algorithm's

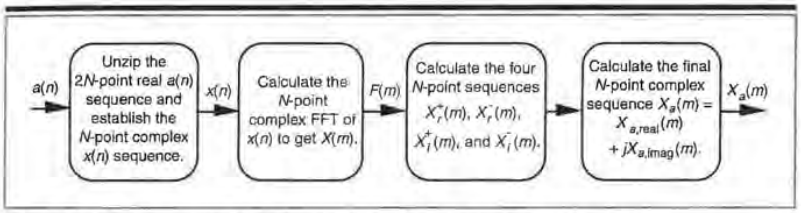


Figure 10-19 Computational flow of the 2*N*-Point Real FFT algorithm.

$a(n)$ input is constrained to be real, $X_a(N)$ through $X_a(2N-1)$ are merely the complex conjugates of their $X_a(0)$ through $X_a(N-1)$ counterparts and need not be calculated. To help us keep all of this straight, Figure 10-19 depicts the computational steps of the 2*N*-Point Real FFT algorithm.

To demonstrate this process by way of example, let's apply the 8-point data sequence from Eq. (10-46) to the 2*N*-Point Real FFT algorithm. Partitioning those Eq. (10-46) samples as dictated by Eq. (10-54), we have our new FFT input sequence:

$$\begin{aligned} x(0) &= 0.3535 + j\,0.3535, \\ x(1) &= 0.6464 + j\,1.0607, \\ x(2) &= 0.3535 - j\,1.0607, \\ x(3) &= -1.3535 - j\,0.3535\,. \end{aligned} \tag{10-61}$$

With $N = 4$ in this example, taking the 4-point FFT of the complex sequence in Eq. (10-61) we get

$$\begin{array}{rrl} X_r(m) & X_i(m) & \\ \downarrow & \downarrow & \\ X(m) = 0.0000 & +j\,0.0000 & \leftarrow m = 0 \text{ term} \\ +1.4142 & -j\,0.5857 & \leftarrow m = 1 \text{ term} \\ +1.4141 & -j\,1.4141 & \leftarrow m = 2 \text{ term} \\ -1.4142 & +j\,3.4141 & \leftarrow m = 3 \text{ term}\,. \end{array} \tag{10-62}$$

Using these values, we now get the intermediate factors from Eqs. (10-55) through (10-58). Calculating our first $X_r^+(0)$ value, again we're reminded that $X(m)$ is periodic with a period N, so $X(4) = X(0)$, and $X_r^+(0) = [X_r(0) + X_r(0)]/2 = 0$. Continuing to use Eqs. (10-55) through (10-58),

$$\begin{array}{llll} X_r^+(0) = 0, & X_r^-(0) = 0, & X_i^+(0) = 0, & X_i^-(0) = 0, \\ X_r^+(1) = 0, & X_r^-(1) = 1.4142, & X_i^+(1) = 1.4142, & X_i^-(1) = -1.9999, \\ X_r^+(2) = 1.4141, & X_r^-(2) = 0, & X_i^+(2) = -1.4144, & X_i^-(2) = 0, \\ X_r^+(3) = 0, & X_r^-(3) = -1.4142, & X_i^+(3) = 1.4142, & X_i^-(3) = 1.9999. \end{array} \tag{10-63}$$

Using the intermediate values from Eq. (10-63) in Eqs. (10-59) and (10-60),

$$X_{a,\text{real}}(0) = (0) + \cos\left(\frac{\pi \cdot 0}{4}\right) \cdot (0) - \sin\left(\frac{\pi \cdot 0}{4}\right) \cdot (0)$$

$$X_{a,\text{imag}}(0) = (0) - \sin\left(\frac{\pi \cdot 0}{4}\right) \cdot (0) - \cos\left(\frac{\pi \cdot 0}{4}\right) \cdot (0)$$

$$X_{a,\text{real}}(1) = (0) + \cos\left(\frac{\pi \cdot 1}{4}\right) \cdot (1.4142) - \sin\left(\frac{\pi \cdot 1}{4}\right) \cdot (1.4142)$$

$$X_{a,\text{imag}}(1) = (-1.9999) - \sin\left(\frac{\pi \cdot 1}{4}\right) \cdot (1.4142) - \cos\left(\frac{\pi \cdot 1}{4}\right) \cdot (1.4142)$$

$$X_{a,\text{real}}(2) = (1.4141) + \cos\left(\frac{\pi \cdot 2}{4}\right) \cdot (-1.4144) - \sin\left(\frac{\pi \cdot 2}{4}\right) \cdot (0)$$

$$X_{a,\text{imag}}(2) = (0) - \sin\left(\frac{\pi \cdot 2}{4}\right) \cdot (-1.4144) - \cos\left(\frac{\pi \cdot 2}{4}\right) \cdot (0)$$

$$X_{a,\text{real}}(3) = (0) + \cos\left(\frac{\pi \cdot 3}{4}\right) \cdot (1.4142) - \sin\left(\frac{\pi \cdot 3}{4}\right) \cdot (-1.4142)$$

$$X_{a,\text{imag}}(3) = (1.9999) - \sin\left(\frac{\pi \cdot 3}{4}\right) \cdot (1.4142) - \cos\left(\frac{\pi \cdot 3}{4}\right) \cdot (-1.4142) \quad (10\text{-}64)$$

Evaluating the sine and cosine terms in Eq. (10-64),

$$X_{a,\text{real}}(0) = (0) + (1) \cdot (0) - (0) \cdot (0) = 0,$$

$$X_{a,\text{imag}}(0) = (0) - (0) \cdot (0) - (1) \cdot (0) = 0,$$

$$X_{a,\text{real}}(1) = (0) + (0.7071) \cdot (1.4142) - (0.7071) \cdot (1.4142) = 0,$$

$$X_{a,\text{imag}}(1) = (-1.9999) - (0.7071) \cdot (1.4142) - (0.7071) \cdot (1.4142) = -3.9999,$$

$$X_{a,\text{real}}(2) = (1.4141) + (0) \cdot (-1.4144) - (1) \cdot (0) = 1.4141,$$

$$X_{a,\text{imag}}(2) = (0) - (1) \cdot (-1.4144) - (0) \cdot (0) = 1.4144,$$

$$X_{a,\text{real}}(3) = (0) + (-0.7071) \cdot (1.4142) - (0.7071) \cdot (-1.4142) = 0, \text{ and}$$

$$X_{a,\text{imag}}(3) = (1.9999) - (0.7071) \cdot (1.4142) - (-0.7071) \cdot (-1.4142) = 0 \,. \quad (10\text{-}65)$$

Combining the results of the terms in Eq. (10-65), we have our final correct answer of

$$X_a(0) = X_{a,\text{real}}(0) + jX_{a,\text{imag}}(0) = 0 + j0 = 0 \angle 0°,$$

$$X_a(1) = X_{a,\text{real}}(1) + jX_{a,\text{imag}}(1) = 0 - j3.999 = 4 \angle -90°,$$

$$X_a(2) = X_{a,\text{real}}(2) + jX_{a,\text{imag}}(2) = 1.4141 + j1.4144 = 2 \angle 45°, \text{ and}$$

$$X_a(3) = X_{a,\text{real}}(3) + jX_{a,\text{imag}}(3) = 0 + j0 = 0 \angle 0°. \quad (10\text{-}66)$$

After going through all the steps required by Eqs. (10-55) through (10-60), the reader might question the efficiency of this 2*N*-Point Real FFT algorithm. Using the same process as the above 2*N*-Point Real FFTs algorithm analysis, let's show that the 2*N*-Point Real FFT algorithm does provide some modest computational saving. First, we know that a single 2*N*-Point radix-2 FFT has $(2N/2)\cdot\log_2 2N = N\cdot(\log_2 N+1)$ butterflies and requires

2*N*-point complex FFT → $4N\cdot(\log_2 N+1)$ real multiplications (10-67)

and

$6N\cdot(\log_2 N+1)$ real additions. (10-67')

If we add up the number of real multiplications and real additions required by the algorithm's *N*-point complex FFT, plus those required by Eqs. (10-55) through (10-58) and those required by Eqs. (10-59) and (10-60), the complete 2*N*-Point Real FFT algorithm requires

2*N*-Point Real FFT algorithm → $2N\cdot\log_2 N + 8N$ real multiplications (10-68)

and

$3N\cdot\log_2 N + 8N$ real additions. (10-68')

OK, using the same hardware considerations (multiplications only) we used to arrive at Eq. (10-52), the percentage gain in multiplication saving of the 2*N*-Point Real FFT algorithm relative to a 2*N*-point complex FFT is

$$\frac{4N\cdot(\log_2 N+1)-(2N\cdot\log_2 N+8N)}{4N\cdot(\log_2 N+1)}\cdot 100\%$$

$$=\frac{2N\cdot\log_2 N+2N-N\cdot\log_2 N-4N}{2N\cdot\log_2 N+2N}\cdot 100\%$$

$$=\frac{\log_2 N-2}{2\cdot\log_2 N+2}\cdot 100\%\ . \qquad (10\text{-}69)$$

The computational (multiplications only) saving from Eq. (10-69) is plotted as the bottom curve of Figure 10-20. In terms of multiplications, the 2*N*-Point Real FFT algorithm provides a saving of >30% when $N\geq 128$ or whenever we transform input data sequences whose lengths are ≥256.

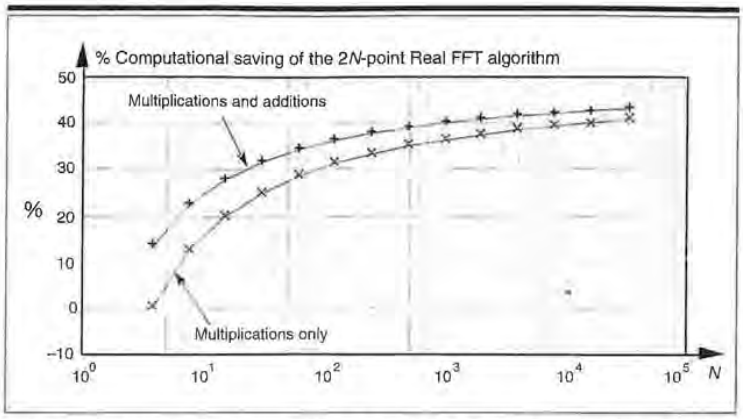


Figure 10-20 Computational saving of the 2*N*-Point Real FFT algorithm over that of a single 2*N*-point complex FFT. The top curve is the saving when both additions and multiplications are used in the comparison. The bottom curve indicates the saving when only multiplications are considered.

Again, for hardware using high-speed multipliers, we consider both multiplication and addition operations. The difference between those combined arithmetic operations in Eqs. (10-67) plus (10-67') and Eqs. (10-68) plus (10-68') is the appropriate comparison. In this case, the percentage gain in computational saving of our algorithm is

$$\frac{4N\cdot(\log_2 N+1)+6N\cdot(\log_2 N+1)-(2N\cdot\log_2 N+8N+3N\cdot\log_2 N+8N)}{4N\cdot(\log_2 N+1)+6N\cdot(\log_2 N+1)}\cdot 100\%$$

$$=\frac{10\cdot(\log_2 N+1)-5\cdot\log_2 N-16}{10\cdot(\log_2 N+1)}\cdot 100\%$$

$$=\frac{5\cdot\log_2 N-6}{10\cdot(\log_2 N+1)}\cdot 100\%. \qquad (10\text{-}70)$$

The full computational (multiplications and additions) saving from Eq. (10-70) is plotted as a function of N in the top curve of Figure 10-20.

10.6 Calculating the Inverse FFT Using the Forward FFT

There are many signal processing applications where the capability to perform the inverse FFT is necessary. This can be a problem if available

hardware or software routines have the capability to perform only the *forward* FFT. Fortunately, there are two slick ways to perform the inverse FFT using the forward FFT algorithm.

10.6.1 First Inverse FFT Method

The first inverse FFT calculation scheme is implemented following the processes shown in Figure 10-21. To see how this works, consider the expressions for the forward and inverse DFTs:

$$\text{Forward DFT} \rightarrow \qquad X(m) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi nm/N}, \tag{10-71}$$

and

$$\text{Inverse DFT} \rightarrow \qquad x(n) = \frac{1}{N}\sum_{m=0}^{N-1} X(m)e^{j2\pi mn/N}. \tag{10-72}$$

To reiterate our goal, we want to use the process in Eq. (10-71) to implement Eq. (10-72).

The first step of our approach is to use complex conjugation. Remember, conjugation (represented by the superscript * symbol) is the reversal of the sign of a complex number's imaginary exponent—if $x = e^{j\varnothing}$, then, $x^* = e^{-j\varnothing}$. So, as a first step, we take the complex conjugate of both sides of Eq. (10-72) to give us

$$x^*(n) = \frac{1}{N}\left[\sum_{m=0}^{N-1} X(m)e^{j2\pi mn/N}\right]^* \tag{10-73}$$

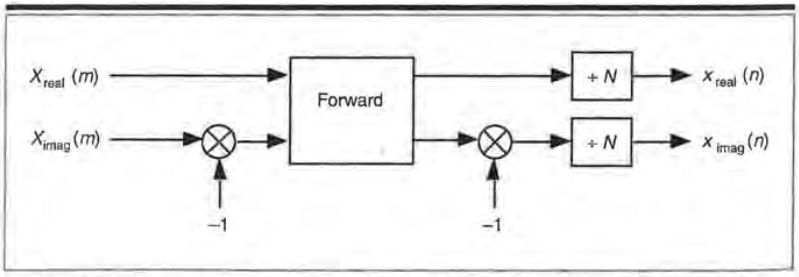


Figure 10-21 Processing diagram of first inverse FFT calculation method.

One of the properties of complex numbers, discussed in Appendix A, is that the conjugate of a product is equal to the product of the conjugates; that is, if $c = ab$, then $c^* = (ab)^* = a^*b^*$. Using this fact, we can show that the conjugate of the right side of Eq. (10-73) is given by

$$x^*(n) = \frac{1}{N}\sum_{m=0}^{N-1} X(m)^* \left(e^{j2\pi mn/N}\right)$$

$$= \frac{1}{N}\sum_{m=0}^{N-1} X(m)^* \, e^{-j2\pi mn/N} \tag{10-74}$$

Hold on, we're almost there. Notice the similarity of Eq. (10-74) to our original forward DFT expression Eq. (10-71). If we perform a forward DFT on the conjugate of the $X(m)$ in Eq. (10-74) and divide the results by N, we get the conjugate of our desired time samples $x(n)$. Taking the conjugate of both sides of Eq. (10-74), we get a more straightforward expression for $x(n)$:

$$x(n) = \frac{1}{N}\left[\sum_{m=0}^{N-1} X(m)^* \, e^{-j2\pi mn/N}\right]^* \tag{10-75}$$

So, to get the inverse FFT of a sequence $X(m)$ using the first inverse FFT algorithm,

Step 1: Conjugate the $X(m)$ input sequence.

Step 2: Calculate the forward FFT of the conjugated sequence.

Step 3: Conjugate the forward FFT's results.

Step 4: Divide each term of the conjugated results by N to get $x(n)$.

10.6.2 Second Inverse FFT Method

The second inverse FFT calculation technique is implemented following the interesting data flow shown in Figure 10-22. In this clever inverse FFT scheme, we don't bother with conjugation. Instead, we merely swap the real and imaginary parts of sequences of complex data[22]. To see why this process works, let's look at the inverse DFT equation again while separating the input $X(m)$ term into its real and imaginary parts and remembering that $e^{j\phi} = \cos(\phi) + j\sin(\phi)$:

$$\text{Inverse DFT} \rightarrow \quad x(n) = \frac{1}{N}\sum_{m=0}^{N-1} X(m)e^{j2\pi mn/N}$$

$$= \frac{1}{N}\sum_{m=0}^{N-1}[X_{\text{real}}(m) + jX_{\text{imag}}(m)]$$

$$\cdot[\cos(2\pi mn/N) + j\sin(2\pi mn/N)] \ . \qquad (10\text{-}76)$$

Multiplying the complex terms in Eq. (10-76) gives us

$$x(n) = \frac{1}{N}\sum_{m=0}^{N-1}[X_{\text{real}}(m)\cos(2\pi mn/N) - X_{\text{imag}}(m)\sin(2\pi mn/N)]$$

$$+ j[X_{\text{real}}(m)\sin(2\pi mn/N) + X_{\text{imag}}(m)\cos(2\pi mn/N)] \ . \qquad (10\text{-}77)$$

Equation (10-77) is the general expression for the inverse DFT, and we'll now quickly show that the process in Figure 10-22 implements this equation. With $X(m) = X_{\text{real}}(m) + jX_{\text{imag}}(m)$ and, swapping these terms,

$$X_{\text{swap}}(m) = X_{\text{imag}}(m) + jX_{\text{real}}(m) \ . \qquad (10\text{-}78)$$

The forward DFT of our $X_{\text{swap}}(m)$ is

$$\text{Forward DFT} \rightarrow \quad \sum_{n=0}^{N-1}[X_{\text{imag}}(m) + jX_{\text{real}}(m)]$$
$$\cdot[\cos(2\pi mn/N) - j\sin(2\pi mn/N)] \ . \qquad (10\text{-}79)$$

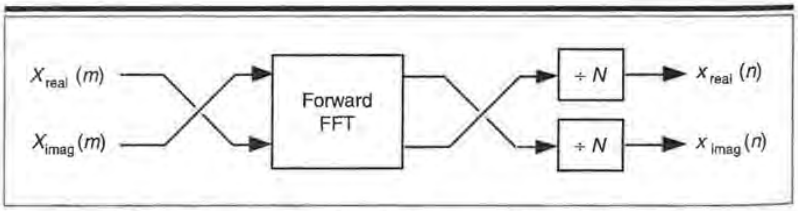


Figure 10-22 Processing diagram of second inverse FFT calculation method.

Multiplying the complex terms in Eq. (10-79) gives us

$$\text{Forward DFT} \rightarrow \sum_{n=0}^{N-1}[X_{\text{imag}}(m)\cos(2\pi mn/N) + X_{\text{real}}(m)\sin(2\pi mn/N)]$$

$$+ j[X_{\text{real}}(m)\cos(2\pi mn/N) - X_{\text{imag}}(m)\sin(2\pi mn/N)] \quad (10\text{-}80)$$

Swapping the real and imaginary parts of the results of this forward DFT gives us what we're after:

$$\text{Forward DFT}_{\text{swap}} \rightarrow \sum_{n=0}^{N-1}[X_{\text{real}}(m)\cos(2\pi mn/N) - X_{\text{imag}}(m)\sin(2\pi mn/N)]$$

$$+ j[X_{\text{imag}}(m)\cos(2\pi mn/N) + X_{\text{real}}(m)\sin(2\pi mn/N)] \quad (10\text{-}81)$$

If we divide Eq. (10-81) by N it would be equal to the inverse DFT expression in Eq. (10-77), and that's what we set out to show. To reiterate, we calculate the inverse FFT of a sequence $X(m)$ using this second inverse FFT algorithm in Figure 10-22:

Step 1: Swap the real and imaginary parts of the $X(m)$ input sequence.

Step 2: Calculate the forward FFT of the swapped sequence.

Step 3: Swap the real and imaginary parts of the forward FFT's results.

Step 4: Divide each term of the swapped sequence by N to get $x(n)$.

10.7 Fast FFT Averaging

Section 8.3 discussed the integration gain possible when averaging multiple FFT outputs to enhance signal-detection sensitivity. Well, there's a smart way to do this if we recall the linearity property of the DFT (which of course applies to the FFT) introduced in Section 3.3. If an input sequence $x_1(n)$ has an FFT of $X_1(m)$ and another input sequence $x_2(n)$ has an FFT of $X_2(m)$, then the FFT of the sum of these sequences $x_{\text{sum}}(n) = x_1(n) + x_2(n)$ is the sum of the individual FFTs, or

$$X_{sum}(m) = X_1(m) + X_2(m)\,. \tag{10-82}$$

So, if we want to average multiple FFT outputs, we can save considerable processing effort by averaging the individual FFT input sample sequences (frames), first, and then take a single FFT. Say, for example, that we wanted to average 20 FFTs to improve our FFT output signal-to-noise ratio. Instead of taking 20 FFTs of 20 frames of input signal data, we should average the 20 frames of input data, first, and then take a single FFT of that average. This avoids the number crunching necessary for 19 FFTs. By the way, for this technique to improve an FFT's signal-detection sensitivity, the original signal sampling must meet the criterion of coherent integration as described in Section 3.12.

10.8 Simplified FIR Filter Structure

If we need to implement an FIR digital filter using the standard structure in Figure 10-23(a), there's a way to simplify the necessary calculations when the filter has an odd number of taps. Let's look at the top of Figure 10-23(a) example where the 5-tap filter coefficients are $h(0)$ through $h(4)$ and the $y(n)$ output is given by

$$y(n) = h(4)x(n-4) + h(3)x(n-3) + h(2)x(n-2) + h(1)x(n-1) + h(0)x(n)\,. \tag{10-83}$$

If the FIR filter's coefficients are symmetrical, we can reduce the number of necessary multipliers; that is, if $h(4) = h(0)$, and $h(3) = h(1)$, we can implement Eq. (10-83) by

$$y(n) = h(4)[x(n-4)+x(n)] + h(3)[x(n-3)+x(n-1)] + h(2)\cdot x(n-2)\,, \tag{10-84}$$

where only three multiplications are necessary, as shown at the bottom of Figure 10-23(b). In our 5-tap filter case, we've eliminated two multipliers at the expense of implementing two additional adders.

In the general case of symmetrical-coefficient FIR filters with S taps, we can trade $(S-1)/2$ multipliers for $(S-1)/2$ adders when S is an odd number. So, in the case of an odd number of taps, we need perform only $(S-1)/2 + 1$ multiplications for each filter output sample. For an even number of symmetrical taps as shown in Figure 10-23(b), the saving afforded by this technique reduces the necessary number of multiplications to $S/2$. For the half-band filters discussed in Section 5.7, with their alternating zero-valued coefficients, the simplified FIR structure in Figure 10-23(b) allows us to get away with only $(S+1)/4 + 1$ multiplications for

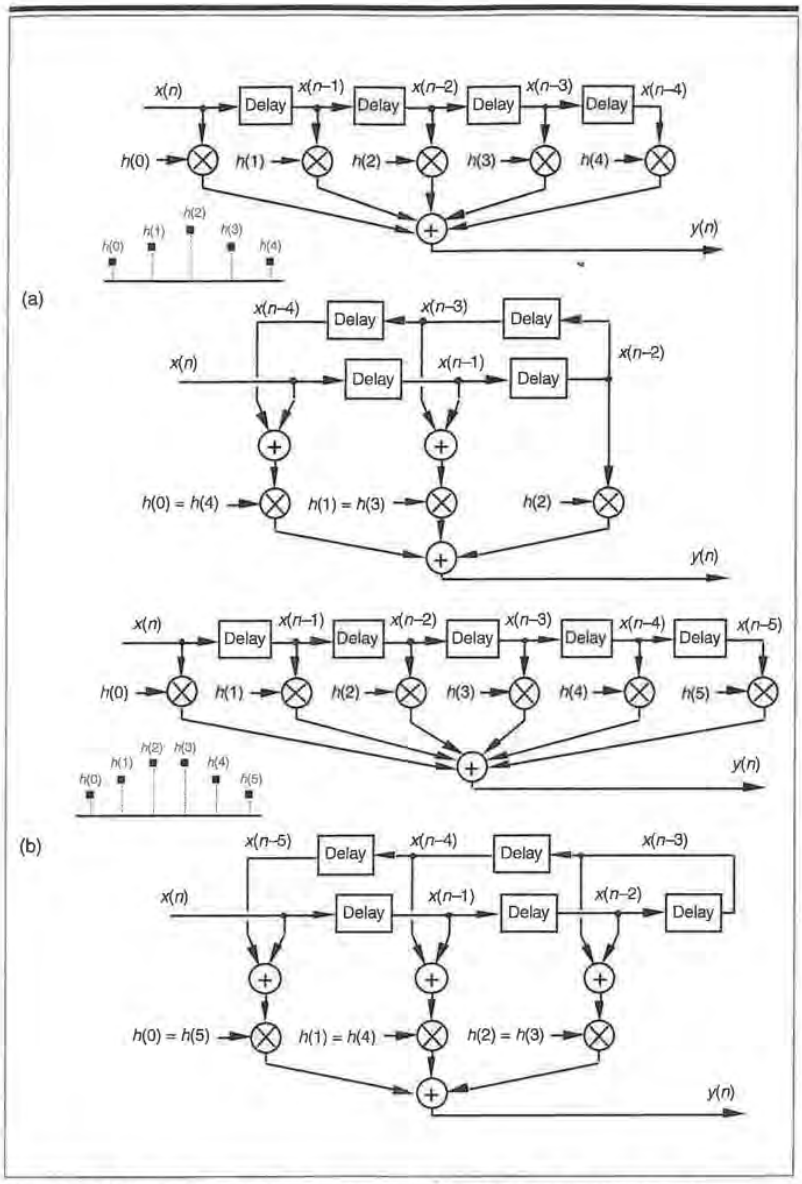


Figure 10-23 Simplified FIR filter implementations: (a) conventional and simplified, structures of an FIR filter with an odd number of taps; (b) conventional and simplified structures of an FIR filter with an even number of taps.

each filter output sample when S is odd and the first filter coefficient $h(0)$ is not zero.

We always benefit whenever we can exchange multipliers for adders. Because multiplication often takes a longer time to perform than addition, this symmetrical FIR filter simplification scheme may speed filter calculations performed in software. For a hardware FIR filter, this scheme can either reduce the number of necessary multiplier circuits or increase the effective number of taps for a given number of available hardware multipliers. Of course, whenever we increase the effective number of filter taps, we improve our filter performance for a given input signal sample rate.

10.9 Accurate A/D Converter Testing Technique

The manufacturers of A/D converters have recently begun to take advantage of digital signal processing techniques to facilitate the testing of their products. A traditional test method involves applying a sinusoidal analog voltage to an A/D converter and using the FFT to obtain spectrum of the digitized samples. Converter dynamic range, missing bits, harmonic distortion, and other nonlinearities can be characterized by analyzing the spectral content of the converter output. These nonlinearities are easy to recognize because they show up as spurious spectral components and increased background noise levels in the FFT spectra.

To enhance the accuracy of the spectral measurements, window functions were originally used on the time-domain converter output samples to reduce the spectral leakage inherent in the FFT. This was fine until the advent of 12- and 14-bit A/D converters. These converters have dynamic ranges so large that their small nonlinearities, evident in their spectra, were being swamped by the sidelobe levels of even the best window functions. (From Figure 9-4 we know that a 14-bit A/D converter can have an SNR ratio of well over 80 dB.) The clever technique that circumvents this problem is to use an analog sinusoidal input voltage whose frequency is an integral fraction of the A/D converter's sample frequency as shown in Figure 10-24(a). That frequency is mf_s/N where m is an integer, f_s is the sample frequency, and N is the FFT size. Figure 10-24(a) shows the $x(n)$ time-domain output of an ideal A/D converter under the condition that its analog input is a sinewave having exactly eight cycles over 128 output samples. In this case, the input frequency normalized to the sample rate f_s is $8f_s/128$ Hz. Recall, from Chapter 3, that the expression mf_s/N defined the analysis frequencies, or bin centers, of the DFT; and a DFT input whose frequency is at a bin center results in no leakage even without the use of a window function. Another way to look at this situation is to real-

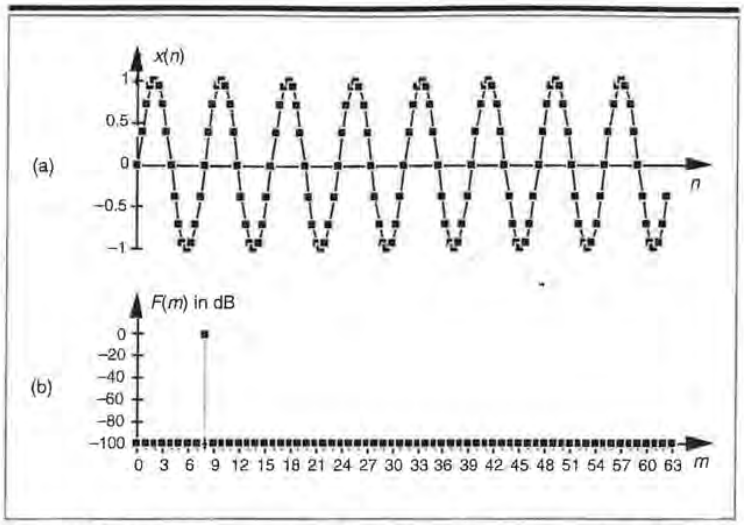


Figure 10-24 Ideal A/D converter output whose input is an analog $8f_s/128$ Hz sinusoid: (a) time-domain samples; (b) frequency-domain spectrum in dB.

ize that the analog mf_s/N frequency sinusoid will have exactly m complete cycles over the N FFT input samples, as indicated by Figure 3-7(b) in Chapter 3.

The first half of a 128-point FFT of $x(n)$ is shown in the logarithmic plot in Figure 10-24(b) where the input tone lies exactly at the $m = 8$ bin center, and DFT leakage has been avoided altogether. Specifically, if the sample rate were 1 MHz, then the A/D's input analog tone would have to be exactly $8 \cdot 10^6/128 = 62.5$ kHz. To implement this scheme, we need to ensure that the analog test generator be synchronized, exactly, with the A/D converter's clock frequency of f_s Hz. Achieving this synchronization is why this A/D converter testing procedure is referred to as *coherent sampling*[23–25]. The analog signal generator and the A/D clock generator providing f_s must not drift in frequency relative to each other—they must remain coherent. (We must take care here from a semantic viewpoint because the quadrature sampling schemes described in Sections 7.1 and 7.2 are also sometimes called coherent sampling, but they are unrelated to this A/D converter testing procedure.)

As it turns out, some values of m are more advantageous than others. Notice in Figure 10-24(a), when $m = 8$, only nine different amplitude values are put out by the A/D converter. Those values are

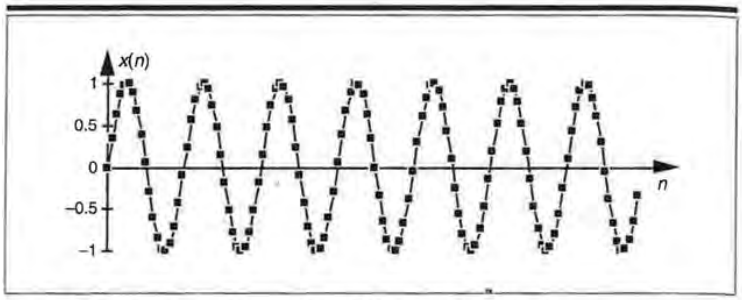


Figure 10-25 Seven-cycle sinusoidal A/D converter output.

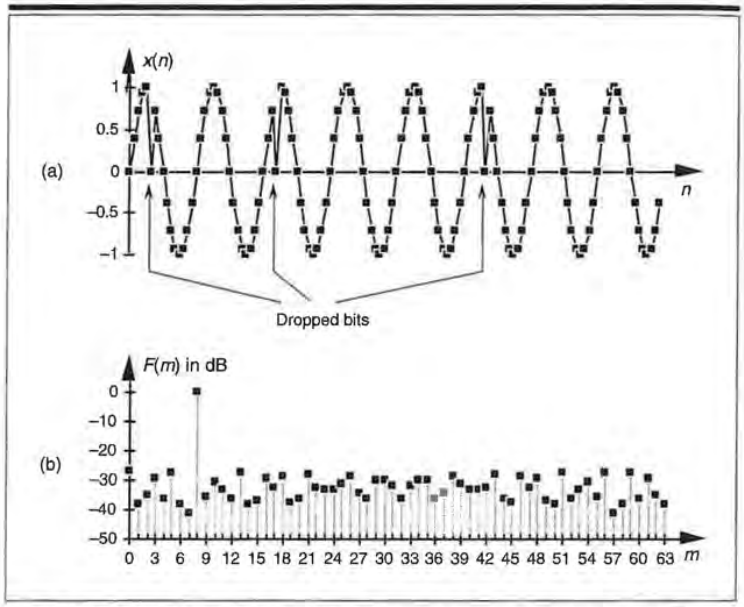


Figure 10-26 Nonideal A/D converter output showing several dropped bits: (a) time-domain samples; (b) frequency-domain spectrum in dB.

repeated over and over. As shown in Figure 10-25, when $m = 7$, we exercise many more than nine different A/D output values. Because it's best to test as many A/D output binary words as possible, in practice, users of this A/D testing scheme have found that making m an odd prime number (3, 5, 7, 11, etc.) minimizes the number of redundant A/D output word values.

Figure 10-26(a) illustrates an extreme example of nonlinear A/D converter operation with several discrete output samples having dropped bits in the time domain $x(n)$ with $m = 8$. The FFT of this distorted $x(n)$ is shown in Figure 10-26(b) where we can see the greatly increased background noise level due to the A/D converter's nonlinearities compared to Figure 10-24(b).

To fully characterize the dynamic performance of an A/D converter, we'd need to perform this testing technique at many different input frequencies and amplitudes.[†] In addition, applying two analog tones to the A/D converter's input is often done to quantify the intermodulation distortion performance of a converter, which, in turn, characterizes the converter's dynamic range. In doing so, both input tones must comply with the mf_s/N restriction. The key issue here is that, when any input frequency is mf_s/N, we can take full advantage of the FFT's processing sensitivity while completely avoiding spectral leakage.

10.10 Fast FIR Filtering Using the FFT

While contemplating the convolution relationships in Eq. (5-31) and Figure 5-41, digital signal processing practitioners realized that convolution could sometimes be performed more efficiently using FFT algorithms than it could be using the direct convolution method[26,27]. This FFT-based convolution scheme, called *fast convolution*, is diagrammed in Figure 10-27. The standard convolution equation, for an M-tap FIR filter, given in Eq. (5-6) is repeated here for reference as

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k) = h(k) * x(n) \quad . \tag{10-85}$$

where $h(k)$ is the impulse response sequence (coefficients) of the FIR filter and the "*" symbol indicates convolution. It has been shown that, when the final $y(n)$ output sequence has a length greater than 30, the process in Figure 10-27 requires fewer multiplications than implementing the convolution expression in Eq. (10-85) directly. Consequently, this fast convolution technique is a very powerful signal processing tool, particularly, when used for

[†] The analog sinewave applied to an A/D converter must, of course, be as *pure* as possible. Any distortion inherent in the analog signal will show up in the final FFT output and could be mistaken for A/D nonlinearity.

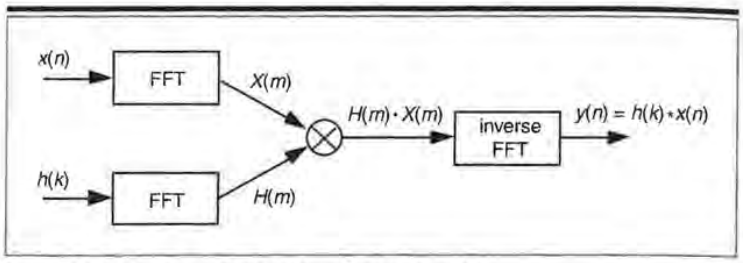


Figure 10-27 Processing diagram of fast convolution.

digital filtering. Very efficient FIR filters can be designed using this technique because, if their impulse response $h(k)$ is constant, then we don't have to bother recalculating $H(m)$ each time a new $x(n)$ sequence is filtered. In this case, the $H(m)$ sequence can be precalculated and stored in memory.

The necessary forward and inverse FFT sizes must, of course, be equal and are dependent on the length of the original $h(k)$ and $x(n)$ sequences. Recall from Eq. (5-29) that, if $h(k)$ is of length P and $x(n)$ is of length Q, the length of the final $y(n)$ sequence will be $(P+Q-1)$. For valid results from this fast convolution technique, the forward and inverse FFT sizes must be equal and greater than $(P+Q-1)$. This means that $h(k)$ and $x(n)$ must both be padded (or stuffed) with zero-valued samples at the end of their respective sequences, to make their lengths identical and greater than $(P+Q-1)$. This zero padding will not invalidate the fast convolution results. So, to use fast convolution, we must choose an N-point FFT size such that $N \geq (P+Q-1)$ and zero pad $h(k)$ and $x(n)$, so that they have new lengths equal to N.

An interesting aspect of fast convolution, from a hardware standpoint, is that the FFT indexing bit-reversal problem discussed in Sections 4.5 and 4.6 is not an issue here. If the identical FFT structures used in Figure 10-27 result in $X(m)$ and $H(m)$ having bit-reversed indices, the multiplication can still be performed directly on the scrambled $H(m)$ and $X(m)$ sequences. Then, an appropriate inverse FFT structure can be used that expects bit-reversed input data. That inverse FFT then provides an output $y(n)$ whose data index is in the correct order!

10.11 Calculation of Sines and Cosines of Consecutive Angles

There are times in digital signal processing when we need our software to calculate lots of sine and cosine values, particularly in implementing certain FFT algorithms[28,29]. Because trigonometric calculations are time consuming to perform, a clever idea has been used to calculate the sines

and cosines of consecutive angles without having to actually call upon standard trigonometric functions.† Illustrating this scheme by way of example, let's say we want to calculate the sines and cosines of all angles from 0° to 90° in 1 degree increments. Instead of performing those 91 sine and 91 cosine trigonometric operations, we can use the following identities found in our trusty math reference book,

$$\sin(A+B) = \sin(A)\cdot\cos(B) + \cos(A)\cdot\sin(B) \tag{10-86}$$

and

$$\cos(A+B) = \cos(A)\cdot\cos(B) - \sin(A)\cdot\sin(B)\,, \tag{10-87}$$

to reduce our computational burden. To see how, lets make $A = \alpha$ and $B = n\alpha$, where $\alpha = 1°$ and n is an integral index $0 \le n \le 89$. Equations (10-86) and (10-87) now become

$$\sin(\alpha + n\alpha) = \sin(\alpha\,[1+n]) = \sin(\alpha)\cdot\cos(n\alpha) + \cos(\alpha)\cdot\sin(n\alpha)\,, \tag{10-88}$$

and

$$\cos(\alpha + n\alpha) = \cos(\alpha\,[1+n]) = \cos(\alpha)\cdot\cos(n\alpha) - \sin(\alpha\cdot\sin(n\alpha)\,. \tag{10-89}$$

OK, here's how we calculate the sines and cosines. First, we know the sine and cosine for the first angle of 0°; i.e., sin(0°) = 0, and cosine(0°) = 1. Next we need to use a standard trigonometric function call to calculate and store, for later use, the sine and cosine of 1°; that is, sin(1°) = 0.017452 and cos(1°) =0.999848. Now we're ready to calculate the sine and cosine of 2° by substituting $n = 1$ and $\alpha = 1°$ into Eqs. (10-88) and (10-89) giving us

$$\sin(1°[1+1]) = \sin(1°)\cdot\cos(1\cdot 1°) + \cos(1°)\cdot\sin(1\cdot 1°),$$

or

$$\sin(2°) = (0.017452)\cdot(0.999848) + (0.999848)\cdot(0.017452) = 0.034899, \tag{10-90}$$

and

$$\cos(1°[1+1]) = \cos(1°)\cdot\cos(1\cdot 1°) - \sin(1°)\cdot\sin(1\cdot 1°),$$

or

$$\cos(2°) = (0.999848)\cdot(0.999848) + (0.017452)\cdot(0.017452) = 0.999391. \tag{10-91}$$

† We're assuming here that there's insufficient memory space available to store all the required sine and cosine values for later recall.

Because we've already calculated sin(1°) and cos(1°), Eqs. (10-90) and (10-91) each required only two multiplies and an add. No trigonometric function needed to be called by software.

Next we're able to calculate the sine and cosine of 3° by substituting $n = 2$ and $\alpha = 1°$ in Eqs. (10-88) and (10-89); that is,

$$\sin(1°[1+2]) = \sin(1°) \cdot \cos(2 \cdot 1°) + \cos(1°) \cdot \sin(2 \cdot 1°),$$

or

$$\sin(3°) = (0.017452) \cdot (0.999391) + (0.999848) \cdot (0.034899) = 0.052336, \quad (10\text{-}92)$$

and

$$\cos(1°[1+2]) = \cos(1°) \cdot \cos(2 \cdot 1°) - \sin(1°) \cdot \sin(2 \cdot 1°),$$

or

$$\cos(3°) = (0.999848) \cdot (0.999391) + (0.017452) \cdot (0.034899) = 0.998629. \quad (10\text{-}93)$$

Again, because we previously calculated sin(2°) and cos(2°), Eqs. (10-92) and (10-93) only require us to perform four multiplications and two additions. The pattern of our calculations is clear now. For successive angles, we merely use the sine and cosine of 1° and the sine and cosine values obtained during the previous angle calculation. In our example, the angle increment was $\alpha = 1°$, but it's good to know that Eqs. (10-88) and (10-89) apply for any fixed angle increment.

10.12 Generating Normally Distributed Random Data

Section D.4 in Appendix D discusses the normal distribution curve as it relates to random data. A problem we may encounter is how to actually generate random data samples whose distribution follows that normal curve. There's a slick way to solve this problem using any software package that can generate uniformly distributed random data, as most of them do[30]. Figure 10-28 shows our situation pictorially where we require random data that's distributed normally with a mean (average) of μ' and a standard deviation of σ', as in Figure 10-28(a), and all we have available is a software routine that generates random data that's uniformly distributed between zero and one as in Figure 10-28(b). As it turns out, there's a principle in advanced probability theory, known as the *Central Limit Theorem*, that says, when random data from an arbitrary distribution is summed over M samples, the probability distribu-

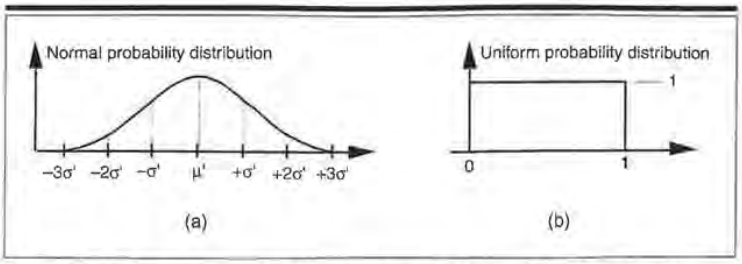


Figure 10-28 Probability distribution functions: (a) normal distribution with mean = μ', and standard deviation σ'; (b) uniform distribution between zero and one.

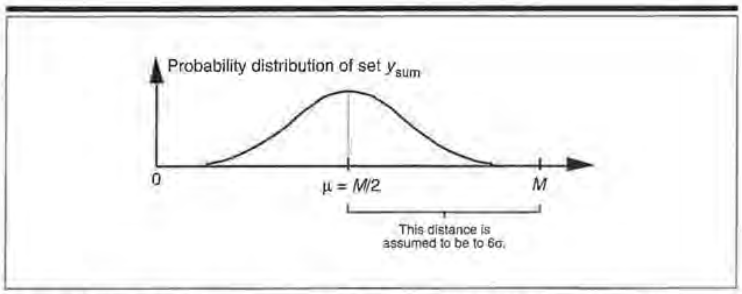


Figure 10-29 Probability distribution of the summed set of random data derived from uniformly distributed data.

tion of the sum begins to approach a normal distribution, as M increases[31,32]. In other words, if we generate a set of N random samples that are uniformly distributed between zero and one, we can begin adding other sets of N samples to the first set. As we continue summing additional sets, the distribution of the N-element set of sums becomes more and more *normal*. We can sound impressive and state that "the sum becomes asymptotically normal." Experience has shown that, for practical purposes, if we sum $M \geq 30$ times, the summed data distribution is essentially normal. With this rule in mind, we're halfway to solving our problem.

After summing M sets of uniformly distributed samples, the summed set y_{sum} will have a distribution like that shown in Figure 10-29. Because we've summed M sets, the mean of y_{sum} is $\mu = M/2$. To determine y_{sum}'s standard deviation σ, we assume that the six sigma point is equal to $M-\mu$; that is

$$6\sigma = M - \mu\,. \tag{10-94}$$

That assumption is valid because we know that the probability of an element in y_{sum} being greater than M is zero, and the probability of having a normal data sample at six sigma is one change in 6 billion, or essentially zero. Because $\mu = M/2$, then from Eq. (10-94), y_{sum}'s standard deviation is set to

$$\sigma = \frac{M-\mu}{6} = \frac{M-M/2}{6} = M/12 \ . \tag{10-95}$$

To convert the y_{sum} data set to our desired data set having a mean of μ' and a standard deviation of σ',

- subtract $M/2$ from each element of y_{sum} to shift its mean to zero,
- next, ensure that $6\sigma'$ is equal to $M/2$ by multiplying each element in the shifted data set by $12\sigma'/M$, and
- finally, center the new data set about the desired μ' by adding μ' to each element of the new data.

The steps in our algorithm are shown in Figure 10-30. If we call our desired normally distributed random data set $y_{desired}$, then the nth element of that set is described mathematically as

$$y_{desired}(n) = \frac{12\sigma'}{M} \cdot \left[\left(\sum_{k=1}^{M} x_k(n) \right) - \frac{M}{2} \right] + \mu' \ . \tag{10-96}$$

Our discussion thus far has had a decidedly software algorithmic flavor, but hardware designers also occasionally need to generate normally distributed (Gaussian) random data at high speeds in their designs. For your hardware designers, reference [33] presents an efficient hardware design technique to generate normally distributed random data using fixed-point arithmetic integrated circuits.

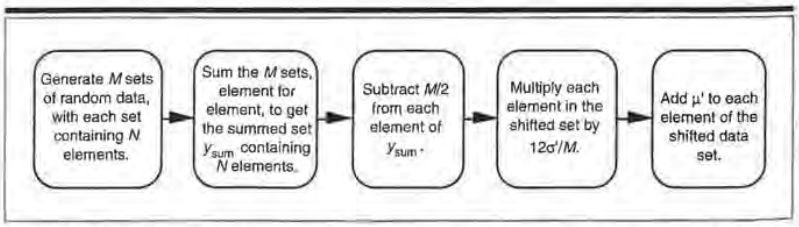


Figure 10-30 Processing steps required to generate normally distributed random data from uniformly distributed data.

References

[1] Freeny, S. "TDM/FDM Translation as an Application of Digital Signal Processing," *IEEE Communications Magazine*, January 1980.

[2] Considine, V. "Digital Complex Sampling," *Electronics Letters*, 19, 4 August 1983.

[3] Harris Semiconductor Corp. "A Digital, 16-Bit, 52 Msps Halfband Filter," *Microwave Journal*, September 1993.

[4] Hack, T. "IQ Sampling Yields Flexible Demodulators," *RF Design Magazine*, April 1991.

[5] Pellon, L. E. "A Double Nyquist Digital Product detector for Quadrature Sampling," *IEEE Trans. on Signal Processing*, Vol. 40, No. 7, July 1992.

[6] Waters, W. M., and Jarrett, B. R. "Bandpass Signal Sampling and Coherent Detection," *IEEE Trans. on Aerospace and Electronic Systems*, Vol. AES-18, No. 4, November 1982.

[7] Palacherls, A. "DSP-mP Routine Computes Magnitude," *EDN*, October 26, 1989.

[8] Mikami, N., Kobayashi, M., and Yokoyama, Y. "A New DSP-Oriented Algorithm for Calculation of the Square Root Using a Nonlinear Digital Filter," *IEEE Trans. on Signal Processing*, Vol. 40, No. 7, July 1992.

[9] Lyons, R. G. "Turbocharge Your Graphics Algorithm," *ESD: The Electronic System Design Magazine*, October 1988.

[10] Adams, W. T., and Brady, J. "Magnitude Approximations for Microprocessor Implementation," *IEEE Micro*, Vol. 3, No. 5, October 1983.

[11] Eldon, J. "Digital Correlator Defends Signal Integrity with Multibit Precision," *Electronic Design*, May 17, 1984.

[12] Smith, W. W. "DSP Adds Performance to Pulse Compression Radar," *DSP Applications*, October 1993.

[13] Harris Semiconductor Corp. HSP50110 Digital Quadrature Tuner Data Sheet, File Number 3651, February 1994.

[14] Bingham, C., Godfrey, M., and Tukey, J. "Modern Techniques for Power Spectrum Estimation," *IEEE Trans. on Audio and Electroacoust.*, Vol. AU-15, No. 2, June 1967.

[15] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*, Vol. 66, No. 1, January 1978.

[16] Nuttall, A. H. "Some Windows with Very Good Sidelobe Behavior," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-29, No. 1, February 1981.

[17] Cox, R. "Complex-Multiply Code Saves Clocks Cycles," *EDN*, June 25, 1987.

[18] Rabiner, L. R., and Gold, B. *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975, pp. 356.

[19] Sorenson, H. V., Jones, D. L., Heideman, M. T., and Burrus, C. S. "Real-Valued Fast Fourier Transform Algorithms," *IEEE Trans. on Acoust. Speech, and Signal Proc.*, Vol. ASSP-35, No. 6, June 1987.

[20] Cooley, J. W., Lewis, P. A., and Welch, P. D. "The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of Sine, Cosine and Laplace Transforms," *Journal Sound Vib.*, Vol. 12, July 1970.

[21] Brigham, E. O. *The Fast Fourier Transform and Its Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974, pp. 167.

[22] Burrus, C. S., et al. *Computer-Based Exercises for Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1994, pp. 53.

[23] Coleman, B., Meehan, P., Reidy, J., and Weeks, P. "Coherent Sampling Helps When Specifying DSP A/D Converters," *EDN*, October 1987.

[24] Ushani, R. "Classical Tests Are Inadequate for Modern High-Speed Converters," *EDN Magazine*, May 9, 1991.

[25] Meehan, P., and Reidy, J. "FFT Techniques Give Birth to Digital Spectrum Analyzer," *Electronic Design*, August 11, 1988, pp. 120.

[26] Stockham, T. G. "High-speed Convolution and Correlation," in *Digital Signal Processing*, Ed. by L. Rabiner and C. Rader, IEEE Press, New Jersey, 1972, pp. 330.

[27] Stockham, T. G. "High-Speed Convolution and Correlation with Applications to Digital Filtering," Chapter 7 in *Digital Processing of Signals*, by B. Gold et al., McGraw-Hill, New York, 1969, pp. 203.

[28] Dobbe, J. G. G. "Faster FFTs," *Dr. Dobb's Journal*, February 1995, pp. 125. (Be careful here. The last equation in Example 5 is incorrect on page 133 of this reference, so be sure and use our Eq. (10-86) above.)

[29] Crenshaw, J. W. "All About Fourier Analysis," *Embedded Systems Programming*, October 1994, pp. 70.

[30] Beadle, E. "Algorithm Converts Random Variables to Normal," *EDN Magazine*, May 11, 1995.

[31] Spiegel, M. R. *Theory and Problems of Statistics*, Shaum's Outline Series, McGraw-Hill Book Co., New York, 1961, pp. 142.

[32] Davenport, W. B., Jr., and Root, W. L. *Random Signals and Noise*, McGraw-Hill Book Co., New York, 1958, pp. 81.

[33] Salibrici, B. "Fixed-point DSP Chip Can Generate Real-time Random Noise," *EDN Magazine*, April 29, 1993.

APPENDIX A

The Arithmetic of Complex Numbers

To understand digital signal processing, we have to get comfortable using complex numbers. The first step toward this goal is learning to manipulate complex numbers arithmetically. Fortunately, we can take advantage of our knowledge of real numbers to make this job easier. Although the physical significance of complex numbers is discussed in Appendix C, the following discussion provides the arithmetic rules governing complex numbers.

A.1 Graphical Representation of Real and Complex Numbers

To get started, real numbers are those positive or negative numbers we're used to thinking about in our daily lives. Examples of real numbers are 0.3, –2.2, 5.1, etc. Keeping this in mind, we see how a real number can be represented by a point on a one-dimensional axis, called the *real* axis, as shown in Figure A-1.

We can, in fact, consider that all real numbers correspond to all of the points on the real axis line on a one-to-one basis.

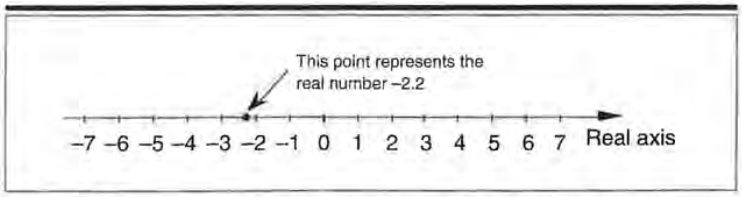


Figure A-1 The representation of a real number as a point on the one-dimensional real axis.

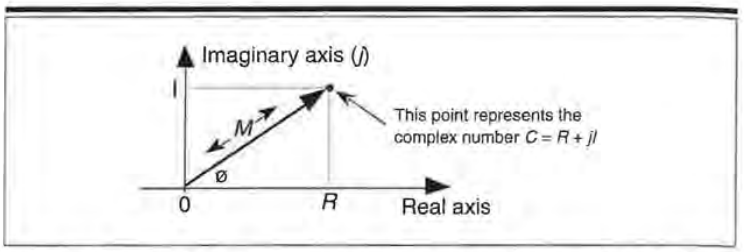


Figure A-2 The phasor representation of the complex number $C = R + jI$ on the complex plane.

A complex number, unlike a real number, has two parts: a real part and an imaginary part. Just as a real number can be considered to be a point on the one-dimensional real axis, a complex number can be treated as a point on a complex plane as shown in Figure A-2. We'll use this geometrical concept to help us understand the arithmetic of complex numbers.†

A.2 Arithmetic Representation of Complex Numbers

A complex number C is represented in a number of different ways in the literature, such as

Rectangular form: $\rightarrow$ $$C = R + jI\,, \tag{A-1}$$

Trigonometric form: $\rightarrow$ $$C = M[\cos(ø) + j\sin(ø)]\,, \tag{A-1'}$$

Exponential form: $\rightarrow$ $$C = M\,e^{jø}\,, \tag{A-1''}$$

Magnitude and angle form: $\rightarrow$ $$C = M \angle ø\,. \tag{A-1'''}$$

Equations (A-1'') and (A-1''') remind us that the complex number C can also be considered the tip of a phasor on the complex plane, with magnitude M, in the direction of ø degrees relative to the positive real axis as shown in Figure A-2. (We'll avoid calling phasor M a *vector* because the term *vector* means different things in different contexts. In linear algebra, *vector* is the term used to signify a one-dimensional matrix. On the other hand, in mechanical engineering and field theory, vectors are used to signify magnitudes and directions, but there are vector operations (*scalar* or

† The complex plane representation of a complex number is sometimes called an Argand diagram—named after the French mathematician Jean Robert Argand (1768–1825).

dot product, and *vector* or *cross-product*) that don't apply to our definition of a phasor.) The relationships between the variables in this figure follow the standard trigonometry of right triangles. Keep in mind that C is a complex number, and the variables R, I, M, and $\varnothing$ are all real numbers. The magnitude of C, sometimes called the *modulus* of C, is

$$M = |C| = \sqrt{R^2 + I^2} \ , \tag{A-2}$$

and, by definition, the phase angle, or *argument*, of C is the arctangent of I/R, or

$$\varnothing = \tan^{-1}\left(\frac{I}{R}\right) . \tag{A-3}$$

The variable $\varnothing$ in Eq. (A-3) is a general angle term. It can have dimensions of degrees or radians. Of course, we can convert back and forth between degrees and radians using π radians = 180°. So, if $\varnothing_r$ is in radians and $\varnothing_d$ is in degrees, then we can convert $\varnothing_r$ to degrees by the expression

$$\varnothing_d = \frac{180\varnothing_r}{\pi} \ . \tag{A-4}$$

Likewise, we can convert $\varnothing_d$ to radians by the expression

$$\varnothing_r = \frac{\pi\varnothing_d}{180} \ . \tag{A-5}$$

The exponential form of a complex number has an interesting characteristic that we need to keep in mind. Whereas only a single expression in rectangular form can describe a single complex number, an infinite number of exponential expressions can describe a single complex number; that is, while, in the exponential form, a complex number C can be represented by $C = Me^{j\varnothing}$, it can also be represented by

$$C = Me^{j\varnothing} = Me^{j(\varnothing + 2\pi n)} , \tag{A-6}$$

where $n = \pm 1, \pm 2, \pm 3, \ldots$ and $\varnothing$ is in radians. When $\varnothing$ is in degrees, Eq. (A-6) is in the form

$$C = Me^{j\varnothing} = Me^{j(\varnothing + n360^\circ)} . \tag{A-7}$$

Equations (A-6) and (A-7) are *almost* self-explanatory. They indicate that the point on the complex plane represented by the tip of the phasor C remains unchanged if we rotate the phasor some integral multiple of 2π radians or an integral multiple of 360°. So, for example, if $C = Me^{j(20°)}$, then

$$C = Me^{j(20°)} = Me^{j(380°)} = Me^{j(740°)}. \quad \text{(A-8)}$$

The variable ø, the angle of the phasor in Figure A-2, need not be constant. We'll often encounter expressions containing a complex sinusoid that takes the form

$$C = Me^{j\omega t}. \quad \text{(A-9)}$$

Equation (A-9) represents a phasor of magnitude M whose angle in Figure A-2 is increasing linearly with time at a rate of ω radians each second. If $\omega = 2\pi$, the phasor described by Eq. (A-9) is rotating counterclockwise at a rate of 2π radians per second—one revolution per second—and that's why ω is called the radian frequency. In terms of frequency, Eq. (A-9)'s phasor is rotating counterclockwise at $\omega = 2\pi f$ radians per second, where f is the cyclic frequency in cycles per second (Hz). If the cyclic frequency is $f = 10$ Hz, the phasor is rotating 20π radians per second. Likewise, the expression

$$C = Me^{-j\omega t} \quad \text{(A-9')}$$

represents a phasor of magnitude M that rotates in a clockwise direction about the origin of the complex plane at a negative radian frequency of $-\omega$ radians per second.

A.3 Arithmetic Operations of Complex Numbers

A.3.1 Addition and Subtraction of Complex Numbers

Which of the above forms for C in Eq. (A-1) is the best to use? It depends on the arithmetic operation we want to perform. For example, if we're adding two complex numbers, the rectangular form in Eq. (A-1) is the easiest to use. The addition of two complex numbers, $C_1 = R_1 + jI_1$ and $C_2 = R_2 + jI_2$, is merely the sum of the real parts plus j times the sum of the imaginary parts as

$$C_1 + C_2 = R_1 + jI_1 + R_2 + jI_2 = R_1 + R_2 + j(I_1 + I_2). \quad \text{(A-10)}$$

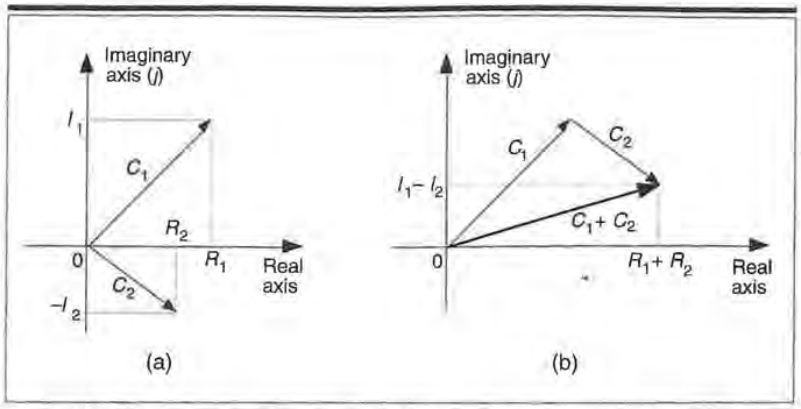


Figure A-3 Geometrical representation of the sum of two complex numbers.

Figure A-3 is a graphical depiction of the sum of two complex numbers using the concept of phasors. Here the sum phasor $C_1 + C_2$ in Figure A-3(a) is the new phasor from the beginning of phasor C_1 to the end of phasor C_2 in Figure A-3(b). Remember, the *R*s and the *I*s can be either positive or negative numbers. Subtracting one complex number from the other is straightforward as long as we find the differences between the two real parts and the two imaginary parts separately. Thus

$$C_1 - C_2 = (R_1 + jI_1) - (R_2 + jI_2) = R_1 - R_2 + j(I_1 - I_2)\,. \qquad \text{(A-11)}$$

An example of complex number addition is discussed in Section 8.3 where we covered the topic of averaging fast Fourier transform outputs.

A.3.2 Multiplication of Complex Numbers

We can use the rectangular form to multiply two complex numbers as

$$C_1C_2 = (R_1 + jI_1)\,(R_2 + jI_2) = (R_1R_2 - I_1\,I_2) + \mathrm{j}(R_1I_2 + R_2\,I_1)\,. \qquad \text{(A-12)}$$

However, if we represent the two complex numbers in exponential form, their product takes the simpler form

$$C_1C_2 = M_1e^{j\varnothing_1}\,M_2e^{j\varnothing_2} = M_1M_2e^{j(\varnothing_1+\varnothing_2)} \qquad \text{(A-13)}$$

because multiplication results in the addition of the exponents.

As a special case of multiplication of two complex numbers, scaling is multiplying a complex number by another complex number whose imaginary part is zero. We can use the rectangular or exponential forms with equal ease as follows:

$$kC = k(R + jI) = kR + jkI\,, \tag{A-14}$$

or in exponential form,

$$kC = k(Me^{j\varnothing}) = kMe^{j\varnothing}\,. \tag{A-15}$$

A.3.3 Conjugation of a Complex Number

The complex conjugate of a complex number is obtained by merely changing the sign of the number's imaginary part. So, if we denote C^* as the complex conjugate of the number $C = R + jI = Me^{j\varnothing}$, then, C^* is expressed as

$$C^* = R - jI = Me^{-j\varnothing}\,. \tag{A-16}$$

There are two characteristics of conjugates that occasionally come in handy. First, the conjugate of a product is equal to the product of the conjugates. That is, if $C = C_1C_2$, then from Eq. (A-13),

$$C^* = (C_1C_2)^* = (M_1M_2e^{j(\varnothing_1+\varnothing_2)})^* = M_1M_2e^{-j(\varnothing_1+\varnothing_2)}$$

$$= M_1e^{-j\varnothing_1}M_2e^{-j\varnothing_2} = C_1^*C_2^*\,. \tag{A-17}$$

Second, the product of a complex number and its conjugate is the complex number's magnitude squared. It's easy to show this in exponential form as

$$CC^* = Me^{j\varnothing} \cdot Me^{-j\varnothing} = M^2e^{j0} = M^2\,. \tag{A-18}$$

(This property is often used in digital signal processing to determine the relative power of a complex sinusoidal phasor represented by $Me^{j\omega t}$.)

A.3.4 Division of Complex Numbers

The division of two complex numbers is also convenient using the exponential and magnitude and angle forms, such as

$$\frac{C_1}{C_2} = \frac{M_1 e^{j\varnothing_1}}{M_2 e^{j\varnothing_2}} = \frac{M_1}{M_2} e^{j(\varnothing_1 - \varnothing_2)}, \tag{A-19}$$

and

$$\frac{C_1}{C_2} = \frac{M_1}{M_2} \angle \varnothing_1 - \varnothing_2 . \tag{A-19'}$$

Although not nearly so handy, we can perform complex division in rectangular notation by multiplying the numerator and the denominator by the complex conjugate of the denominator as

$$\frac{C_1}{C_2} = \frac{R_1 + jI_1}{R_2 + jI_2}$$

$$= \frac{R_1 + jI_1}{R_2 + jI_2} \cdot \frac{R_2 - jI_2}{R_2 - jI_2}$$

$$= \frac{(R_1R_2 + jI_1I_2) + j(R_2I_1 - R_1I_2)}{R_2^2 + I_2^2} . \tag{A-20}$$

A.3.5 Inverse of a Complex Number

A special form of division is the inverse, or reciprocal, of a complex number. If $C = Me^{j\varnothing}$, its inverse is given by

$$\frac{1}{C} = \frac{1}{Me^{j\varnothing}} = \frac{1}{M} e^{-j\varnothing} . \tag{A-21}$$

In rectangular form, the inverse of $C = R + jI$ is given by

$$\frac{1}{C} = \frac{1}{R + jI} = \frac{R - jI}{R^2 + I^2} . \tag{A-22}$$

We get Eq. (A-22) by substituting $R_1 = 1$, $I_1 = 0$, $R_2 = R$, and $I_2 = I$ in Eq. (A-20).

A.3.6 Complex Numbers Raised to a Power

Raising a complex number to some power is easily done in the exponential form. If $C = Me^{j\varnothing}$, then

$$C^k = M^k (e^{j\varnothing})^k = M^k e^{jk\varnothing}. \tag{A-23}$$

For example, if $C = 3e^{j125°}$, then C cubed is

$$(C)^3 = 3^3(e^{j3\cdot125°}) = 27e^{j375°} = 27e^{j15°}. \tag{A-24}$$

We conclude this appendix with four final complex arithmetic operations that are not very common in digital signal processing—but you may need them sometime.

A.3.7 Roots of a Complex Number

The kth root of a complex number C is the number that, multiplied by itself k times, results in C. The exponential form of C is the best way to explore this process. When a complex number is represented by $C = Me^{j\varnothing}$, remember that it can also be represented by

$$C = Me^{j(\varnothing + n360°)}. \tag{A-25}$$

In this case, the variable $\varnothing$ in Eq. (A-25) is in degrees. There are k distinct roots when we're finding the kth root of C. By distinct, we mean roots whose exponents are less than 360°. We find those roots by using the following:

$$\sqrt[k]{C} = \sqrt[k]{Me^{j(\varnothing+n360°)}} = \sqrt[k]{M}e^{j(\varnothing+n360°)/k}. \tag{A-26}$$

Next, we assign the values 0, 1, 2, 3, . . ., k–1 to n in Eq. (A-26) to get the k roots of C. OK, we need an example here! Let's say we're looking for the cube (3rd) root of $C = 125e^{j(75°)}$. We proceed as follows:

$$\sqrt[3]{C} = \sqrt[3]{125e^{j(75°)}} = \sqrt[3]{125e^{j(75°+n360°)}} = \sqrt[3]{125}e^{j(75°+n360°)/3}. \tag{A-27}$$

Next we assign the values $n = 0$, $n = 1$, and $n = 2$ to Eq. (A-27) to get the three roots of C. So the three distinct roots are

1st root → $\sqrt[3]{C} = 5e^{j(75°+0\cdot360°)/3} = 5e^{j(25°)}$;

2nd root → $\sqrt[3]{C} = 5e^{j(75°+1\cdot360°)/3} = 5e^{j(435°)/3} = 5e^{j(145°)}$;

and

3rd root → $\sqrt[3]{C} = 5e^{j(75°+2\cdot360°)/3} = 5e^{j(795°)/3} = 5e^{j(265°)}$.

A.3.8 Natural Logarithms of a Complex Number

Taking the natural logarithm of a complex number $C = Me^{j\varnothing}$ is straightforward using exponential notation; that is

$$\ln C = \ln(Me^{j\varnothing}) = \ln M + \ln(e^{j\varnothing}) = \ln M + j\varnothing\,, \tag{A-28}$$

where $0 \le \varnothing < 2\pi$. By way of example, if $C = 12e^{j\pi/4}$, the natural logarithm of C is

$$\ln C = \ln(12e^{j\pi/4}) = \ln(12) + j\pi/4 = 2.485 + j0.785\,. \tag{A-29}$$

This means that $e^{(2.485 + j0.785)} = e^{2.485} \cdot e^{j0.785} = 12e^{j\pi/4}$.

A.3.9 Logarithm to the Base 10 of a Complex Number

We can calculate the base 10 logarithm of the complex number $C = Me^{j\varnothing}$ using

$$\log_{10} C = \log_{10}(Me^{j\varnothing}) = \log_{10} M + \log_{10}(e^{j\varnothing}) = \log_{10} M + j\varnothing \cdot \log_{10}(e)\,.^{\dagger} \tag{A-30}$$

Of course e is the irrational number, approximately equal to 2.71828, whose log to the base 10 is approximately 0.43429. Keeping this in mind, we can simplify Eq. (A-30) as

$$\log_{10} C \approx \log_{10} M + j(0.43429 \cdot \varnothing)\,. \tag{A-31}$$

Repeating the above example with $C = 12e^{j\pi/4}$ and using the Eq. (A-31) approximation, the base 10 logarithm of C is

---

† For the second term of the result in Eq. (A-30) we used $\log_a(x^n) = n \cdot \log_a x$ according to the law of logarithms.

$$\log_{10} C = \log_{10}(12e^{j\pi/4}) = \log_{10}(12) + j(0.43429 \cdot \pi/4)$$

$$= 1.079 + j(0.43429 \cdot 0.785) = 1.079 + j0.341 . \quad \text{(A-32)}$$

The result from Eq. (A-32) means that

$$10^{(1.079 + j0.341)} = 10^{1.079} \cdot 10^{j0.341} = 12 \cdot (e^{2.302})^{j0.341}$$

$$= 12e^{j(2.302 \cdot 0.341)} = 12e^{j0.785} = 12e^{j\pi/4} . \quad \text{(A-33)}$$

A.3.10 Log to the Base 10 of a Complex Number Using Natural Logarithms

Unfortunately, some software mathematics packages have no base 10 logarithmic function and can calculate only natural logarithms. In this situation, we just use

$$\log_{10}(x) = \frac{\ln(x)}{\ln(10)} \quad \text{(A-34)}$$

to calculate the base 10 logarithm of x. Using this *change of base* formula, we can find the base 10 logarithm of a complex number $C = Me^{j\varnothing}$; that is,

$$\log_{10} C = \frac{\ln C}{\ln 10} = (\log_{10} e)(\ln C). \quad \text{(A-35)}$$

Because $\log_{10}(e)$ is approximately equal to 0.43429, we use Eq. (A-35) to state that

$$\log_{10} C \approx 0.43429 \cdot (\ln C) = 0.43429 \cdot (\ln M + j\varnothing) . \quad \text{(A-36)}$$

Repeating, again, the example above of $C = 12e^{j\pi/4}$, the Eq. (A-36) approximation allows us to take the base 10 logarithm of C using natural logs as

$$\log_{10} C = 0.43429 \cdot (\ln(12) + \pi/4)$$

$$= 0.43429 \cdot (2.485 + j0.785) = 1.079 + j0.341 , \quad \text{(A-37)}$$

giving us the same result as Eq. (A-32).

A.4 Some Practical Implications of Using Complex Numbers

At the beginning of Section A.3, we said that the choice of using the rectangular versus the polar form of representing complex numbers depends on the type of arithmetic operations we intend to perform. It's interesting to note that the rectangular form has a practical advantage over the polar form when we consider how numbers are represented in a computer. For example, let's say we must represent our complex numbers using a four-bit sign-magnitude binary number format. This means that we can have integral numbers ranging from –7 to +7, and our range of complex numbers covers a square on the complex plane as shown in Figure A-4(a) when we use the rectangular form. On the other hand, if we used 4-bit numbers to represent the magnitude of a complex number in polar form, those numbers must reside on or within a circle whose radius is 7 as shown in Figure A-4(b). Notice how the four shaded corners in Figure A-4(b) represent locations of valid complex values using the rectangular form, but are *out of bounds* if we use the polar form. Put another way, a complex number calculation, yielding an acceptable result in rectangular form, could result in an overflow error if we used polar notation in our computer. We could accommodate the complex value $7 + j7$ in rectangular form but not its polar equivalent because the magnitude of that polar number is greater than 7.

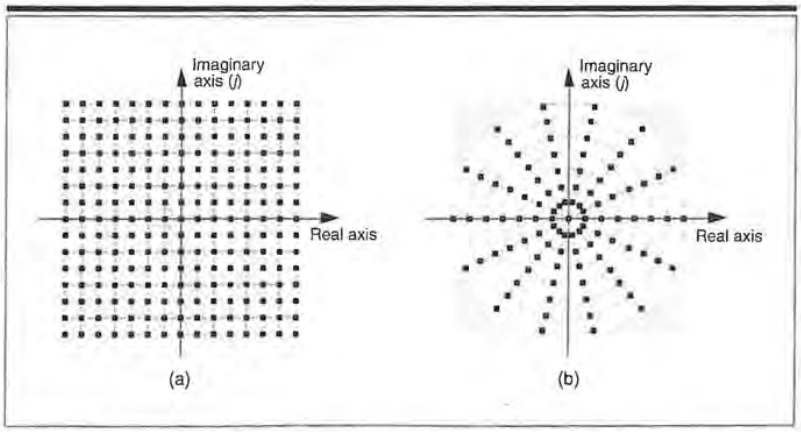


Figure A-4 Complex integral numbers represented as points on the complex plane using a four-bit sign-magnitude data format: (a) using rectangular notation; (b) using polar notation.

Although we avoid any further discussion here of the practical implications of performing complex arithmetic using standard digital data formats, it is an intricate and interesting subject. To explore this topic further, the inquisitive reader is encouraged to start with the references.

References

[1] Plauger, P. J. "Complex Math Functions," *Embedded Systems Programming*, August 1994.

[2] Kahan, W. "Branch Cuts for Complex Elementary Functions or Much Ado About Nothing's Sign Bit," *Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis*, Clarendon Press, 1987.

[3] Plauger, P. J. "Complex Made Simple," *Embedded Systems Programming*, July 1994.

APPENDIX B

Closed Form of a Geometric Series

In the literature of digital signal processing, we often encounter geometric series expressions like

$$\sum_{n=p}^{N-1} r^n = \frac{r^p - r^N}{1-r}\,, \tag{B-1}$$

or

$$\sum_{n=0}^{N-1} e^{-j2\pi nm/N} = \frac{1-e^{-j2\pi m}}{1-e^{-j2\pi m/N}}\,. \tag{B-2}$$

Unfortunately, many authors make a statement like "and we know that," and drop Eqs. (B-1) or (B-2) on the unsuspecting reader who's expected to accept these expressions on faith. Assuming that you don't have a Ph.D. in mathematics, you may wonder exactly what arithmetic sleight of hand allows us to arrive at Eqs. (B-1) or (B-2). To answer this question, let's consider a general expression for a geometric series such as

$$S = \sum_{n=p}^{N-1} ar^n = ar^p + ar^{p+1} + ar^{p+2} + \ldots + ar^{N-1}\,, \tag{B-3}$$

where n, N, and p are integers and a and r are any constants. Multiplying Eq. (B-3) by r, gives us

$$Sr = \sum_{n=p}^{N-1} ar^{n+1} = ar^{p+1} + ar^{p+2} + \ldots + ar^{N-1} + ar^N\,. \tag{B-4}$$

Subtracting Eq. (B-4) from Eq. (B-3) gives the expression

$$S - Sr = S(1 - r) = ar^p - ar^N ,$$

or

$$S = a \cdot \frac{r^p - r^N}{1 - r} . \tag{B-5}$$

So here's what we're after. The *closed form* of the series is

Closed form of a general geometric series: →

$$\sum_{n=p}^{N-1} ar^n = a \cdot \frac{r^p - r^N}{1 - r} . \tag{B-6}$$

(By closed form, we mean taking an infinite series and converting it to a simpler mathematical form without the summation.) When $a = 1$, Eq. (B-6) validates Eq. (B-1). We can quickly verify Eq. (B-6) with an example. Letting $N = 5$, $p = 0$, $a = 2$, and $r = 3$, for example, we can create the following list:

n	$ar^n = 2 \cdot 3^n$
0	$2 \cdot 3^0 = 2$
1	$2 \cdot 3^1 = 6$
2	$2 \cdot 3^2 = 18$
3	$2 \cdot 3^3 = 54$
4	$2 \cdot 3^4 = 162$
	The sum of this column is $\sum_{n=0}^{4} 2 \cdot 3^n = 242$.

Plugging our example N, p, a, and r values into Eq. (B-6),

$$\sum_{n=p}^{N-1} ar^n = a \cdot \frac{r^p - r^N}{1 - r} = 2 \cdot \frac{3^0 - 3^5}{1 - 3} = 2 \cdot \frac{1 - 243}{-2} = 242 , \tag{B-7}$$

which equals the sum of the rightmost column in the list above.

As a final step, the terms of our earlier Eq. (B-2) are in the form of Eq. (B-6) as $p = 0$, $a = 1$, and $r = e^{-j2\pi m/N}$.† So plugging those terms from Eq. (B-2) into Eq. (B-6) gives us

$$\sum_{n=0}^{N-1} e^{-j2\pi nm/N} = 1 \cdot \frac{e^{-j2\pi m0/N} - e^{-j2\pi mN/N}}{1 - e^{-j2\pi m/N}} = \frac{1 - e^{-j2\pi m}}{1 - e^{-j2\pi m/N}}, \qquad \text{(B-8)}$$

confirming Eq. (B-2).

† From the math identity $a^{xy} = (a^x)^y$, we can say $e^{-j2\pi nm/N} = (e^{-j2\pi m/N})^n$, so $r = e^{-j2\pi m/N}$.