# Streamlining Digital Signal Processing

## A Tricks of the Trade Guidebook

Edited by

**Richard G. Lyons**

*Besser Associates*
*Mountain View, California*

**IEEE**

IEEE PRESS

BICENTENNIAL
1807
WILEY
2007
BICENTENNIAL

**WILEY-INTERSCIENCE**
**A John Wiley & Sons, Inc., Publication**

# Streamlining Digital
# Signal Processing

# Streamlining Digital Signal Processing

## A Tricks of the Trade Guidebook

Edited by

**Richard G. Lyons**

*Besser Associates*
*Mountain View, California*

**IEEE**

IEEE PRESS

*This book is dedicated to all the signal processing engineers who struggle to learn their craft, and willingly share that knowledge with their engineering brethren—people of whom the English poet Chaucer would say, "Gladly would he learn and gladly teach."*

# Contents

# Preface

**T**his book presents recent advances in digital signal processing (DSP) to simplify, or increase the computational speed of, common signal processing operations. The topics here describe clever DSP *tricks of the trade* not covered in conventional DSP textbooks. This material is practical real-world DSP tips and tricks, as opposed to the traditional, highly specialized, math-intensive research subjects directed at industry researchers and university professors. Here we go beyond the standard *DSP fundamentals* textbook and present new, but tried-"n"-true, clever implementations of digital filter design, spectrum analysis, signal generation, high-speed function approximation, and various other DSP functions.

Our goal in this book is to create a resource that is relevant to the needs of the working DSP engineer by helping bridge the theory-to-practice gap between introductory DSP textbooks and the esoteric, difficult-to-understand academic journals. We hope the material in this book makes the practicing DSP engineer say, "Wow! That's pretty neat—I have to remember this; maybe I can use it sometime." While this book will be useful to experienced DSP engineers, due to its gentle tutorial style it will also be of considerable value to the DSP beginner.

The mathematics used here is simple algebra and the arithmetic of complex numbers, making this material accessible to a wide engineering and scientific audience. In addition, each chapter contains a reference list for those readers wishing to learn more about a given DSP topic. The chapter topics in this book are written in a standalone number, so the subject matter can be read in any desired order.

The contributors to this book make up a *dream team* of experienced DSP engineer-authors. They are not only knowledgeable in signal processing theory, they are "make it work" engineers who build working DSP systems. (They actually know which end of the soldering iron is hot.) Unlike many authors whose writing seems to say, "I understand this topic and I defy you to understand it," our contributors go all-out to convey as much DSP understanding as possible. As such the

chapters of this book are postcards from our skilled contributors on their endless quest for signal processing's Holy Grail: accurate processing results at the price of a bare minimum of computations.

We welcome you to this DSP tricks of the trade guidebook. I and the IEEE Press hope you find it valuable.

RICHARD G. LYONS
E-mail: R.Lyons@ieee.org

"If you really wish to learn then you must mount the machine and become acquainted with its tricks by actual trial."
—Wilbur Wright, co-inventor of the first successful airplane, 1867–1912

# Contributors

**Mark Allie**
University of Wisconsin—Madison
Madison, Wisconsin

**Amy Bell**
Virginia Tech
Blacksburg, Virginia

**Greg Berchin**
Consultant in Signal Processing
Colchester, Vermont

**Mark Borgerding**
3dB Labs, Inc.
Cincinnati, Ohio

**Joan Carletta**
University of Akron
Akron, Ohio

**Lionel Cordesses**
Technocentre, Renault
Guyancourt, France

**Matthew Donadio**
Night Kitchen Interactive
Philadelphia, Pennsylvania

**Shlomo Engelberg**
Jerusalem College of Technology
Jerusalem, Israel

**Huei-Wen Ferng**
National Taiwan University of Science
    and Technology
Taipei, Taiwan, ROC

**Fredric Harris**
San Diego State University
San Diego, California

**Laszlo Hars**
Seagate Research
Pittsburgh, Pennsylvania

**Robert Inkol**
Defense Research and Development
Ottawa, Canada

**Eric Jacobsen**
Abineau Communications
Flagstaff, Arizona

**Alain Joyal**
Defense Research and Development
Ottawa, Canada

**Peter Kootsookos**
UTC Fire & Security Co.
Farmington, Connecticut

**Kishore Kotteri**
Microsoft Corp.
Redmond, Washington

**Ricardo Losada**
The MathWorks, Inc.
Natick, Massachusetts

**Richard Lyons**
Besser Associates
Mountain View, California

**James McNames**
Portland State University
Portland, Oregon

**Vincent Pellissier**
The MathWorks, Inc.
Natick, Massachusetts

**Charles Rader**
Retired, formerly with MIT Lincoln
    Laboratory
Lexington, Massachusetts

**Sreeraman Rajan**
Defense Research and Development
Ottawa, Canada

**Josep Sala**
Technical University of Catalonia
Barcelona, Spain

**David Shiung**
MediaTek Inc.
Hsin-chu, Taiwan, ROC

**Rainer Storn**
Rohde & Schwarz GmbH & Co. KG
Munich, Germany

**Clay Turner**
Wireless Systems Engineering Inc.
Satellite Beach, Florida

**Krishnaraj Varma**
Hughes Network Systems
Germantown, Maryland

**Vladimir Vassilevsky**
Abvolt Ltd.
Perry, Oklahoma

**Sichun Wang**
Defense Research and Development
Ottawa, Canada

**Jyri Ylöstalo**
Nokia Siemens Networks
Helsinki, Finland

# Efficient Digital Filters

# Chapter 1

# Lost Knowledge Refound: Sharpened FIR Filters

**Matthew Donadio**

*Night Kitchen Interactive*

**W**hat would you do in the following situation? Let's say you are diagnosing a DSP system problem in the field. You have your trusty laptop with your development system and an emulator. You figure out that there was a problem with the system specifications and a symmetric FIR filter in the software won't do the job; it needs reduced passband ripple, or maybe more stopband attenuation. You then realize you don't have any filter design software on the laptop, and the customer is getting angry. The answer is easy: You can take the existing filter and *sharpen* it. Simply stated, filter sharpening is a technique for creating a new filter from an old one [1]–[3]. While the technique is almost 30 years old, it is not generally known by DSP engineers nor is it mentioned in most DSP textbooks.

## 1.1  IMPROVING A DIGITAL FILTER

Before we look at filter sharpening, let's consider the first solution that comes to mind, filtering the data twice with the existing filter. If the original filter's transfer function is $H(z)$, then the new transfer function (of the $H(z)$ filter cascaded with itself) is $H(z)^2$. For example, let's assume the original lowpass $N$-tap FIR filter, designed using the Parks-McClellan algorithm [4], has the following characteristics:

  Number of coefficients: $N = 17$

  Sample rate: $F_s = 1$
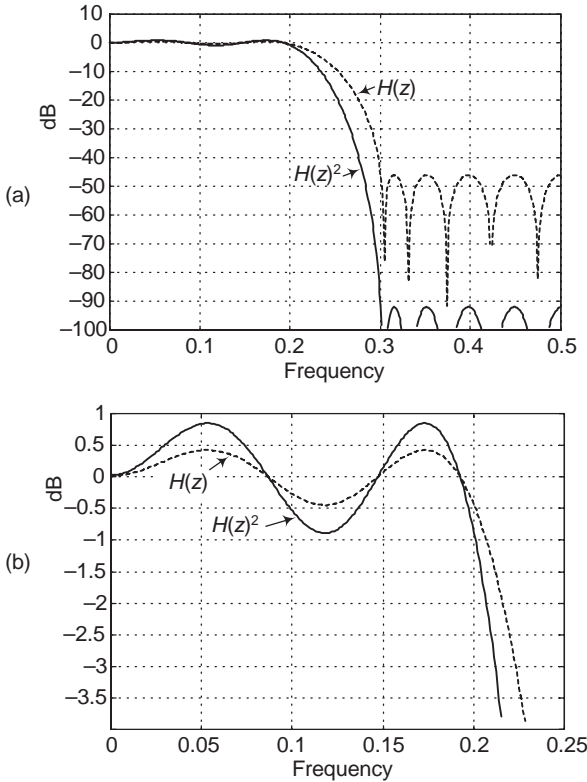
**Figure 1-1**    $H(z)$ and $H(z)^2$ performance: (a) full frequency response; (b) passband response.

Passband width: $f_{pass}=0.2$

Passband deviation: $\delta_{pass}=0.05$ (0.42 dB peak ripple)

Stopband frequency: $f_{stop}=0.3$

Stopband deviation: $\delta_{stop}=0.005$ (−46 dB attenuation)

Figure 1-1(a) shows the performance of the $H(z)$ and cascaded $H(z)^2$ filters. Everything looks okay. The new filter has the same band edges, and the stopband attenuation is increased. But what about the passband? Let's zoom in and take a look at Figure 1-1(b). The squared filter, $H(z)^2$, has larger deviations in the passband than the original filter. In general, the squaring process will:

1. Approximately double the error (response ripple) in the passband.
2. Square the errors in the stopband (i.e., double the attenuation in dB in the stopband).
3. Leave the passband and stopband edges unchanged.
4. Approximately double the impulse response length of the original filter.
5. Maintain filter phase linearity.
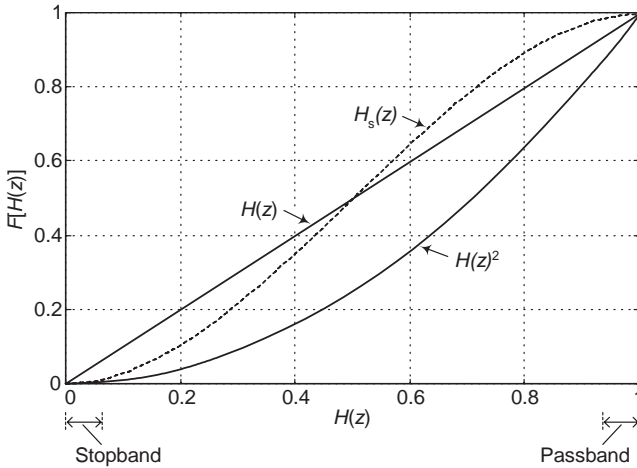
**Figure 1-2**   Various $F[H(z)]$ functions operating on $H(z)$.

It is fairly easy to examine this operation to see the observed behavior if we view the relationship between $H(z)$ and $H(z)^2$ in a slightly unconventional way. We can think of filter squaring as a function $F[H(z)]$ operating on the $H(z)$ transfer function. We can then plot the output amplitude of this function, $H(z)^2$, versus the amplitude of the input $H(z)$ to visualize the amplitude change function.

The plot for $F[H(z)]=H(z)$ is simple; the output is the input, so the result is the straight line as shown in Figure 1-2. The function $F[H(z)]=H(z)^2$ is a quadratic curve. When the $H(z)$ input amplitude is near zero, the $H(z)^2$ output amplitude is closer to zero, which means the stopband attenuation is increased with $H(z)^2$. When the $H(z)$ input amplitude is near one, the $H(z)^2$ output band is approximately twice as far away from one, which means the passband ripple is increased.

The squaring process improved the stopband, but degraded the passband. The improvement was a result of the amplitude change function being horizontal at 0. So to improve $H(z)$ in both the passband and stopband, we want the $F[H(z)]$ amplitude function to be horizontal at both $H(z)=0$ and $H(z)=1$ (in other words, have a first derivative of zero at these points). This results in the output amplitude changing more slowly than the input amplitude as we move away from 0 and 1, which lowers the ripple in these areas. The simplest function that meets this will be a cubic of the form

$$F(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3. \tag{1-1}$$

Its derivative (with respect to $x$) is

$$F(x) = c_1 + 2c_2 x + 3c_3 x^2. \tag{1-2}$$

Specifying $F(x)$ and $F'(x)$ for the two values of $x=0$ and $x=1$ allows us to solve (1–1) and (1–2) for the $c_n$ coefficients as

$$F(x)|_{x=0} = 0 \Rightarrow c_0 = 0 \tag{1-3}$$

$$F(x)|_{x=0} = 0 \Rightarrow c_1 = 0 \tag{1-4}$$

$$F(x)|_{x=1} = 1 \Rightarrow c_2 + c_3 = 1 \tag{1-5}$$

$$F(x)|_{x=1} = 0 \Rightarrow 2c_2 + 3c_3 = 0. \tag{1-6}$$

Solving (1–5) and (1–6) simultaneously yields $c_2 = 3$ and $c_3 = -2$, giving us the function

$$F(x) = 3x^2 - 2x^3 = (3 - 2x)x^2. \tag{1-7}$$

Stating this function as the sharpened filter $H_s(z)$ in terms of $H(z)$, we have

$$H_s(z) = 3H(z)^2 - 2H(z)^3 = [3 - 2H(z)]H(z)^2. \tag{1-8}$$

The function $H_s(z)$ is the dotted curve in Figure 1-2.

## 1.2  FIR FILTER SHARPENING

$H_s(z)$ is called the *sharpened* version of $H(z)$. If we have a function whose $z$-transform is $H(z)$, then we can outline the filter sharpening procedure, with the aid of Figure 1-3, as the following:

1. Filter the input signal, $x(n)$, once with $H(z)$.
2. Double the filter output sequence to obtain $w(n)$.
3. Subtract $w(n)$ from $3x(n)$ to obtain $u(n)$.
4. Filter $u(n)$ twice by $H(z)$ to obtain the output $y(n)$.

Using the sharpening process results in the improved $H_s(z)$ filter performance shown in Figure 1-4, where we see the increased stopband attenuation and reduced passband ripple beyond that afforded by the original $H(z)$ filter.

It's interesting to notice that $H_s(z)$ has the same half-power frequency ($-6\,\text{dB}$ point) as $H(z)$. This condition is not peculiar to the specific filter sharpening example used here—it's true for all $H_s(z)$s implemented as in Figure 1-3. This characteristic, useful if we're sharpening a halfband FIR filter, makes sense if we substitute 0.5 for $H(z)$ in (1–8), yielding $H_s(z) = 0.5$.
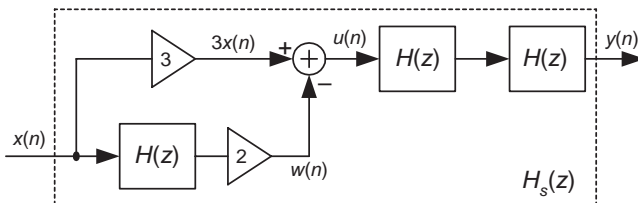


**Figure 1-3**    Filter sharpening process.
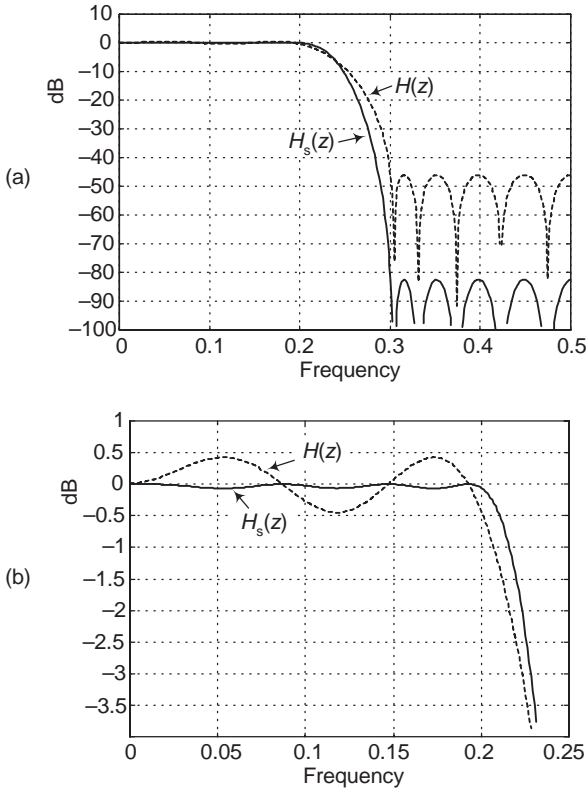
**Figure 1-4**  $H(z)$ and $H_s(z)$ performance: (a) full frequency response; (b) passband response.

## 1.3  IMPLEMENTATION ISSUES

The filter sharpening procedure is very easy to perform, and is applicable to a broad class of FIR filters—including lowpass, bandpass, and highpass FIR filters having symmetrical coefficients and even-order (an odd number of taps). Even multi-passband FIR filters, under the restriction that all passband gains are equal, can be sharpened.

From an implementation standpoint, to correctly implement the sharpening process in Figure 1-3 we must delay the $3x(n)$ sequence by the group delay, $(N–1)/2$ samples, inherent in $H(z)$. In other words, we must time-align $3x(n)$ and $w(n)$. This is analogous to the need to delay the real path in a practical Hilbert transformer. Because of this time-alignment constraint, filter sharpening is not applicable to filters having nonconstant group delay, such as minimum-phase FIR filters or infinite impulse response (IIR) filters. In addition, filter sharpening is inappropriate for Hilbert transformer, differentiating FIR filters, and filters with shaped bands such as

sinc compensated filters and raised cosine filters, because cascading such filters corrupts their fundamental properties.

If the original $H(z)$ FIR filter has a nonunity passband gain, the derivation of (1–8) can be modified to account for a passband gain $G$, leading to a *sharpening polynomial* of:

$$H_{\text{s,gain}>1}(z) = \frac{3H(z)^2}{G} - \frac{2H(z)^3}{G^2} = \left[\frac{3}{G} - \frac{2H(z)}{G^2}\right]H(z)^2. \qquad (1\text{–}9)$$

Notice when $G=1$, $H_{\text{s,gain}>1}(z)$ in (1–9) is equal to our $H_{\text{s}}(z)$ in (1–8).

## 1.4    CONCLUSIONS

We've presented a simple method for transforming a FIR filter into one with better passband and stopband characteristics, while maintaining phase linearity. While filter sharpening may not be often used, it does have its place in an engineer's toolbox. An optimal (Parks-McClellan-designed) filter will have a shorter impulse response than a sharpened filter with the same passband and stopband ripple, and thus be more computationally efficient. However, filter sharpening can be used whenever a given filter response cannot be modified, such as software code that makes use of an unchangeable filter subroutine. The scenario we described was hypothetical, but all practicing engineers have been in situations in the field where a problem needs to be solved without the full arsenal of normal design tools. Filter sharpening could be used when improved filtering is needed but insufficient ROM space is available to store more filter coefficients, or as a way to reduce ROM requirements. In addition, in some hardware filter applications using *application-specific integrated circuits* (ASICs), it may be easier to add additional chips to a filter design than it is to design a new ASIC.

## 1.5    REFERENCES

[1] J. Kaiser and R. Hamming, "Sharpening the Response of a Symmetric Nonrecursive Filter by Multiple Use of the Same Filter," *IEEE Trans. Acoustics, Speech, Signal Proc.*, vol. ASSP-25, no. 5, 1977, pp. 415–422.

[2] R. Hamming, *Digital Filters*. Prentice Hall, Englewood Cliffs, 1977, pp. 112–117.

[3] R. Hamming, *Digital Filters*, 3rd ed. Dover, Mineola, New York, 1998, pp. 140–145.

[4] T. Parks and J. McClellan, "A Program for the Design of Linear Phase Finite Impulse Response Digital Filters," *IEEE Trans. Audio Electroacoust.*, vol. AU-20, August 1972, pp. 195–199.

## EDITOR COMMENTS

When $H(z)$ is a unity-gain filter we can eliminate the multipliers shown in Figure 1-3. The multiply-by-two operation can be implemented with an arithmetic left-shift by one binary bit. The multiply-by-three operation can be implemented by adding a binary signal sample to a shifted-left-by-one-bit version of itself.
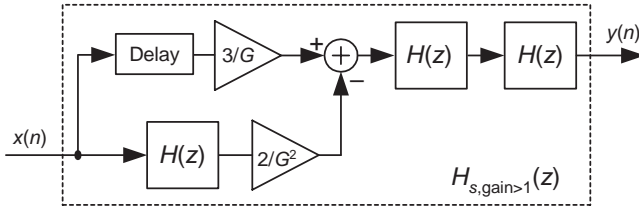
**Figure 1-5** Nonunity gain filter sharpening.

To further explain the significance of (1–9), the derivation of (1–8) was based on the assumption that the original $H(z)$ filter to be sharpened had a passband gain of one. If the original filter has a nonunity passband gain of $G$, then (1–8) will not provide proper sharpening; in that case (1–9) must be used as shown in Figure 1-5. In that figure we've included a *Delay* element, whose length in samples is equal to the group delay of $H(z)$, needed for real-time signal synchronization.

It is important to realize that the $3/G$ and $2/G^2$ scaling factors in Figure 1-5 provide optimum filter sharpening. However, those scaling factors can be modified to some extent if doing so simplifies the filter implementation. For example, if $2/G^2 = 1.8$, for ease of implementation, the practitioner should try using a scaling factor of 2 in place of 1.8 because multiplication by 2 can be implemented by a simple binary left-shift by one bit. Using a scaling factor of 2 will not be optimum but it may well be acceptable, depending on the characteristics of the filter to be sharpened. Software modeling will resolve this issue.

As a historical aside, *filter sharpening* is a process refined and expanded by the accomplished R. Hamming (of Hamming window fame) based on an idea originally proposed by the great American mathematician John Tukey, the inventor of the radix-2 fast Fourier transform (FFT).

# Chapter 2

# Quantized FIR Filter Design Using Compensating Zeros

**Amy Bell**
*Virginia Tech*

**Joan Carletta**
*University of Akron*

**Kishore Kotteri**
*Microsoft Corp.*

**T**his chapter presents a design method for translating a finite impulse response (FIR) floating-point filter design into an FIR fixed-point multiplierless filter design. This method is simple and fast, and provides filters with high performance. Conventional wisdom dictates that finite word-length (i.e., quantization) effects can be minimized by dividing a filter into smaller, cascaded sections. The design method presented here takes this idea a step further by showing how to quantize the cascaded sections so that the finite wordlength effects in one section are guaranteed to compensate for the finite wordlength effects in the other section. This simple method, called *compensating zeros*, ensures that: (1) the quantized filter's frequency response closely matches the unquantized filter's frequency response (in both magnitude and phase); and (2) the required hardware remains small and fast.

Digital filter design typically begins with a technique to find double-precision, floating-point filter coefficients that meet some given performance specifications—like the magnitude response and phase response of the filter. Two well-known techniques for designing floating-point FIR filters are the windowing method and the equiripple Parks-McClellan method [1], [2]. If the filter design is

for a real-time application, then the filter must be translated to fixed-point, a more restrictive form of mathematics that can be performed much more quickly in hardware. For embedded systems applications, a multiplierless implementation of a filter is advantageous; it replaces multiplications with faster, cheaper shifts and additions. Translation to a fixed-point, multiplierless implementation involves quantizing the original filter coefficients (i.e., approximating them using fixed-point mathematics). The primary difficulty with real-time implementations is that this translation alters the original design; consequently, the desired filter's frequency response characteristics are often not preserved.

Multiplierless filter design can be posed as an optimization problem to minimize the degradation in performance; simulated annealing, genetic algorithms, and integer programming are among the many optimization techniques that have been employed [3]. However, in general, optimization techniques are complex, can require long run times, and provide no performance guarantees. The compensating zeros technique is a straightforward, intuitive method that renders optimization unnecessary; instead, the technique involves the solution of a linear system of equations. It is developed and illustrated with two examples involving real-coefficient FIR filters; the examples depict results for the frequency response as well as hardware speed and size.

## 2.1 QUANTIZED FILTER DESIGN FIGURES OF MERIT

Several important figures of merit are used to evaluate the performance of a filter implemented in hardware. The quantized filter design evaluation process begins with the following two metrics.

1. ***Magnitude MSE.*** Magnitude mean-squared-error (MSE) represents the average of the squared difference between the magnitude response of the quantized (fixed-point) filter and the unquantized (ideal, floating-point) filter over all frequencies. A linear phase response can easily be maintained after quantization by preserving symmetry in the quantized filter coefficients.

2. ***Hardware complexity.*** In a multiplierless filter, all mathematical operations are represented by shifts and additions. This requires that each quantized filter coefficient be expressed as sums and differences of powers of two: for each coefficient, a representation called *canonical signed digit* (CSD) is used [3]. CSD format expresses a number as sums and differences of powers of two using a minimum number of terms. Before a quantized filter design is implemented in actual hardware, the hardware complexity is estimated in terms of $T$, the total number of non-zero terms used when writing all filter coefficients in CSD format. In general, the smaller $T$ is, the smaller and faster

will be the hardware implementation. For application-specific integrated circuit and field-programmable gate array filter implementations, a fully parallel hardware implementation requires $T$–1 adders; an embedded processor implementation requires $T$–1 addition operations.

Once the filter has been implemented in hardware, it can be evaluated more directly. Important metrics from a hardware perspective include: hardware size, throughput (filter outputs per second), and latency (time from filter input to corresponding filter output). The relative importance of these metrics depends on the application.

The goal of the quantized filter design is to achieve a small-magnitude MSE while keeping the hardware costs low. In general, the higher the value of $T$, the closer the quantized filter coefficients are to the unquantized coefficients and the smaller the magnitude MSE. Conversely, smaller $T$ implies worse-magnitude MSE. Hence, there is a trade-off between performance and hardware cost; $T$ can be thought of as the parameter that controls this trade-off.

## 2.2   FILTER STRUCTURES

Filter designs can be implemented in hardware using various structures. The three most common structures are *direct, cascade*, and *lattice*. In general, pole-zero, infinite impulse response (IIR) filters are more robust to quantization effects when the cascade and lattice structures are employed; performance degrades quickly when the direct structure is used [1], [2].

For all-zero, FIR filters, the direct structure usually performs well (if the zeros are not very clustered, but are moderately uniformly distributed) [1], [2]. Moreover, since most FIR filters have linear phase (the filter coefficients are symmetric), the lattice structure cannot be used because at least one reflection coefficient equals ±1. Although the direct structure is a good choice for many FIR filter implementations, the cascade structure offers at least one advantage. When an FIR filter is quantized using a direct structure, the quantization of one coefficient affects all of the filter's zeros. In contrast, if an FIR filter is quantized with a cascade structure, the quantization of coefficients in one of the cascaded sections affects only those zeros in its section—the zeros in the other cascaded sections are isolated and unaffected. Depending on the application, it may be important to more closely approximate the unquantized locations of some zeros than others.

The compensating zeros method uses a cascade structure. However, it goes beyond a "simple quantization" technique that uniformly divvies up the given $T$ non-zero terms in CSD format across the coefficients in the cascaded sections. The next section first illustrates a simple quantization approach for an FIR filter design using a cascade structure; then the compensating zeros method is developed and used to redesign the same FIR filter. The result is an improvement in the magnitude MSE for the same $T$.

## 2.3  EXAMPLE 1: A WINDOWED FIR FILTER

Consider a lowpass, symmetric, length-19 FIR filter designed using a rectangular window. The filter has a normalized (such that a frequency of one corresponds to the sampling rate) passband edge frequency of 0.25 and exhibits linear phase. The floating-point filter coefficients are listed in Table 2-1 and Figure 2-1 shows the unquantized magnitude response of this filter.

Figure 2-2 illustrates the pole-zero plot for $h(n)$. To implement this filter in the cascade form, $h(n)$ is split into two cascaded sections whose coefficients are $c_1(n)$ and $c_2(n)$. This is accomplished by distributing the zeros of $h(n)$ between $c_1(n)$ and $c_2(n)$. To separate the zeros of $h(n)$ into the two cascaded sections, the $z$-plane is scanned from $\omega = 0$ to $\omega = \pi$. As they are encountered, the zeros are placed alternately in the two sections. The first zero encountered is at $z = 0.66e^{j0.324}$. This zero, its conjugate, and the two reciprocals are put in one section. The next zero at $z = 0.69e^{j0.978}$, its conjugate, and the reciprocal pair are placed in the other section. This proceeds until all of the zeros of the unquantized filter are divided among the two cascade sections.

The steps in this zero-allocation process are as follows: Compute the roots of $h(n)$; partition those roots into two sets of roots as described above; and determine

**Table 2-1**   Unquantized Windowed FIR Filter Coefficients, $h(n)$

| $n$ | $h(n)$ |
|---|---|
| 0, 18 | 0.03536776513153 |
| 1, 17 | −1.94908591626e−017 |
| 2, 16 | −0.04547284088340 |
| 3, 15 | 1.94908591626e−017 |
| 4, 14 | 0.06366197723676 |
| 5, 13 | −1.9490859163e−017 |
| 6, 12 | −0.10610329539460 |
| 7, 11 | 1.94908591626e−017 |
| 8, 10 | 0.31830988618379 |
| 9 | 0.50000000000000 |
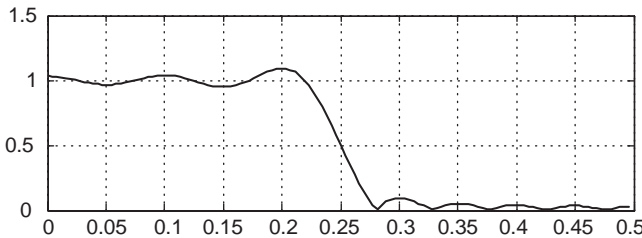


**Figure 2-1**   Frequency magnitude response of $h(n)$ for the windowed FIR filter.
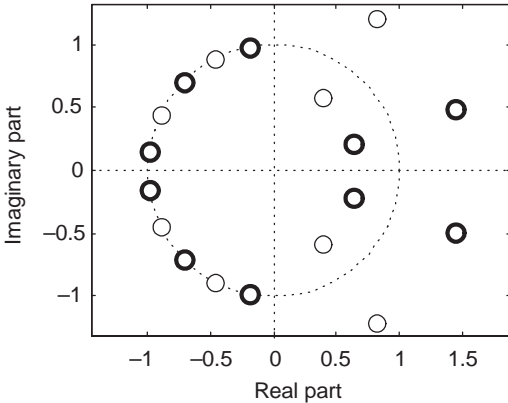
**Figure 2-2**  Pole-zero plot for $h(n)$. The zeros are divided into two cascaded sections by placing the thin zeros in the first section, $c_1(n)$, and the bold zeros in the second section, $c_2(n)$.

**Table 2-2**  Unquantized Cascaded Coefficients for $h(n)$

| $c_1(n)$ | | $c_2(n)$ | |
|---|---|---|---|
| $n_1$ | | $n_2$ | |
| 0, 8 | 1.0000000 | 0, 10 | 1.0000000 |
| 1, 7 | 0.3373269 | 1, 9 | −0.3373269 |
| 2, 6 | 0.9886239 | 2, 8 | −2.1605488 |
| 3, 5 | 1.9572410 | 3, 7 | −0.8949404 |
| 4 | 3.0152448 | 4, 6 | 1.8828427 |
| | | 5 | 3.5382228 |
| | $k=0.0353678$ | | |

the two sets of coefficients, $c_1(n)$ and $c_2(n)$, for the two polynomials associated with the two sets of roots. The section with fewer zeros becomes the first section in the cascade, $c_1(n)$, and the section with more zeros becomes the second section, $c_2(n)$ (this approach provides more degrees of freedom in our design method—see design rule-of-thumb number 6 in Section 2.7).

For the example, the zero allocation is illustrated in Figure 2-2 where the 8 thin zeros go to $c_1(n)$, which has length 9, and the 10 bold zeros go to $c_2(n)$, which has length 11. This method of splitting up the zeros has the advantage of keeping the zeros relatively spread out within each section, thereby minimizing the quantization effects within each section. Because complex conjugate pairs and reciprocal pairs of zeros are kept together in the same cascade section, the two resulting sections have symmetric, real-valued coefficients. The resulting floating-point cascade $c_1(n)$ and $c_2(n)$ coefficients are shown in Table 2-2.

**Figure 2-3**    Direct form of $h(n)$ and the equivalent cascade form using $c_1(n)$, $c_2(n)$, and $k$.

Figure 2-3 depicts the block diagram corresponding to $h(n)$ and the equivalent cascaded form. Coefficients $c_1(n)$ and $c_2(n)$ are normalized so that the first and last coefficients in each section are 1; this ensures that at least two of the coefficients in each cascade section are efficiently represented in CSD format. Consequently, it is necessary to include a gain factor, $k$ in Table 2-2, following the cascade sections. The magnitude response of the cascaded filter is identical to Figure 2-1.

Now consider a fixed-point, quantized, multiplierless design of this cascade structure so that it can be implemented in fast hardware. Assume that there are a fixed total number of CSD terms, $T$, for representing the two unquantized cascaded sections and the unquantized gain factor. Two different techniques are considered for quantizing the filter: a *simple quantization* method that treats each filter section independently, and our proposed *compensating zeros* method in which the quantization errors in one section are compensated for in the next section.

## 2.4   SIMPLE QUANTIZATION

For the simple quantization method, in the process of distributing a fixed number of CSD terms $T$ to a single cascade section with $n$ coefficients, all reasonable distributions are examined. These "reasonable distributions" consider all of the "mostly uniform" $T$ allocation schemes to $n$ coefficients: All coefficients receive at least one CSD term and the remaining CSD terms are allocated to those coefficients that are most different (in terms of percent different) from their unquantized values. Extremely nonuniform allocation schemes (e.g., one coefficient receives all of the $T$ and the remaining coefficients are set to zero) are not considered.

Of all the distribution schemes examined, the distribution that gives the best result (i.e., the smallest-magnitude MSE) is chosen. (*Note*: This does not require an optimization technique; for reasonably small values of $T$, it is simple to organize a search that looks in the area around the floating-point coefficients, which is the only area where high-quality solutions lie). This process ensures that there is no better simple quantization scheme for the given cascaded filter.

In applying the simple quantization method to the windowed FIR filter example, the unquantized cascade coefficients, $c_1(n)$ and $c_2(n)$, are independently quantized to the simple quantized cascade coefficients, $c_1'(n)$ and $c_2'(n)$. In this example, a total of $T=25$ CSD terms was chosen; this choice results in small hardware while still providing a reasonable approximation to the desired filter. Based on the relative lengths of the sections, 9 CSD terms are used for $c_1'(n)$, 14 terms are used for $c_2'(n)$, and 2 terms are used for the quantized gain factor $k'$. The resulting simple quantized coefficients are listed in Table 2-3 (in the CSD format, an underscore indicates that the power of 2 is to be subtracted instead of added).

**Table 2-3**    Simple-Quantized Cascaded Coefficients for
$h(n)$, $(T=25)$

| | $c_1'(n)$ $(T=9)$ | | | $c_2'(n)$ $(T=14)$ | |
|---|---|---|---|---|---|
| $n_1$ | Decimal | CSD | $n_2$ | Decimal | CSD |
| 0, 8 | 1.00 | 001.00 | 0, 10 | 1.000 | 001.000 |
| 1, 7 | 0.25 | 000.01 | 1, 9 | −0.250 | 000.0$\underline{1}$0 |
| 2, 6 | 1.00 | 001.00 | 2, 8 | −2.000 | 0$\underline{1}$0.000 |
| 3, 5 | 2.00 | 010.00 | 3, 7 | −1.000 | 00$\underline{1}$.000 |
| 4 | 4.00 | 100.00 | 4, 6 | 1.875 | 010.00$\underline{1}$ |
| | | | 5 | 3.500 | 100.$\underline{1}$00 |
| | $k'=0.0351563$ | | | 0.00001001 $(T=2)$ | |
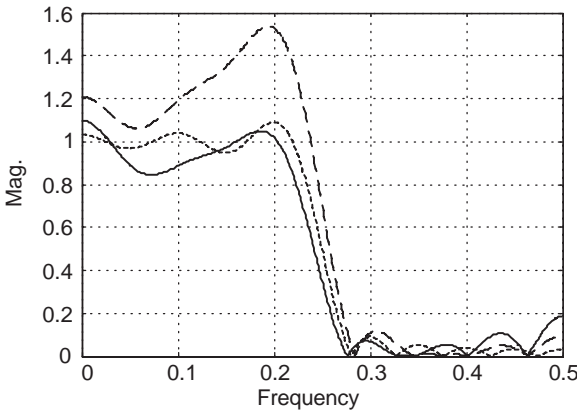


**Figure 2-4**    Frequency responses of the unquantized windowed filter $h(n)$ (dotted), simple
quantization (dashed), and compensating zeros quantization (solid).

The frequency response of the simple quantized filter, $c_1'(n)$ and $c_2'(n)$, is compared with the unquantized filter, $h(n)$, in Figure 2-4. Although a linear phase response is retained after simple quantization (i.e., the simple quantized coefficients are symmetric), the magnitude response is significantly different from the original unquantized case.

## 2.5  COMPENSATING ZEROS QUANTIZATION

The proposed compensating zeros quantization method takes the quantization error of the first cascaded section into account when quantizing the second section. The key to this method is the desire that the frequency response of the quantized cascade structure match the frequency response of the original, unquantized direct structure.

Quantization using the compensating zeros method begins with the quantization of the first cascade section $c_1(n)$ to $c_1'(n)$ and the gain factor $k$ to $k'$ (using the simple quantization method described in the previous section). Next, instead of quantizing $c_2(n)$ to $c_2'(n)$, $c_{comp}(n)$ is computed such that $c_1'(n)$ cascaded with $c_{comp}(n)$ is as close as possible to the original filter $h(n)$. Coefficients $c_{comp}(n)$ are called the *compensating section*, since their aim is to compensate for the performance degradation resulting from the quantization of $c_1'(n)$; the computation of $c_{comp}(n)$ is developed below.

If $C_1(z)$, $C_2(z)$, $C_1'(z)$, $C_2'(z)$, and $C_{comp}(z)$ are the transfer functions of $c_1(n)$, $c_2(n)$, $c_1'(n)$, $c_2'(n)$, and $c_{comp}(n)$ respectively, then the transfer function of the unquantized cascaded filter $H(z)$ can be written as

$$H(z) = kC_1(z)C_2(z), \tag{2--1}$$

where $k$ is the gain factor. The transfer function of the semiquantized filter using the compensating zeros method is given by

$$H'_{comp}(z) = k'C_1'(z)C_{comp}(z). \tag{2--2}$$

$H'_{comp}(z)$ is called the semiquantized filter because $c_{comp}(n)$ has floating-point coefficients. The goal is for the semiquantized and unquantized transfer functions to be equal, that is, $H'_{comp}(z) = H(z)$, or

$$k'C_1'(z)C_{comp}(z) = kC_1(z)C_2(z). \tag{2--3}$$

In (2--3), $C_1(z)C_2(z)$ and $k$ on the right-hand side are the known, unquantized cascade filters. After $c_1(n)$ and $k$ are quantized, $C_1'(z)$ and $k'$ on the left-hand side are known. Thus, (2--3) can be solved for $c_{comp}(n)$.

Since the 11-tap $c_{comp}(n)$ is symmetric (with the first and last coefficients normalized to 1), it can be expressed in terms of only five unknowns. In general, for a length-$N$ symmetric filter with normalized leading and ending coefficients, there are $M$ unique coefficients where $M = \lceil (N-2)/2 \rceil$. (The $\lceil x \rceil$ notation means: the next integer larger than $x$; or if $x$ is an integer, $\lceil x \rceil = x$.)

Equation (2--3) can be evaluated at $M = 5$ values of $z$ to solve for the five unknowns in $c_{comp}(n)$. Since the frequency response is the primary concern, these values of $z$ are on the unit circle. For the example, (2--3) is solved at the frequencies $f = 0$, 0.125, 0.2, 0.3, and 0.45 (i.e., $z = 1$, $e^{j0.25}$, $e^{j0.4}$, $e^{j0.6}$, $e^{j0.9}$).

Table 2-4 lists the computed floating-point coefficients of $c_{comp}(n)$. Now that $c_{comp}(n)$ has been obtained, it is quantized (using simple quantization and the remaining $T$) to arrive at $c_2'(n)$. The final, quantized $c_2'(n)$ filter coefficients using this compensating zeros method are also given in Table 2-4. Thus the compensating zeros quantized filter coefficients are the $c_1'(n)$ and $k'$ from Table 2-2 in cascade with the $c_2'(n)$ in Table 2-4.

The frequency response of the compensating zeros quantized filter is compared with the unquantized filter and the simple quantized filter in Figure 2-4; the overall frequency response of the compensating zeros quantized implementation is closer to the unquantized filter than the simple quantized implementation. The small value of $T = 25$ employed in this example hampers the ability of even the compensating

**Table 2-4**  Unquantized $c_{comp}(n)$ and Compensating
Zeros-Quantized $c'_2(n)$ for $h(n)$ ($T=25$)

| $n$ | $c_{comp}(n)$ | $c'_2(n)$ ($T=14$) | |
|---|---|---|---|
| | | Decimal | CSD |
| 0, 10 | 1.0000000 | 1.00 | 001.00 |
| 1, 9 | −0.7266865 | −0.75 | 00$\underline{1}$.01 |
| 2, 8 | −1.1627044 | −1.00 | 00$\underline{1}$.00 |
| 3, 7 | −0.6238289 | −0.50 | 000.$\underline{1}$0 |
| 4, 6 | 1.2408329 | 1.00 | 001.00 |
| 5 | 2.8920707 | 3.00 | 10$\underline{1}$.00 |

zeros quantized magnitude response to closely approximate the unquantized magnitude response. The magnitude MSE for compensating zeros quantization (7.347e−3) is an order of magnitude better than the magnitude MSE for simple quantization (4.459e−2).

For a fixed value of $T$, there are two ways to quantize the cascaded coefficients: simple and compensating zeros. If $T$ is large enough, then simple quantization can achieve the desired frequency response. However, when $T$ is restricted, compensating zeros quantization provides an alternative that outperforms simple quantization. In the example, it turns out that for $T=26$, the simple quantization method can achieve the same magnitude MSE as the $T=25$ compensating zeros quantization method. This improvement is achieved when the one extra $T$ is assigned to the first cascaded section; no improvement is realized when it is assigned to the second cascaded section. There is an *art* to how to assign the extra $T$: These terms should be allocated to the coefficients—in either cascaded section—that are most different (in terms of percent different) from their unquantized values.

The compensating zeros quantization procedure is outlined as follows:

**Step 1:** Derive the unquantized cascade coefficients, $c_1(n)$, $c_2(n)$, and $k$, from the given, direct unquantized coefficients, $h(n)$.

**Step 2:** Quantize $c_1(n)$ to $c'_1(n)$ and $k$ to $k'$ using the simple quantization method.

**Step 3:** Select a set of $M$ unique positive frequencies. For symmetric filters, $M$ is given by $M=\lceil (N-2)/2 \rceil$; for nonsymmetric filters, $M$ is given by $M=\lceil (N-1)/2 \rceil$, where $N$ is the length of the second cascaded section's $c_2(n)$.

**Step 4:** Solve for the $M$ unknown $c_{comp}(n)$ coefficients by solving (2–3). For symmetric filters, (2–3) is evaluated at the $M$ positive frequencies selected in step 3. For nonsymmetric filters, (2–3) is solved at $M$ positive frequencies and the corresponding $M$ negative frequencies.

**Step 5:** Using the simple quantization method, quantize the floating-point $c_{comp}(n)$ coefficients—with the remaining $T$ from step 2—to arrive at $c'_2(n)$.

**Step 6:** Compute the direct form coefficients of the compensating zeros quantized filter using $H'(z)=k'C_1'(z)C_2'(z)$ (if desired).

Although the quantized $c_2'(n)$ values obtained using the compensating zeros method (in Table 2-4) are not very different from the values obtained using the simple method (in Table 2-3), it is important to note that simple quantization could not possibly find these improved coefficients. In the simple quantization case, the goal is to closely approximate the floating-point $c_1(n)$ and $c_2(n)$ coefficients in Table 2-2. However, in the compensating zeros quantization case, the goal is to closely approximate a different set of floating-point coefficients—$c_1(n)$ and $c_{comp}(n)$. Figure 2-5 compares the zero location plots of the compensating zeros quantized filter (bold circles) and the unquantized filter (thin circles).

Figure 2-5 also shows the zeros $z_1$ and $z_2$ in the first quadrant; recall that $z_1$ is in $c_1(n)$ and $z_2$ is in $c_2(n)$. The quantization of $c_1(n)$ to $c_1'(n)$ moves $z_1$ away from the unit circle. Consequently, the compensating zeros quantization method moves $z_2$ toward the unit circle (Table 2-5 shows the distances of $z_1$ and $z_2$ from the origin before and after quantization). This compensating movement of the quantized zeros is the reason the quantized magnitude response more closely resembles the unquantized magnitude response; it also motivates the name of the quantization method.

The hardware performance of the simple quantized and compensating zeros quantized filters was evaluated by implementing the filters on an Altera field-
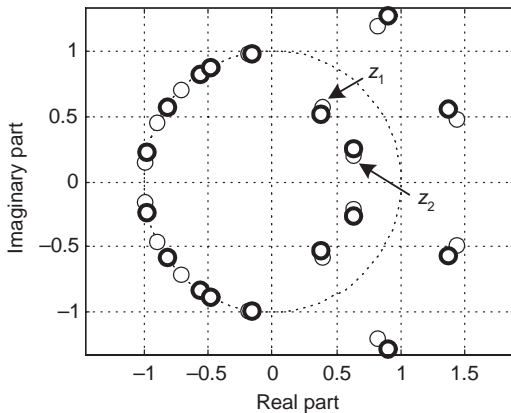


**Figure 2-5**    Zero plot comparing the unquantized zeros of $h(n)$ (thin circles) and the compensating zeros quantized zeros of $h'(n)$ (bold circles).

**Table 2-5**    Distance of the Zeros in Figure 2-5 from the Origin

|  | $|z_2|$ | $|z_1|$ |
| --- | --- | --- |
| Unquantized | 0.66061185 | 0.69197298 |
| Comp. Quantized | 0.67774476 | 0.644315248 |

programmable gate array (FPGA). For each filter, given the desired CSD coefficients, filter synthesis software, written in C, was used to generate a synthesizable description of the filter in VHDL, a hardware-description language. The software automatically chooses appropriate bit widths for internal signals such that errors due to truncation and overflow are completely avoided. In this way, an implemented fixed-point filter provides exactly the same outputs as the same filter would in infinite precision, given the quantized, finite precision filter coefficients. Fixed-point, two's complement adders are used, but the number of integer and fraction bits for each hardware adder are chosen so that no round-off or overflow can occur.

The VHDL description generated is a structural one. A high-performance, multiplierless implementation is achieved. For each section in a filter, a chain of registers is used to shift the data in, and the data is shifted in accordance with the filter coefficients before being summed. For example, if one of the filter coefficients were $18 = 2^4 + 2^1$, the corresponding data word would go to two shifters and be shifted four and one places, respectively, before being summed. A pipelined tree of *carry save adders* (CSAs) is used for the summation. The CSA tree produces two outputs that must be summed, or *vector-merged*, to produce the final filter output. For the results presented here, we use a ripple carry adder for the vector merge, taking care to exploit special-purpose routing (*carry chains*) provided on Altera FPGAs to make ripple carry addition fast.

Table 2-6 summarizes the hardware performance of the filters. Data formats for all signals are shown as (*n, l*), where *n* is the total number of bits, including the sign bit, and $2^l$ is the weight of the least significant bit. Both filters take in inputs of data format (8, 0) (i.e., eight-bit two's complement integers). They vary in terms of their output data formats, depending on the precision of the coefficients used. Throughput is the most important performance metric; it measures how many inputs can be processed per second. Latency also bears on performance, but is less critical; it measures how many clock cycles a particular set of data takes to pass through the system, from input to corresponding output.

The results of Table 2-6 show that the compensating zeros quantized filter is slightly smaller and faster than the simple quantized filter. This is because the coefficients for the compensating zeros quantized case turn out to be slightly less wide (in terms of bits) than those for the simple quantized case, so that the adders also turn out to be slightly less wide.

**Table 2-6**    Hardware Metrics for the Windowed FIR Example

|  | Simple quantized | Compensating zeros quantized |
|---|---|---|
| Hardware complexity (logic elements) | 1042 | 996 |
| Throughput (Mresults/second) | 82.41 | 83.93 |
| Latency (clock cycles) | 15 | 15 |
| Input data format | (8, 0) | (8, 0) |
| Output data format | (23, −13) | (21, −13) |

## 2.6  EXAMPLE 2: A BIORTHOGONAL FIR FILTER

Now consider another example that illustrates the cascade structure's advantage of isolating some of the zeros in an FIR filter. The biorthogonal FIR wavelet filters are best known for their inclusion in the most recent version of the international image-compression standard, JPEG2000 [4]; in this example, the 20-tap, biorthogonal, FIR, lowpass analysis wavelet filter is examined [5]. This filter has a passband edge frequency of 0.31 and exhibits linear phase. This filter has 9 zeros clustered at $z=-1$, and 10 zeros on the right-hand side of the $z$-plane.

For biorthogonal filters, the stopband magnitude response is characterized by the cluster of zeros at $z=-1$; it is advantageous to employ the cascade structure to place all of the zeros at $z=-1$ into an isolated cascade section. This cascade section will keep the zeros exactly at $z=-1$, have only integer coefficients, and require no quantization. Furthermore, this one large cascade section can be split up into smaller $T$-friendly sections having 4, 2, or 1 zeros to reduce the total $T$ required. The remaining zeros on the right-hand side of the $z$-plane determine the passband characteristics and are separated into two cascade sections, $c_1(n)$ and $c_2(n)$, as before. Here the nine zeros at $z=-1$ are divided into two sections of four zeros, $c_3(n)$ and $c_4(n)$, and one section of one zero, $c_5(n)$.

Using the simple quantization method, the unquantized cascade sections $c_1(n)$ and $c_2(n)$ are quantized, but $c_3(n)$, $c_4(n)$ and $c_5(n)$ are captured exactly; a total of $T=38$ CSD is chosen. The compensating zeros quantization method follows the procedure outlined above for $c_1(n)$ and $c_2(n)$; the remaining three cascaded sections are again captured exactly; a total of $T=38$ CSD was used.

Figure 2-6 illustrates that the magnitude response of the compensating zeros quantized implementation is closer to the unquantized filter than the simple quantized implementation. The magnitude MSE for compensating zeros quantization (9.231e–3) is an order of magnitude better than the magnitude MSE for simple quantization (8.449e–2). For the biorthogonal FIR filter, it turns out that four extra
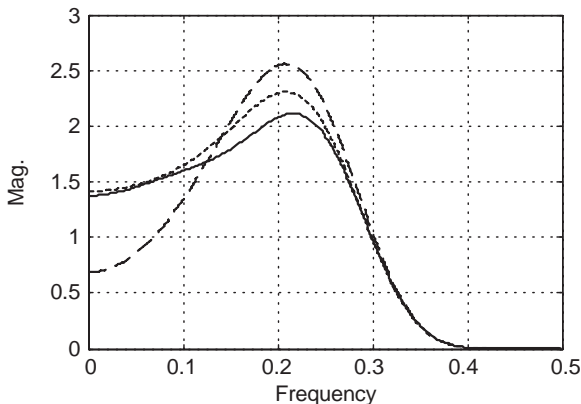


**Figure 2-6**   Frequency response of the unquantized biorthogonal filter (dotted) compared with the two quantized filters: simple quantization (dashed) and compensating zeros quantization (solid).

$T$, ($T=42$), are required for the simple quantization method to achieve the same magnitude MSE as the $T=38$ compensating zeros quantization method.

## 2.7  DESIGN RULES-OF-THUMB

The following guidelines recommend how to derive the maximum benefit from the compensating zeros quantization technique.

1. As $T$ increases for a given filter design, the performance of simple quantization approaches compensating zeros quantization. The performance advantage of the compensating zeros method is realized only when $T$ presents a real constraint on the fixed-point filter design.

2. In the compensating zeros technique, the first cascaded section must be quantized so that it is different from the unquantized filter (i.e., $C_1'(z)$ must be different from $C_1(z)$). This affords the second cascaded section the opportunity to compensate for the quantization effects in the first section.

3. For nonlinear phase FIR filters the filter coefficients are no longer symmetric; consequently, (2–3) must be solved for both positive and negative frequencies to ensure that real coefficients are maintained.

4. The set of frequencies at which (2–3) is evaluated determines how well the compensated magnitude response matches the unquantized response. There is no optimal set of frequencies; instead, several frequency sets may need to be evaluated in order to identify the set that yields the closest match.

5. The original filter must be long enough so that when it is divided into the cascade structure, the sections have sufficient length so that compensation can occur.

6. It is desirable to quantize the shorter cascaded section first and then compensate with the longer cascaded section. The longer the compensating section, the larger the set of frequencies at which (2–3) is evaluated, and the better the match between the quantized and unquantized frequency responses.

7. The compensating zeros method can be employed for multiple cascaded sections. If a filter is divided into $N$ cascaded sections, $c_1$ through $c_N$, then the design begins as described in the example for the first two cascaded sections, $c_1$ and $c_2$. Next, $c_1$ and $c_2$ are combined into one quantized filter and $c_3$ is designed to compensate for the quantization effects in both $c_1$ and $c_2$ (by solving (2–3)). This process continues until the last cascaded section, $c_N$, is designed. Furthermore, not all cascaded sections have to be used to perform the quantization compensation; the choice is up to the designer.

## 2.8  CONCLUSIONS

For hardware filter implementations that use the cascade multiplierless structure, the compensating zeros quantization method outperforms simple quantization given a

small fixed-value of $T$. The compensating zeros technique quantizes the cascaded sections so that the finite word-length effects in one section are guaranteed to compensate for the finite word-length effects in the other section. The algorithm involves no optimization—just the solution of a linear system of equations. Moreover, compensating zeros quantization ensures that: (1) the quantized filter's frequency response closely matches the unquantized filter's frequency response (magnitude and phase), and (2) the required hardware remains small and fast. This technique can be applied to any unquantized FIR filter and it can exploit the cascade structure's ability to isolate some of the zeros in the filter.

## 2.9   REFERENCES

[1]  J. PROAKIS and D. MANOLAKIS, *Digital Signal Processing Principles, Algorithms, and Applications*, 3 ed. Prentice Hall, 1995, pp. 500–652.

[2]  A. OPPENHEIM, R. SCHAFER, and J. BUCK, *Discrete-Time Signal Processing*, 2nd ed. Prentice Hall, 1999, pp. 366–510.

[3]  C. LIM, R. YANG, D. LI, and J. SONG, "Signed Power-of-Two Term Allocation Scheme for the Design of Digital Filters," *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Proc.*, vol. 46, no. 5, May 1999, pp. 577–584.

[4]  "T.800: Information Technology—JPEG2000 Image Coding System," ISO/IEC 15444–1:2002. [Online: http://www.itu.int.]

[5]  I. Daubechies, "Ten Lectures on Wavelets," *SIAM*, 1992, pp. 277.

# Chapter 3

# Designing Nonstandard Filters with Differential Evolution

**Rainer Storn**
*Rohde & Schwarz GmbH & Co. KG*

**M**any filter design tasks deal with standard filter types such as lowpass, highpass, bandpass, or bandstop filters and can be solved with off-the-shelf filter design tools [1]–[3]. In some cases, however, the requirements for the digital filters deviate from the standard ones. This chapter describes a powerful technique for designing nonstandard filters such as the following:

- *Minimum phase filters.* These often appear in voice applications where minimum signal delay is an issue. The constraint here is that the zeros of the transfer function must remain inside the unit circle.

- *Recursive filters with linearized phase.* If high selectivity and low hardware expense are requirements in data applications, then IIR-filters with linearized phase are often the best choice. Phase linearization is traditionally done with all-pass filters, but direct design in the z-domain generally yields a lower filter order.

- *Recursive filters with pole radius restrictions.* Such restrictions may apply to reduce the sensitivity to coefficient quantization or to get a short impulse response length. The latter can be important in communications applications like modems.

- *Constraints in the frequency and time domains.* A typical application is filters for ISDN (Integrated Services Digital Network) equipment where constraint masks are defined for the time and frequency domains.

- *Magnitude constraints that deviate from the standard LP, HP, BP, BS type.* Filters with "odd-looking" magnitude requirements occur fairly often in the DSP engineer's world. Sinc-compensated filters, filters that are supposed to remove specific unwanted tones in addition to another filter task (e.g. a lowpass with a notch), and differentiators are common examples.

- *Postfilters.* Some developments have to work in conjunction with existing HW, be it analog or digital. Additional filtering (e.g., to improve certain behavior) must take the already-existing filter into account in order to meet an overall filter specification.

Such designs are often resolved by resorting to filter design experts and/or expensive filter design software, both of which can incur substantial costs. There is a simpler and less expensive way, however, that offers a solution to many unusual filter design tasks. It uses the power of a genetic algorithm called *differential evolution* (DE), which is still not widely known in the signal processing community [4]. The approach is to recast the filter design problem as a minimization problem and use the poles and zeros as parameters. In this approach the ensuing error function must be minimized or ideally be driven to zero.

Filter design problems that are recast as minimization problems generally yield multimodal results that are very difficult to minimize. Yet DE is powerful enough to successfully attack even these types of error functions. The approach that will be described in the following has even found its way into a commercial product [5].

## 3.1 RECASTING FILTER DESIGN AS A MINIMIZATION PROBLEM

The most general form of a digital filter is

$$H(z) = \frac{U(z)}{D(z)} = \frac{\sum\limits_{n=0}^{N_z} a(n) \cdot z^{-n}}{1 + \sum\limits_{m=1}^{M_p} b(m) \cdot z^{-m}} = A_0 \frac{\prod\limits_{n=0}^{N_z-1} (z - z_0(n))}{\prod\limits_{m=0}^{M_p-1} (z - z_p(m))} \tag{3-1}$$

The degree of $H(z)$ is defined as the maximum of $N_z$ and $M_p$. The parameters $a(n)$ and $b(m)$ are called the *coefficients* of the filter while the $z_0(n)$ and $z_p(m)$ denote the zeros and poles of the filter respectively.

The idea of treating filter design as a minimization problem is not new [6]. How we approach the minimization here, however, is radically different from the classical methods. While classical methods resort to calculus-based minimization algorithms like Newton and Quasi-Newton methods rendering a complicated system of equations to solve, a very simple yet powerful genetic algorithm called differential evolution (DE) will be applied. Before explaining how DE works, let us formulate our
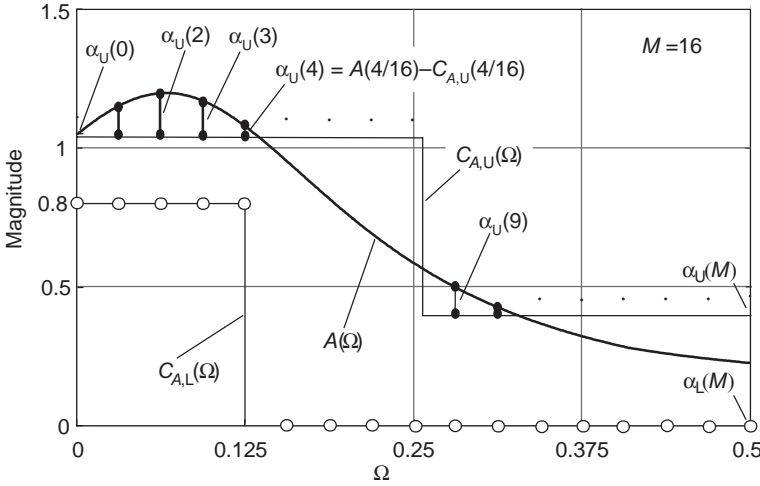
**Figure 3-1**    Example tolerance scheme for the magnitude $A(\Omega)$.

problem. In digital filter design the frequency magnitude response $A(\Omega)$, the phase angle $\varphi(\Omega)$, and the group delay $G(\Omega)$ are important quantities. Some or all of the quantities $A(\Omega)$, $\varphi(\Omega)$, and $G(\Omega)$ may be subject to certain constraints in a filter design problem. $A(\Omega)$ is used here as an example to explain the principles of a constraint-based design.

Upper and lower constraints $C_{A,U}(\Omega)$ and $C_{A,L}(\Omega)$ define a tolerance scheme as shown in Figure 3-1.

What is of interest is how much $A(\Omega)$ violates the tolerance scheme in the frequency domain. The task is to define an error function $e_A$ that becomes larger the more $A(\Omega)$ violates the tolerance scheme and that becomes zero when $A(\Omega)$ satisfies the constraints. It is straightforward to define the area outside the tolerance scheme enclosed by $A(\Omega)$ as the error. However, it is computationally simpler and more efficient to use the sum of squared error samples $\alpha_U(m)$, $\alpha_L(m)$ as the error function $e_A$.

If $e_A$ is zero, then all constraints are met, so the *stopping criterion* for the minimization is clearly defined as $e_A = 0$. Optimization problems that are defined by constraints only and are solved once all constraints are met are called *constraint satisfaction problems*. For other quantities like, for example, the group delay, the same principle can applied to build $e_G$. If, for example, constraints exist for both magnitude and group delay the stopping criterion may be defined as $e_{\text{total}} = e_A + e_G = 0$.

The principle of just adding error terms that define the constraints to be met can be applied for all constraints levied on the filter design. Among these may be phase constraints, constraints to keep the poles inside the unit circle (stability), keeping the zeros inside the unit circle (minimum phase), and so on. Note that the partial error terms need not be weighted in order to end up with a successful design. DE has enough optimization quality to not require weights.

The natural parameters to vary in order to drive the total error to zero are either the coefficients $a(n)$ and $b(m)$ or the radii and angles of the zeros $z_0(n)$ and the poles $z_p(m)$, along with the gain factor $A_0$. While varying the coefficients may seem attractive at first, because they represent what is finally needed to implement a filter, the trick here is that working with the poles and zeros is much more convenient. The primary reason is that it is much simpler to ensure stability this way.

For each iteration of the minimization a stability criterion has to be checked—at least for IIR filters. If the parameters are zeros and poles, one has just to prevent the poles from moving outside the unit circle. If the coefficients are used as parameters, the stability criterion is more computationally intensive because it requires some knowledge about the roots of $H(z)$. Using poles and zeros as parameters also simplifies the design of minimum phase filters, for example, to ensure that the zeros stay inside the unit circle.

## 3.2  MINIMIZATION WITH DIFFERENTIAL EVOLUTION

In order to adjust the parameters $z_0(n)$ and $z_p(m)$ such that the error $e_{total}$ becomes zero we use the differential evolution (DE) method. DE belongs to the class of direct search minimization methods [4]. These methods are in contrast with gradient-based methods, which rely heavily on calculus and usually lead to rather complicated system equations [6]. Gradient-based methods also require that the function to be minimized can always be differentiated. This requirement cannot be fulfilled, for example, when coefficient quantization is incorporated into the minimization.

Direct search minimization methods act a bit like a person in a boat on a lake who wants to find the deepest spot of the lake. The person cannot see to the bottom of the lake so he has to use a plumbline to probe the deepness. If the bottom of the lake has several valleys of varying depth, it becomes difficult to find the deepest spot.

DE operates according to the following scenario. $N_P$ base points are spread over the the *error function surface* to initialize the minimization. This is shown by the example in Figure 3-2(a) that depicts the contour lines of a multimodal function (called *peaks function*) and the initialization points chosen for DE. Each dot represents a candidate solution for the values of parameters $p1$ and $p2$ (the axes in Figure 3-2) and we seek to find the values of those parameters that minimize the error function. After initialization each starting point is perturbed by a random weighted difference vector $V$ and its error value is compared with the error value of another randomly selected point of the population. The perturbed point with the lower error value wins in this pairwise competition. This scheme is repeated from generation to generation.

The difference vector $V$ is built from two other randomly chosen points of the population itself. It is then weighted with some constant factor $F$ before it is used for perturbation. For each point to be perturbed a new difference vector defined by two randomly selected points of the population is chosen. As a rule of thumb $F$ is chosen around 0.85. Figure 3-2(a) illustrates this idea. Once each point of the popu-
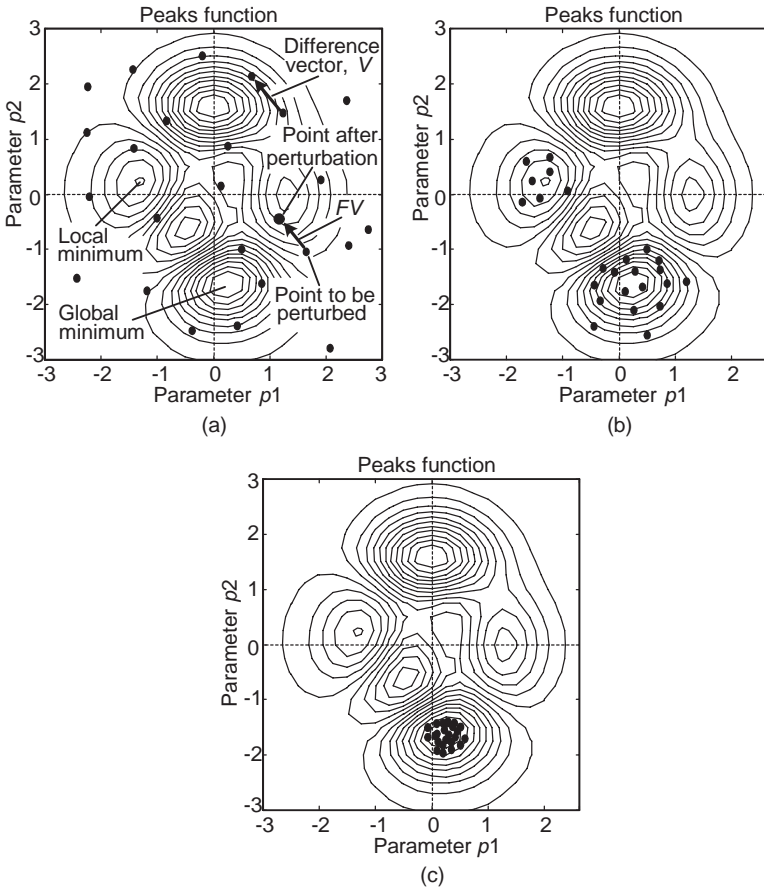
**Figure 3-2**   The convergence behavior of DE.

lation is perturbed, the perturbed points compete against the original points. Each new point has to compete against a distinct old point. The winner of each match is the point that has the lower error function value. Eventually a new population of points has emerged that is ready for the next iteration. Because the competition is a 1-to-1 competition, the number of population points (population vectors) is the same in each iteration.

Figures 3-2(b) and (c) show how the point population converges toward the global minimum. One can also see that some points stay at a local minimum for a while, but finally they have to vanish because they lose in the competition against points that are in the vicinity of the true global minimum. Now it becomes evident why the use of difference vectors for perturbation is advantageous: While the population is moving toward the global minimum the difference vectors become shorter, which is exactly what should happen, and their directions adapt to the error function surface at hand. This is the great advantage of basing perturbation on the population itself rather than applying a predefined probability density function (e.g., gaussian), the parameters of which (e.g., standard deviation) are not straightforward to choose.

All mathematical details of differential evolution, along with computer source code in C, MATLAB®, and other programming languages, can be found in [4].

## 3.3 A DIFFERENTIAL EVOLUTION DESIGN EXAMPLE

Now it is time to look at an example to show that the DE design method really works. The following example has been computed by the filter design program FIWIZ, which is a commercial program that, among others, employs the above method for its filter design tasks [5]. Figure 3-3 shows a lowpass tolerance scheme, the dotted curves, needed for a graphics codec that has tight constraints in both the magnitude response (in dB) and group delay [7]. The $y$-axis of the group delay plot is scaled in units of the sampling time $T_s$. The tolerance constraint for the group delay may be shifted along the $y$-axis. In addition, the coefficient wordlength shall be constrained to 24 bits and biquad stages shall be used for filter realization. Figure 3-3 shows a fifth-order design that uses an elliptical filter approach being able to satisfy the magnitude constraints. The severe group delay constraints, however, are not met.

However, DE offers much more flexibility so that group delay constraints can be considered during the filter design. The result, a twelfth-order IIR filter, can be
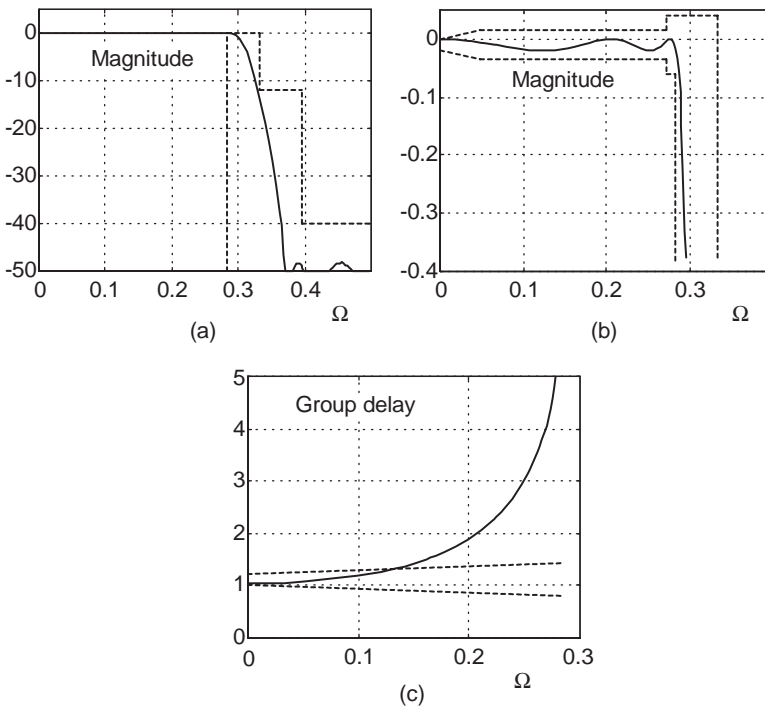


**Figure 3-3**    Magnitude and group delay constraints for an IIR lowpass filter.
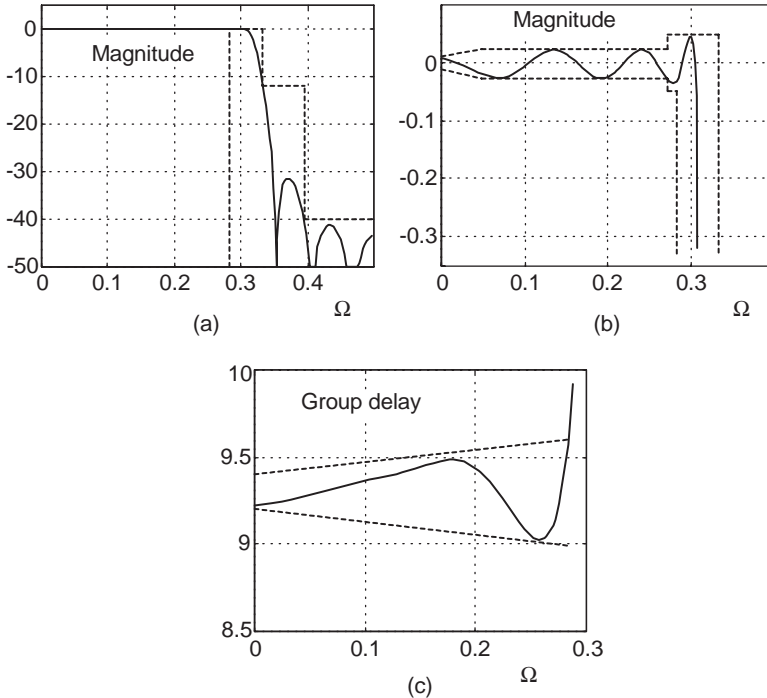
**Figure 3-4**    DE-designed twelfth-order IIR filter performance.

seen as the performance shown in Figure 3-4. Due to forcing the zeros outside the unit circle this design was able to be finished in 130,500 iterations (roughly 18 minutes on a Pentium III 650 MHz PC). Despite the large number of 37 parameters a value of $N_P = 30$ was sufficient. We see that the DE-designed IIR filter satisfies the stringent group delay constraints.

## 3.4   CONCLUSIONS

In this chapter an alternative method for nonstandard filter design has been described. This method recasts the filter design problem as a minimization problem and solves the minimization via the DE minimizer, for which public domain software has been made available in [8]. The advantages of this method are its simplicity as well as the capability to design unconventional filter types. A great asset of this approach is that it can be applied with a minimal knowledge of digital filter design theory.

## 3.5   REFERENCES

[1]  "Digital Filter Design Software." [Online: http://www.dspguru.com/sw/tools/filtdsn2.htm.]
[2]  "GUI: Filter Design and Analysis Tool (Signal Processing Toolbox)." [Online: http://www.mathworks.com/access/helpdesk/help/toolbox/signal/fdtoo11a.html.]

[3]  "Filter Design Software." [Online: http://www.poynton.com/Poynton-dsp.html.]

[4]  K. PRICE, R. STORN, and J. LAMPINEN, *Differential Evolution—A Practical Approach to Global Optimization*. Springer, 2005.

[5]  "Digital Filter Design Software Fiwiz." [Online: http://www.icsi.berkeley.edu/~storn/fiwiz.html.]

[6]  A. ANTONIOU, *Digital Filters—Analysis, Design, and Applications*. McGraw-Hill, 1993.

[7]  R. Storn, "Differential Evolution Design of an IIR-Filter with Requirements for Magnitude and Group Delay," *IEEE International Conference on Evolutionary Computation ICEC 96*, pp. 268–273.

[8]  "Differential Evolution Homepage." [Online: http://www.icsi.berkeley.edu/~storn/code.html.]

# Chapter 4

# Designing IIR Filters with a Given 3 dB Point

**Ricardo A. Losada**
*The MathWorks, Inc.*

**Vincent Pellissier**
*The MathWorks, Inc.*

**O**ften in IIR filter design our critical design parameter is the cutoff frequency at which the filter's power decays to half (−3 dB) the nominal passband value. This chapter presents techniques that enable designs of discrete-time Chebyshev and elliptical filters given a 3 dB attenuation frequency point. These techniques place Chebyshev and elliptic filters on the same footing as Butterworth filters, which traditionally have been designed for a given 3 dB point. The result is that it is easy to replace a Butterworth design with either a Chebyshev or an elliptical filter of the same order and obtain a steeper rolloff at the expense of some ripple in the passband and/or stopband of the filter.

We start by presenting a technique that solves the problem of designing discrete-time Chebyshev type I and II IIR filters given a 3 dB attenuation frequency point. Traditionally, to design a lowpass Chebyshev (type I) IIR filter we start with the following set of desired specifications: $\{N, \omega_p, A_p\}$. $N$ is the filter order, $\omega_p$ the passband-edge frequency, and $A_p$ is the desired attenuation at $\omega_p$ (see Figure 4-1). The problem is that it's impractical to set $A_p = 3$ dB and design for the specification set $\{N, \omega_p, 3\}$; due to the filter's equiripple behavior, all the ripples in the passband would reach the −3 dB point, yielding intolerable passband ripple.

To solve this problem, our designs are based on analytical relations that can be found in the analog domain between the passband-edge frequency and the 3 dB cutoff frequency in the case of type I Chebyshev filters and between the
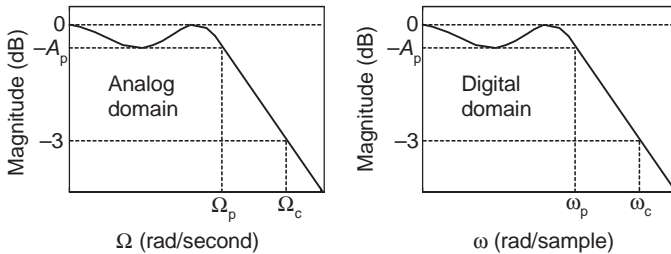
**Figure 4-1** Definition of analog/digital frequency parameters.

stopband-edge frequency and the 3 dB cutoff frequency in the case of type II Chebyshev filters. We use the inverse bilinear transformation in order to map the specifications given in the digital domain into analog specifications. We then use the analytical relation between the frequencies we have mentioned above to translate a set of specifications into another one we know how to handle, and finally we use the bilinear transformation to map the new set of specifications we can handle back to the digital domain.

In the case of highpass, bandpass, and bandstop filters, we show a trick where we use an arbitrary *prototype* lowpass filter, along with the Constantinides spectral transformations, in order to build on what was done in the lowpass case and allow for the design of these other types of filters with given 3 dB points.

We then turn our attention to elliptic filters. For this case, we present a simple technique using the Constantinides spectral lowpass-to-lowpass transformation on a halfband elliptic filter to obtain the desired filter.

We will use $\Omega$ to indicate analog frequency (radians per second), and a set such as $\{N,\Omega_c,A_p\}$ to indicate analog design specifications. Normalized (or "digital") frequency is denoted by $\omega$ (radians per sample or simply radians), and we will use a set such as $\{N,\omega_c,A_p\}$ to indicate discrete-time or digital design specifications. The various analog and digital frequency-domain variables used in this chapter are illustrated in Figure 4-1. Example design specifications are described below. The lowpass case is described first and then the highpass, bandpass, and bandstop cases are developed.

## 4.1 LOWPASS CHEBYSHEV FILTERS

For the lowpass Chebyshev filter case, to solve the 3 dB passband ripple problem cited earlier, we can translate a given set of specifications that include the cutoff frequency $\omega_c$ to the usual set $\{N,\omega_p,A_p\}$ in order to use the existing algorithms to design the filter. This amounts to exchanging one parameter in the specifications set with $\omega_c$. So the given specification set for the design would be $\{N,\omega_c,A_p\}$ or $\{N,\omega_p,\omega_c\}$. (We concentrate here on designs with a fixed filter order. It would be also possible to translate the set $\{\omega_p,\omega_c,A_p\}$ to $\{N,\omega_p,A_p\}$ but it would require rounding the required

order to the next integer, thereby exceeding one of the specifications.) We will show how to do this, but first we show how to find the 3 dB point in an analog Chebyshev type I lowpass filter given $\{N, \Omega_p, A_p\}$.

## 4.2 DETERMINING THE 3 DB POINT OF AN ANALOG CHEBYSHEV TYPE I FILTER

Analog Chebyshev type I lowpass filters have a magnitude squared response given by

$$|H(j\Omega)|^2 = \frac{1}{1 + \varepsilon_p^2 C_N^2 \left( \dfrac{\Omega}{\Omega_p} \right)}$$

where the quantity $\varepsilon_p$ controlling the passband ripples is related to the passband attenuation $A_p$ by

$$\varepsilon_p = \sqrt{10^{A_p/10} - 1} \tag{4–1}$$

and the Chebyshev polynomial of degree $N$ is defined by

$$C_N(x) \begin{cases} \cos(N\cos^{-1}(x)) & \text{if } |x| \le 1 \\ \cosh(N\cosh^{-1}(x)) & \text{if } |x| > 1 \end{cases}$$

For lowpass filters it is reasonable to assume $\Omega_c/\Omega_p > 1$, therefore we can determine the 3 dB frequency by determining $\varepsilon_p$ from (4–1) and finding the point at which the magnitude squared is equal to 0.5 by solving for $\Omega_c$ in

$$\frac{1}{2} = \frac{1}{1 + \varepsilon_p^2 C_N^2 \left( \dfrac{\Omega_c}{\Omega_p} \right)}$$

Solving for $\Omega_c$, it has been shown in [1] that we get

$$\Omega_c = \Omega_p \cosh\left( \frac{1}{N} \cosh^{-1}\left( \frac{1}{\varepsilon_p} \right) \right) \tag{4–2}$$

## 4.3 DESIGNING THE LOWPASS FILTER

Now that we have derived the relation between $\Omega_c$ and $\Omega_p$, we outline the design of the digital Chebyshev filter. The design process consists of three steps:

1. Given $\{N, \omega_c, A_p\}$ or $\{N, \omega_p, \omega_c\}$, determine $\{N, \Omega_c, A_p\}$ or $\{N, \Omega_p, \Omega_c\}$ as appropriate by prewarping the digital frequencies using $\Omega = \tan(\omega/2)$ (inverse bilinear transformation).

2. Using (4–1) and (4–2), find the missing parameter, either $\Omega_p$ or $A_p$.
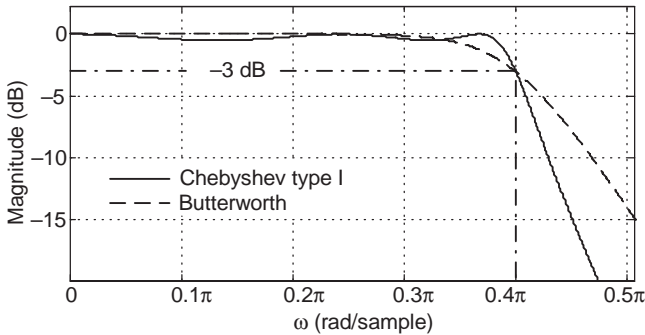
**Figure 4-2**    A Chebyshev filter with $\{N=5, \omega_c=0.4\pi, A_p=0.5\,\text{dB}\}$. For comparison purposes, a Butterworth filter with $\{N=5, \omega_c=0.4\pi\}$ is shown.

    **3.** If the missing parameter was $\Omega_p$, determine $\omega_p$ from $\omega_p=2\tan^{-1}(\Omega_p)$ (bilinear transformation).

At this point we should have the specification set $\{N,\omega_p,A_p\}$, so we can use existing design algorithms readily available in filter design software packages (such as the cheby1 command in MATLAB) to design the Chebyshev type I lowpass filter.

### EXAMPLE 1

Suppose we want to design a filter with the following specification set: $\{N=5, \omega_c=0.4\pi, A_p=0.5\,\text{dB}\}$. We can use the technique outlined above to translate the cutoff frequency $\omega_c=0.4\pi$ to the passband-edge frequency $\omega_p=0.3827\pi$. The design is shown in Figure 4-2. A Butterworth filter of the same order and same 3 dB point is shown for reference. This example illustrates the application of our technique. It allows us to easily keep the same cutoff frequency as a Butterworth design, yet allowing some passband ripple as a trade-off for a steeper rolloff.

## 4.4  CHEBYSHEV TYPE II FILTERS

A similar approach can be taken for Chebyshev type II (inverse Chebyshev) filters. However, in this case it is a matter of translating the specification sets $\{N,\omega_c,A_s\}$ or $\{N,\omega_c,\omega_s\}$ ($\omega_s$ is the stopband-edge frequency and $A_s$ is the stopband attenuation) to $\{N,\omega_s,A_s\}$ and then using filter design software (such as the cheby2 command in MATLAB) to obtain the filter coefficients. To do this, we need to use the expression for the magnitude squared of the frequency response of Chebyshev type II analog frequencies, described in [1], in order to determine $\Omega_c$ in terms of $\Omega_s$:

$$|H(j\Omega)|^2 = \frac{C_N^2\left(\dfrac{\Omega_s}{\Omega}\right)}{\varepsilon_s^2 + C_N^2\left(\dfrac{\Omega_s}{\Omega}\right)}$$
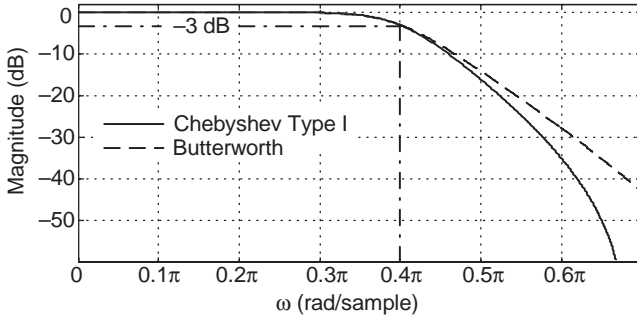
**Figure 4-3**    A Chebyshev type II filter with $\{N=5,\ \omega_c=0.4\pi,\ A_s=60\,dB\}$. For comparison purposes, a Butterworth filter with $\{N=5,\ \omega_c=0.4\pi\}$ is shown.

Once again, setting the magnitude squared to 0.5 and solving for $\Omega_c$ we obtain:

$$\Omega_c = \frac{\Omega_s}{\cosh\left(\dfrac{1}{N}\cosh^{-1}(\varepsilon_s)\right)}$$

where the quantity $\varepsilon_s$ controlling the depth of the stopband is related to the stopband attenuation $A_s$ by

$$\varepsilon_s = \sqrt{10^{A_s/10}-1}$$

**EXAMPLE 2**

Chebyshev type II filters provide a good way of attaining a very similar response to Butterworth in the passband, but with a sharper rolloff for a fixed filter order. Using the technique we have described we can design a Chebyshev type II filter with the following specifications $\{N=5,\ \omega_c=0.4\pi,\ A_s=60\,dB\}$. The result is shown in Figure 4-3. Again, we include the same Butterworth filter response from the previous example.

## 4.5   HIGHPASS, BANDPASS, AND BANDSTOP CHEBYSHEV FILTERS

The relation between the analog cutoff frequency and the analog passband-edge or stopband-edge frequency we used in the previous section is valid only for lowpass filters. In order to design highpass and other types of filters, we can build on what we have done for lowpass filters and use the Constantinides frequency transformations in [2],[3]. For instance, given the highpass specifications $\{N,\omega_c',\omega_p'\}$ or $\{N,\omega_c',A_p\}$, we need to translate the desired cutoff frequency $\omega_c'$ to either $A_p$ or $\omega_p'$ in order to use filter design software to design for the usual set $\{N,\omega_p',A_p\}$.

The trick consists in taking advantage of interesting properties of the Constantinides frequency transformations. The passband ripples are conserved and the transformation projecting the lowpass cutoff frequency $\omega_c$ to the highpass cutoff

frequency $\omega'_c$ will also project the lowpass passband-edge frequency $\omega_p$ to the high-pass passband-edge frequency $\omega'_p$. The problem comes down to determining the lowpass specification set $\{N,\omega_c,\omega_p\}$ that designs a filter that when transformed meets the given highpass specifications $\{N,\omega'_c,\omega'_p\}$. We never actually design the lowpass filter, however.

The full procedure is as follows:

1. Given the highpass specifications $\{N,\omega'_c,\omega'_p\}$, choose *any* value for the cutoff frequency $\omega_c$ of a digital lowpasss filter.

2. Knowing $\omega_c$ and $\omega'_c$, find the parameter $\alpha$ used in the lowpass-to-highpass Constantinides transformation:

$$\alpha = -\frac{\cos\left(\dfrac{\omega_c + \omega'_c}{2}\right)}{\cos\left(\dfrac{\omega_c - \omega'_c}{2}\right)} \tag{4-3}$$

3. Given the desired passband-edge frequency $\omega'_p$ of the highpass filter and $\alpha$, determine the passband-edge frequency $\omega_p$ of the digital lowpass filter from

$$z^{-1} = \frac{z'^{-1} + \alpha}{1 + \alpha z'^{-1}} \tag{4-4}$$

evaluated at $z = e^j\omega_p$ and $z' = e^j\omega'_p$.

4. At this point, we have the lowpass specifications $\{N,\omega_c,\omega_p\}$. With this, we can determine $A_p$ as we have seen in the previous section to translate the specifications set to $\{N,\omega_c,A_p\}$.

5. $A_p$ being conserved by the frequency transformation, we can now substitute $A_p$ for $\omega'_c$ in the highpass specification set and use the new set $\{N,\omega'_p,A_p\}$ to design highpass Chebyshev type I filters with filter design software.

Notice that the procedure is similar if the highpass specification set is $\{N,\omega'_c,A_p\}$. Once again we choose a value for $\omega_c$. We now have the lowpass specification set $\{N,\omega_c,A_p\}$; we can now translate to the lowpass specification set $\{N,\omega_p,A_p\}$ as seen previously and, similar to what is done above, determine $\omega_p$ through the Constantinides frequency transformations to end up with the highpass specification set $\{N,\omega'_p,A_p\}$.

For bandpass and bandstop filters, we need to halve the order when we convert from the original specifications to the lowpass specifications since this is implicit in the Constantinides transformation, but other than that the procedure is basically the same.

For instance, in the bandpass case, say we start with the specification set $\{N',\omega'_{c1},\omega'_{c2},A_p\}$ where $\omega'_{c1},\omega'_{c2}$ are the lower and upper 3 dB frequencies respectively. We once again choose any value for $\omega_c$, the cutoff frequency of the lowpass filter. Now we have the lowpass specification set $\{N,\omega_c,A_p\}$ where $N=N'/2$. We translate that set to the lowpass specification set $\{N,\omega_p,A_p\}$. We can then use the lowpass to bandpass transformation (see [2] or [3]) and find the two solutions (since the trans-
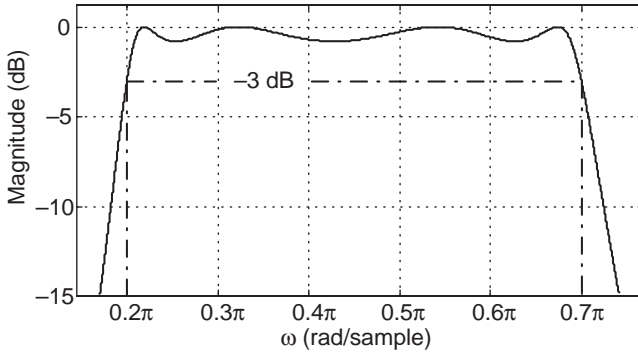
**Figure 4-4**    A Chebyshev type I bandpass filter with $\{N'=8, \omega'_{c1}=0.2\pi, \omega'_{c2}=0.7\pi, \omega'_{p2}-\omega'_{p1}=0.48\pi\}$.

formation is a quadratic) for $\omega'$ when $\omega=\omega_p$. These two solutions are $\omega'_{p1}, \omega'_{p2}$, the lower and higher passband-edge frequencies. We end up with the specification set $\{N', \omega'_{p1}, \omega'_{p2}, A_p\}$, which our design algorithm can accommodate to obtain the bandpass filter that has an attenuation of $A_p$ at the passband-edges $\omega'_{p1}, \omega'_{p2}$ and has an attenuation of 3 dB at $\omega'_{c1}, \omega'_{c2}$ .

If the original bandpass specification set we have is $\{N', \omega'_{c1}, \omega'_{c2}, \omega'_{p1}, \omega'_{p2}\}$, we can again choose $\omega_c$ and using the lowpass-to-bandpass transformation and $\omega'_{p1}, \omega'_{p2}$ compute $\omega_p$ and then determine $A_p$. Note that in this case we don't really need $\omega'_{p1}, \omega'_{p2}$ but only the difference between them, $\omega'_{p2}-\omega'_{p1}$, since this is all that is needed to compute the lowpass-to-bandpass transformation.

### EXAMPLE 3

We design a Chebyshev type I bandpass filter with the following set of specifications: $\{N'=8, \omega'_{c1}=0.2\pi, \omega'_{c2}=0.7\pi, \omega'_{p2}-\omega'_{p1}=0.48\pi\}$. The results are shown in Figure 4-4.

## 4.6  ELLIPTICAL FILTERS

In the elliptical filter case, a simple way to obtain a desired 3 dB point is to design a (lowpass) halfband elliptic filter and use the Constantinides spectral transformations to obtain a lowpass, highpass, bandpass, or bandstop filter with its 3 dB point located at a desired frequency.

Halfband IIR filters have the property that their 3 dB point is located exactly at the halfband frequency, $0.5\pi$. We can use this frequency as the originating point in the Constantinides spectral transformations. The destination point will be the final frequency where a 3 dB attenuation is desired.

It is worth noting that passband and stopband ripple of a halfband IIR filter are related. Therefore, it is not possible to have independent control over each when we design such a filter. However, it turns out that for even relatively modest stopband attenuations, the corresponding passband ripple of a halfband IIR filter is extremely
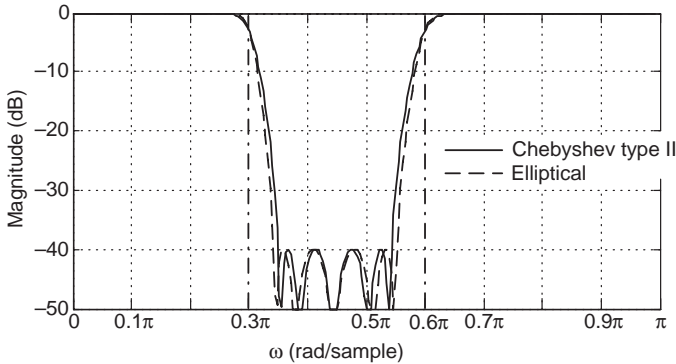
**Figure 4-5**    A Chebyshev type II and an elliptical bandstop filter with {N = 10, $\omega'_{c1}$ = 0.3π, $\omega'_{c2}$ = 0.6π}.

small. As an example, for a stopband attenuation of only 40 dB, the passband ripple is about $4 \times 10^{-4}$ dB. With such a small passband ripple, elliptical filters designed using the technique outlined here are an attractive alternative to Chebyshev type II designs. The small passband ripple is of little consequence in many applications; however, it allows for a steeper transition between passband and stopband than a comparable Chebyshev type II filter.

---

**EXAMPLE 4**

We design a fifth-order elliptical halfband filter with a stopband attenuation of 40 dB. We then transform this filter to a tenth-order bandstop filter with lower and upper 3 dB points given by 0.3π and 0.6π. The results are shown in Figure 4-5. For comparison purposes, a tenth-order Chebyshev type II bandstop filter with the same 40 dB of attenuation is shown. Notice the steeper transitions from passband to stopband in the elliptical filter.

If full control of both passband and stopband ripples is desired for elliptical designs, a solution that allows for such control along with control over the location of the 3 dB point has been given in [4].

## 4.7    CONCLUSIONS

The design of IIR filters with a given 3 dB frequency point is common in practical applications. We presented ways of designing lowpass, highpass, bandpass, and bandstop Chebyshev type I and II filters that meet a 3 dB constraint. The methodology presented uses bilinear transformations, analytical expressions for the magnitude squared response of lowpass analog Chebyshev type I and II filters, and—in the case of highpass, bandpass, and bandstop filters—the Constantinides spectral transformations.

In the case of elliptical filters, we presented an approach based on the design of halfband elliptical filters and the use of the Constantinides spectral transformations that is a viable alternative to Chebyshev type II designs. We also included a

reference to a more general solution that can be used if the halfband approach is inadequate.

The designs attainable with the techniques presented here can be used to substitute practical Butterworth designs by trading off some passband/stopband ripple for better transition performance while meeting the same 3 dB specification. The differences in the phase behavior between Butterworth, Chebyshev, and elliptical filters should be brought into account when phase is an issue. The techniques presented here are robust and applicable to a wide range of IIR filter design specifications; they have been included in the Filter Design Toolbox for MATLAB.

## 4.8  REFERENCES

[1] S. ORFANIDIS, *Introduction to Signal Processing*. Prentice Hall, Upper Saddle River, NJ, 1996.

[2] A. CONSTANTINIDES, "Spectral Transformations for Digital Filters," *Proc. IEE*, 117, August 1970, pp. 1585–1590.

[3] S. MITRA, *Digital Signal Processing: A Computer-Based Approach*, 2nd ed. McGraw-Hill, New York, 2001.

[4] S. ORFANIDIS, "High-Order Digital Parametric Equalizer Design," *J. Audio Eng. Soc.*, vol. 53, November 2005, pp. 1026–1046.

# Chapter 5

# Filtering Tricks for FSK Demodulation

**David Shiung**

*MediaTek Inc.*

**Huei-Wen Ferng**

*National Taiwan Univ. of Science and Technology*

**Richard Lyons**

*Besser Associates*

**I**n the past decades, economical implementations of digital systems have always been appealing research topics. In this chapter we present a useful trick used to make the implementation of a digital noncoherent frequency shift keying (FSK) demodulator more economical from a hardware complexity standpoint, with the goal of minimizing its computational workload.

## 5.1  A SIMPLE WIRELESS RECEIVER

The RF front-end of a wireless receiver, shown in Figure 5-1, is mainly designed using analog components. The information of a noncoherent FSK signal is conveyed at zero-crossing points during each bit period. One often uses a limiter rather than an analog-to-digital converter (ADC) for the sake of hardware simplicity because the output of a limiter contains only two states and functionally behaves as a one-bit ADC. Thus the signal at node B is a sequence of ones and zeros [1]. In the following, we shall call the signal at node A the intermediate-frequency (IF) signal.

Although a variety of receiver architectures (e.g., low-IF and zero IF receivers) exist, we use the superheterodyne system as an example [2]. Owing to fabrication
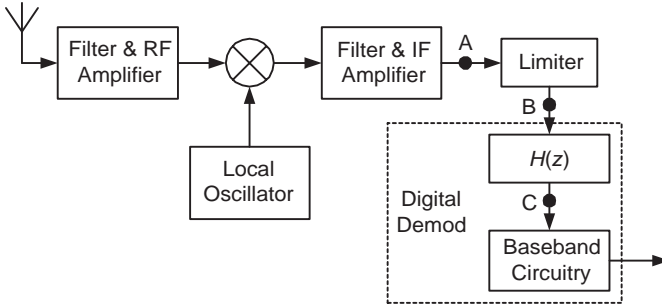
**Figure 5-1**    A noncoherent FSK demodulator preceded by a limiter and RF front-end circuitry.
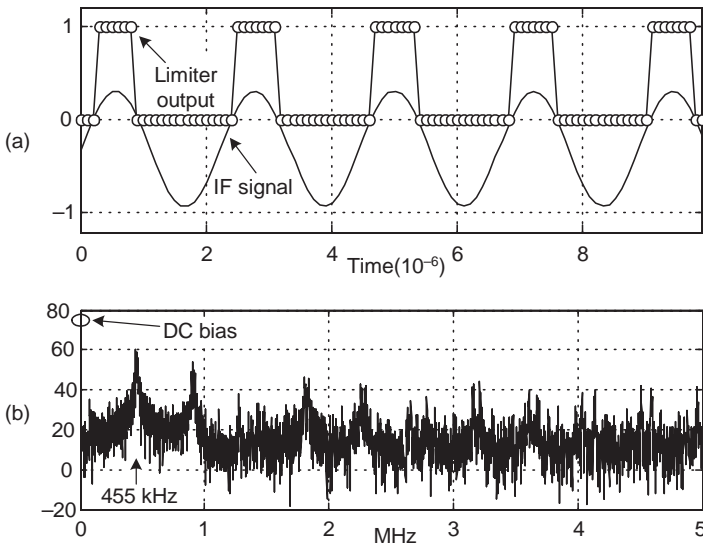


**Figure 5-2**    Limiter signals: (a) limiter IF input and binary output; (b) output spectrum.

variation, the duty cycle of the IF signal is frequently not equal to 50% [2], [3]. That is, the time duration of pulse "1" is not equal to that of pulse "0". The curve in Figure 5-2(a) shows our analog IF signal and the discrete samples in the figure show the binary sequence at the output of the limiter (node B in Figure 5-1).

In our application, the $f_s$ sampling frequency of the digital demodulator is 10 MHz; the IF signal is located at frequency $f_I = 455$ kHz, the frequency deviation $f_D$ is set to ±4 kHz; the data rate $R$ is set to 4 kbps; and a 30% duty cycle is assumed. The signal assumed in this example has a modulation index of 2 (i.e., $|2f_D/R| = 2$) so that the best bit error rate (BER) performance may be achieved for many popular FSK demodulators [4].

Examining the spectra of limiter output sequence in Figure 5-2(b) shows that there are harmonics located at integral multiples of $f_I = 455$ kHz. In addition to those

harmonics, there is a very large DC bias (zero-Hz spectral component) on the limiter output signal that must be eliminated. The primary DSP trick in this discussion is using a comb filter to solve this DC bias (DC offset) problem, which, in addition, will also help us filter the limiter output signal's harmonics.

## 5.2  USING A COMB FILTER

The comb filter we use to eliminate the DC bias and minimize follow-on filter complexity is the standard $N$-delay comb filter shown in Figure 5-3(a) [5]. Its periodic passbands and nulls are shown in Figure 5-3(b) for $N=8$.

The $z$-domain transfer function of $K$ cascaded comb filters is

$$H(z) = (1 - z^{-N})^K \qquad (5\text{--}1)$$

where in our application $K=1$.

The following two equations allow us to locate the local minimum ($L_{\min}$) and local maximum ($L_{\max}$) in the positive-frequency range of $|H(\omega)|$.

$$L_{\min}: \omega = \frac{2\pi}{N} \cdot i, \quad i = 0, 1, \ldots, \left\lfloor \frac{N}{2} \right\rfloor \qquad (5\text{--}2)$$

$$L_{\max}: \omega = \frac{\pi}{N} + \frac{2\pi}{N} \cdot i, \quad i = 0, 1, \ldots, \left\lfloor \frac{N-1}{2} \right\rfloor, \qquad (5\text{--}3)$$

where $\lfloor q \rfloor$ means the integer part of $q$. Notice that these results are valid only for $N \geq 2$. If we regard the locations of $L_{\max}$ as passbands and treat $L_{\min}$ as stopbands of $H(z)$, the comb filter can be viewed as a multiband filter.

In general, the coefficient for each tap of an FIR filter is in floating-point format. Thus, multipliers and adders are generally required to perform signal filtering. However, $H(z)$ needs no multiplier and requires only an adder and a few registers. If the input sequence to such a filter is binary, as in our application, the adder of the comb filter becomes a simple XOR logic operation!

If there exists unwanted interference near DC, we may use multiple comb filters in cascade to further suppress these interfering signals. This may sometimes occur when low-frequency noise is present. However, the price paid in this case is non-trivial adders and multipliers. Some approaches have been proposed to simplify the hardware [6].
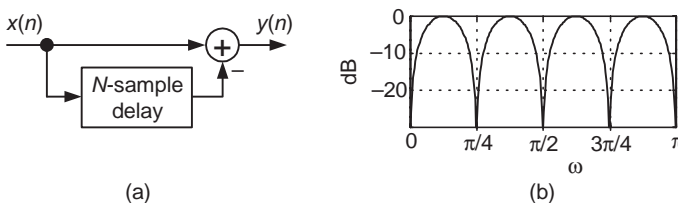


**Figure 5-3**    An $N=8$ comb filter: (a) structure; (b) response $|H(\omega)|$.
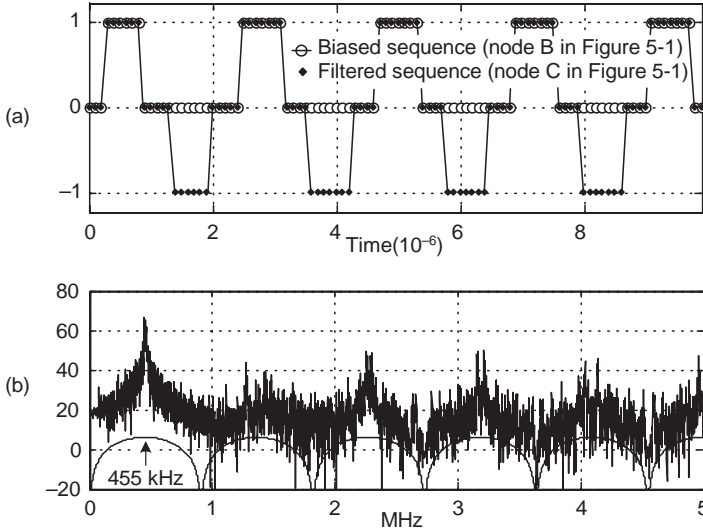
**Figure 5-4**    DC component removal for a binary IF signal using $H(z)$ with $N=11$.

If the parameter $N$ of $H(z)$ is carefully chosen such that $2\pi f_I/f_s$ is located at the first $L_{max}$ of the comb filter's magnitude response, all harmonics located at even multiples of $f_I$ and the zero Hz (DC) component of the hard-limited signal applied to the comb filter can be suppressed. This operation is very helpful in reducing the hardware complexity of the following circuitry and enhancing the system performance of the noncoherent FSK receiver.

The signals into and out of the $H(z)$ comb filter, when $N=11$, are shown in Figure 5-4(a). Examining the spectra of the filtered output signal, in Figure 5-4(b), we find that only our signal of interest at 455 kHz and odd harmonics of the fundamental frequency $f_I$ exist. For reference, the comb filter's $|H(\omega)|$ is shown in Figure 5-4(b). Since no low-frequency interference is assumed, we use $H(z)$, whose frequency response is superimposed in Figure 5-4(b), to suppress the DC offset of the IF signal. Comparing Figure 5-2(b) and Figure 5-4(b), it is found that even harmonics and the DC component of the input signal are all diminished to an unnoticeable degree. The odd harmonics are attenuated by follow-on filters where their computational workload is reduced because of the comb filter. Note that this efficient comb filtering also works for an M-ary FSK signaling by properly choosing the system parameters (e.g., $f_I, f_s, f_D, R$, and $N$).

## 5.3   BENEFICIAL COMB FILTER BEHAVIOR

The locations of the comb filter output harmonics remain unchanged even if the duty cycle of the limiter output curve in Figure 5-2(a) changes. The length of the comb filter ($N$) is determined only by the sampling frequency $f_s$ of the digital system and the IF frequency $f_I$. This fact, which is not at all obvious at first glance, means that
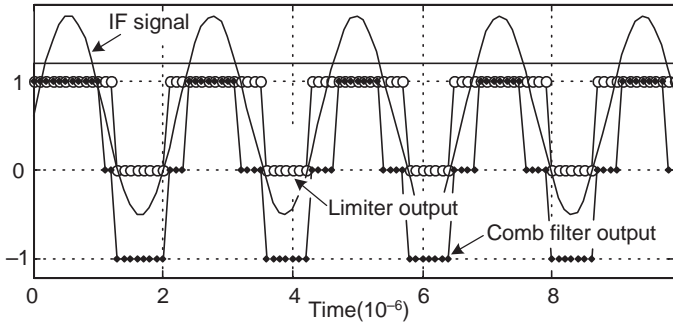
**Figure 5-5** Limiter output signal at 60% duty cycle and $N=11$ comb filter output.

our comb filter trick is inherently immune to varying peak-to-peak amplitudes, or changing DC bias levels, of the analog IF signal that occurs in many practical applications. To demonstrate this favorable comb filter behavior, let us assume the bias level (the average) of the analog IF signal in Figure 5-2(a) increased to a positive value such that the limiter's binary output sequence had a duty cycle of 60% as shown in Figure 5-5. There we see how our $N=11$ comb filter's output sequence is bipolar (DC bias removal has occurred), maintains a 50% duty cycle, and has a fundamental frequency equal to that of the IF signal.

A comb filter may also be used at the baseband to solve the same DC bias removal problem at demodulator output. Unlike that in the IF signal, the DC offset at demodulator output is a consequence of uncorrected frequency drift between the transmitter and the receiver [7]. Thus, we can effectively combat variation in the bit duration of the demodulated data caused by the DC offset that occurred in the baseband circuitry using the same methodology.

## 5.4   CONCLUSIONS

Here we showed how an efficient comb filter can be used to both remove the DC offset of our signals and reduce the complexity of follow-on filtering. The advantages of the comb filter are its implementation simplicity (a single exclusive-or circuit) and its ability to combat the problem of DC offset for both IF and baseband signals. Through examples, we show that the duty cycle for signals at the IF can be easily corrected to a 50% duty cycle. This corrected signal is amenable to additional processing at its IF frequency or may be down-converted to a baseband frequency for further processing.

## 5.5   REFERENCES

[1]  M. Simon and J. Springett, "The Performance of a Noncoherent FSK Receiver Preceded by a Bandpass Limiter," *IEEE Trans. Commun.*, vol. COM-20, December 1972, pp. 1128–1136.
[2]  B. Razavi, *RF Microelectronics*, 1st ed. Prentice Hall, Englewood Cliffs, NJ, 1998.

[3] P. HUANG, et al., "A 2-V CMOS 455-kHz FM/FSK Demodulator Using Feedforward Offset Cancellation Limiting Amplifier," *IEEE J. Solid-State Circuits*, vol. 36, January 2001, pp. 135–138.

[4] J. PROAKIS, *Digital Communications*, 3rd ed. McGraw-Hill, New York, 1995.

[5] E. HOGENAUER, "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Trans. Acoust. Speech, Signal Processing*, vol. ASSP-29, no. 2, April 1981, pp. 155–162.

[6] K. PARHI, *VLSI Digital Signal Processing Systems*, 1st ed. John Wiley & Sons, New York, 1999.

[7] F. WESTMAN, et al., "A Robust CMOS Bluetooth Radio/Modem System-on-Chip," *IEEE Circuits & Device Mag.*, November 2002, pp. 7–16.

## EDITOR COMMENTS

To further describe the comb filter used in this chapter, the details of that filter are shown in Figure 5-6(a), where the BSR means a binary (single-bit) shift register. This filter, whose input is bipolar binary bits (+1, −1), is a specific form of the more general comb filter shown in Figure 5-6(b) where the $w(n)$ input is a multi-bit binary word. In the literature of DSP, such a comb filter is typically depicted as that shown in Figure 5-6(c) where, in this case, $N=11$.

The Figure 5-6(c) comb filter's $z$-domain transfer function is

$$H(z) = (1 - z^{-N}) \qquad (5\text{–}4)$$

yielding $z$-plane zeros located at multiples of the $N$th root of one as shown in Figure 5-7(a) when, for example, $N=8$. That filter's frequency magnitude response is plotted in Figure 5-7(b). Comb filter behavior when $N=9$ is shown in Figures 5-7(c) and (d).

The frequency magnitude response of this comb filter is
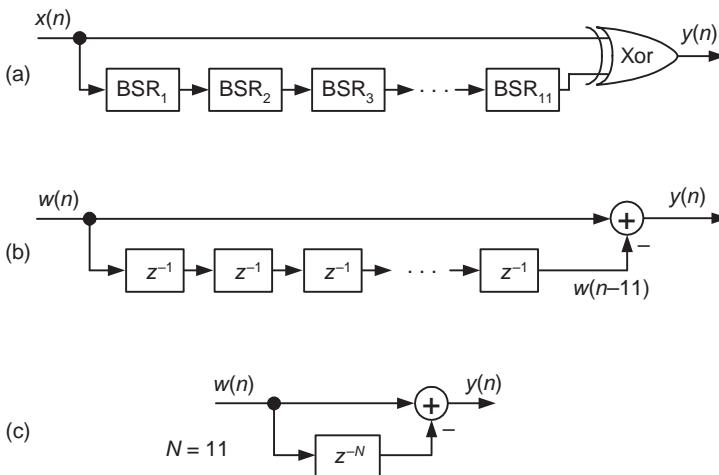


**Figure 5-6**    Comb filter: (a) single-bit $x(n)$; (b) multi-bit $x(n)$; (c) typical comb filter depiction.
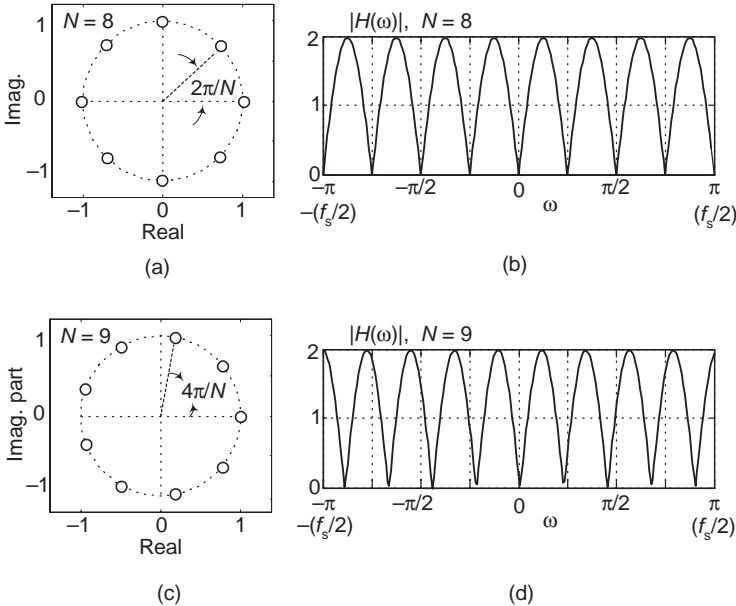
**Figure 5-7**    Comb filter characteristics: (a) $z$-plane zeros when $N=8$; (b) frequency response when $N=8$; (c) $z$-plane zeros when $N=9$; (d) frequency response when $N=9$.

$$|H(\omega)| = 2|\sin(\omega N/2)| \qquad\qquad (5\text{–}5)$$

having maximum magnitude of 2 as plotted in Figure 5-7. When $N$ is odd (e.g., $N=9$), a comb filter's $z$-plane zeros are those shown in Figure 5-7(c), with the filter's associated frequency magnitude response depicted in Figure 5-7(d).

As an aside, we mention here that an *alternative* comb filter can be built using the network in Figure 5-8(a) where addition is performed as opposed to the subtraction in Figure 5-6(c). This alternative comb filter (not applicable to DC bias removal) gives us a bit of design flexibility in using comb filters because it passes low-frequency signals due to its frequency magnitude peak at zero Hz. This filter's $z$-domain transfer function is

$$H_{\text{alt}}(z) = (1 + z^{-N}) \qquad\qquad (5\text{–}6)$$

with its $z$-plane zeros located as shown in Figure 5-8(b) when, for example, $N=8$. That filter's frequency magnitude response is plotted in Figure 5-8(c). When $N$ is odd (e.g., $N=9$), the alternate comb filter's $z$-plane zeros are those shown in Figure 5-8(d), with the $N=9$ filter's associated frequency magnitude response depicted in Figure 5-8(e).

(a)



(b) (c)



(d) (e)
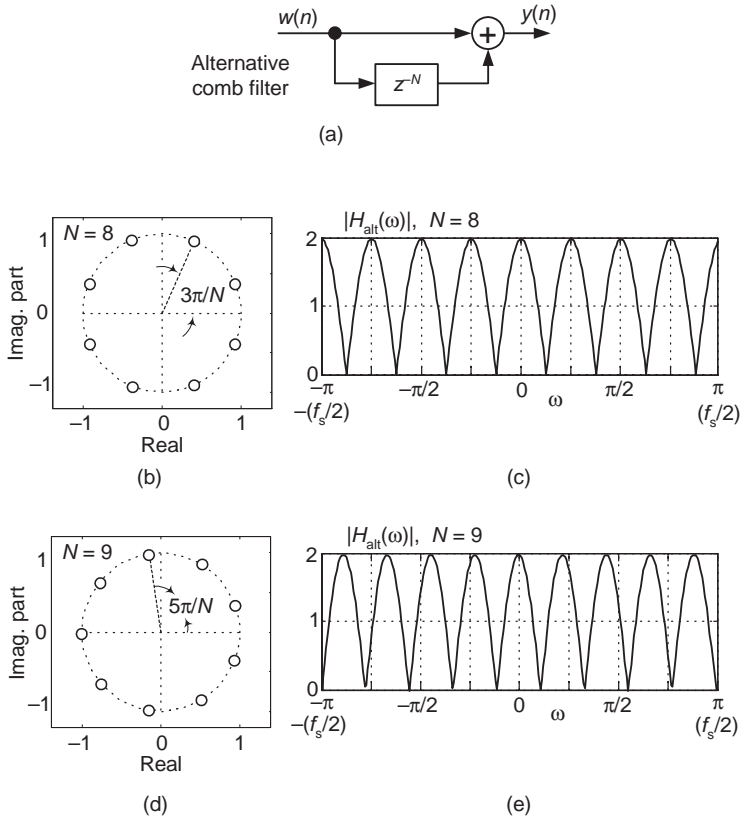
**Figure 5-8** Alternative comb filter: (a) structure; (b) $z$-plane zeros when $N=8$; (c) frequency response when $N=8$; (d) $z$-plane zeros when $N=9$; (e) frequency response when $N=9$.

Following the notation in (5–2) and (5–3), the alternative comb filter's local minimum ($L_{min}$) and local maximum ($L_{max}$) in the positive-frequency range of $|H_{alt}(\omega)|$ are

$$L_{min}: \omega = \frac{\pi}{N} + \frac{2\pi}{N} \cdot i, \quad i = 0, 1, \ldots, \left\lfloor \frac{N-1}{2} \right\rfloor \tag{5–7}$$

$$L_{max}: \omega = \frac{2\pi}{N} \cdot i, \quad i = 0, 1, \ldots, \left\lfloor \frac{N}{2} \right\rfloor. \tag{5–8}$$

# Chapter 6

# Reducing CIC Filter Complexity

**Ricardo A. Losada**
*The MathWorks, Inc.*

**Richard Lyons**
*Besser Associates*

Cascaded integrator-comb (CIC) filters are used in high-speed interpolation and decimation applications. This chapter provides tricks to reduce the complexity and enhance the usefulness of CIC filters. The first trick shows a way to reduce the number of adders and delay elements in a multistage CIC interpolation filter. The result is a multiplierless scheme that performs high-order linear interpolation using CIC filters. The second trick shows a way to eliminate the integrators from CIC decimation filters. The benefit is the elimination of unpleasant data word growth problems.

## 6.1 REDUCING INTERPOLATION FILTER COMPLEXITY

CIC filters are widely used for efficient multiplierless interpolation. Typically, such filters are not used standalone; instead, they are usually used as part of a multisection interpolation scheme, generally as the last section where the data has already been interpolated to a relatively high data rate. The fact that the CIC filters need to operate at such high rates makes their multiplierless-nature attractive for hardware implementation.

Typical CIC interpolator filters usually consist of cascaded single stages reordered in such a way that all the comb filters are grouped together as are all the

integrator filters. By looking closely at a single-stage CIC interpolator, we will show a simple trick to reduce the complexity of a multistage implementation. Because multistage CIC interpolators have a single-stage CIC interpolator at its core, this trick will simplify the complexity of any CIC interpolator.

Consider the single-stage CIC interpolator in Figure 6-1(a). The " $\uparrow R$" symbol means insert $R-1$ zero-valued samples in between each sample of the output of the first adder comb. For illustration purposes, assume $R=3$. Now imagine an arbitrary $x(k)$ input sequence and assume the delays' initial conditions are equal to zero. When the first $x(0)$ sample is presented at the CIC input, $u(n)=\{x(0),0,0\}$. The first $y(n)$ output will be $x(0)$; then this output is fed back and added to zero. So the second $y(n)$ output will be $x(0)$ as well, and the same for the third $y(n)$. Overall, the first $x(0)$ filter input sample produces the output sequence $y(n)=\{x(0),x(0),x(0)\}$. The next sample input to the comb is $x(1)$ making $u(n)=\{x(1)-x(0),0,0\}$. The integrator delay has the value $x(0)$ stored. We add it to $x(1)-x(0)$ to get the next output $y(n)=x(1)$. The value $x(1)$ is stored in the integrator delay and is then added to zero to produce the next output $y(n)=x(1)$. Continuing in this manner, the second input sample to the CIC filter, $x(1)$, produces the output sequence $y(n)=\{x(1),x(1),x(1)\}$. This behavior repeats so that for a given CIC input sequence $x(k)$, the output $y(n)$ is a sequence where each input sample is repeated $R$ times. This is shown in Figures 6-1(b) and 6-1(c) for $R=3$.

Naturally, when implementing a single-stage CIC filter in real hardware, it is not necessary to use the adders and delays (or the "zero-stuffer") shown in Figure 6-1(a). It is simply a matter of repeating each input sample $R-1$ times, imposing no hardware cost.

Let us next consider a multistage CIC filter such as the one shown in Figure 6-2(a) having 3 stages. At its core, there is a single-stage CIC interpolator. Our first trick, then, is to replace the innermost single-stage interpolator with a black-box, which we call a *hold interpolator*, whose job is to repeat each input sample $R-1$
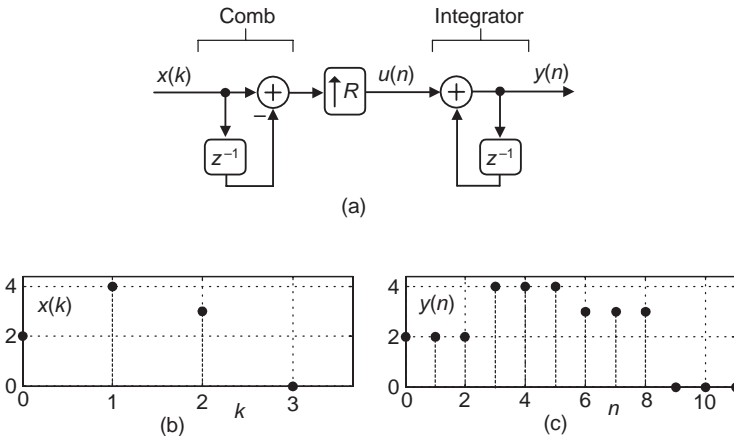


**Figure 6-1**    CIC interpolation filter: (a) structure; (b) input sequence; (c) output sequence for $R=3$.
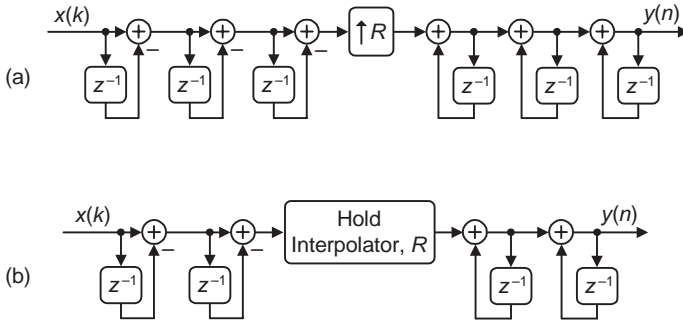
**Figure 6-2**    Three-stage CIC interpolation filter: (a) standard structure; (b) reduced-complexity structure.

times as explained above. Such a reduced-complexity CIC scheme is shown in Figure 6-2(b).

Note that in the comb sections, the number of bits required for each adder tends to increase as we move from left to right. Therefore, the adder and delay that can be removed from the comb section will typically be the ones that require the greatest number of bits in the entire comb section for a standard implementation of a CIC interpolator. So this trick enables us to remove the adder and delay in that section that will save us the greatest number of bits. However, this is not the case in the integrator section, where we remove the adder and delay that would require the least number of bits of the entire section (but still as many or more than any adder or delay from the comb section).

## 6.2  AN EFFICIENT LINEAR INTERPOLATOR

Linear interpolators, as their name implies, interpolate samples between two adjacent samples (of the original signal to be interpolated) by placing them in an equidistant manner on the straight line that joins those two adjacent samples [1]. The behavior is illustrated in Figure 6-3 for the case when $R=3$. Using an example, we now present a very efficient scheme to compute those interpolated samples in a way that requires no multiplies.

As with CIC filters, the performance of linear interpolators is not that great when they are used on their own. However, linear interpolators are usually not used that way. The reason is that if the interpolation factor $R$ is high, the error introduced by assuming a straight line between two adjacent samples can be large. On the other hand, if interpolation is done in multiple sections, linear interpolation at the end when the signal samples are already very close together will introduce only a small error. Linear interpolation requires a relatively small amount of computation, which is why linear interpolators are used at very high sample rates.

To compute the interpolated $y(n)$ samples using a digital filter, we can use the simple structure shown in Figure 6-4. Let's not concern ourselves with startup
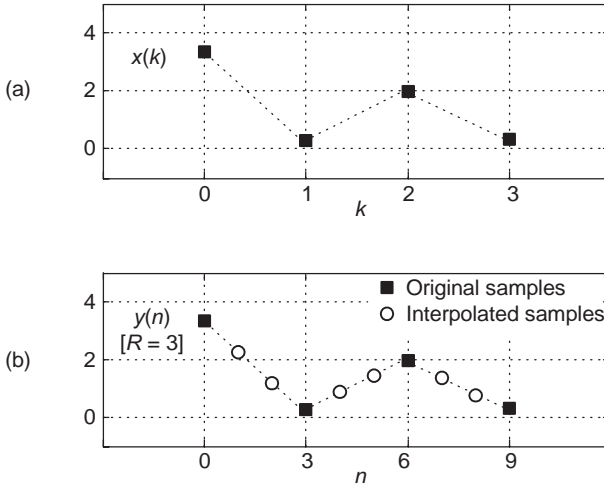
**Figure 6-3**    Linear interpolation, $R=3$: (a) input $x(k)$ samples; (b) interpolated $y(n)$ samples.
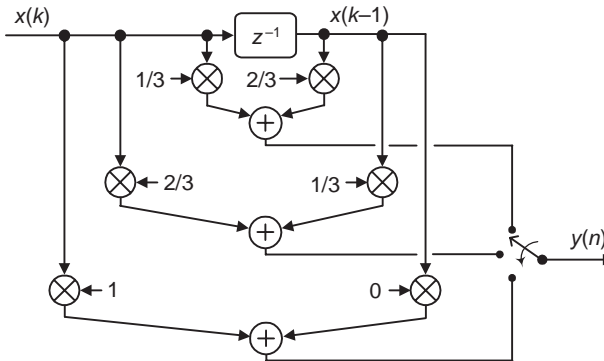


**Figure 6-4**    Interpolation filter, polyphase implementation.

transients and assume the filter is in steady state and we have already computed $y(n)$ for $n=1$, 2, 3 in Figure 6-3(b). We now show how $y(n)$ is computed for $n=4$, 5, 6. The procedure repeats itself from there on. The input to the filter is the signal $x(2)$, while the delay register holds the value of the previous input sample $x(1)$. To compute $y(4)$, we form a weighted average of the $x(1)$ and $x(2)$ samples. Because $y(4)$ is twice as close to $x(1)$ as it is to $x(2)$, we multiply $x(1)$ by 2/3 and $x(2)$ by 1/3. Next, the input signal and the content in the delay element remain the same since the input is operating at the slow rate. In order to compute $y(5)$, we change the weights to be 1/3 and 2/3. Finally, we "compute" $y(6)$ using the weights 0 and 1.

The procedure we have described is in fact a polyphase implementation of a linear interpolator with $R=3$, as shown in Figure 6-4. There are three polyphase branches since $R=3$. Because every input sample needs to be multiplied by each
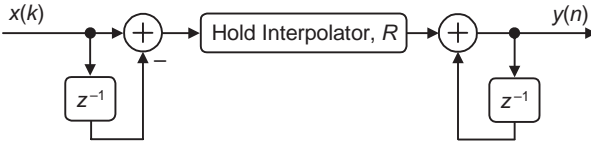
**Figure 6-5** Multiplierless linear interpolator.

one of the polyphase coefficients, the polyphase implementation of the linear inter-polator requires four multiplications for each input sample (we don't count the multiplication by 1 or by 0). In general, for an interpolation factor of $R$, the number of multiplications required per input sample is $2R-2$.

By grouping these polyphase coefficients, we can form the transfer function of the $R=3$ linear interpolator as:

$$H_{\text{linear}}(z) = \frac{1}{3} + \frac{2z^{-1}}{3} + z^{-2} + \frac{2z^{-3}}{3} + \frac{z^{-4}}{3}. \tag{6–1}$$

Now we notice that this transfer function is nothing but a scaled version of a two-stage CIC interpolator. Indeed, for a two-stage CIC we have

$$H_{\text{cic}}(z) = \left(\frac{1-z^{-3}}{1-z^{-1}}\right)^2 = (1+z^{-1}+z^{-2})^2 = 3H_{\text{linear}}(z). \tag{6–2}$$

Equation (6–2) shows that by using two-stage CIC interpolators, we can implement linear interpolation by $R=3$ without the need for the $2R-2$ multipliers. Next, we can use the hold interpolator trick, presented earlier, to simplify the linear interpola-tor even further.

Using a hold interpolator that inserts $R-1=2$ repeated values for each input sample we can perform efficient linear interpolation as in Figure 6-5. This imple-mentation requires only two adders and two delays no matter what the value of $R$. The order of this efficient linear interpolator can, of course, be increased by merely increasing the sample repetition factor $R$.

## 6.3  NONRECURSIVE CIC DECIMATION FILTERS

CIC filters are computationally efficient and simple to implement. However, there's trouble in paradise. One of the difficulties in using CIC filters is accommodating large data word growth, particularly when implementing integrators in multistage CIC filters. Here's a clever trick that eases the word-width growth problem using nonrecursive CIC decimation filter structures, obtained by means of *polynomial factoring*. These nonrecursive structures achieve computational simplicity through *polyphase decomposition* if the sample rate reduction factor $R$ is an integer power of two.

Recall that the transfer function of an $L$th-order decimation CIC filter can be expressed in either a recursive form or a nonrecursive form as given by:
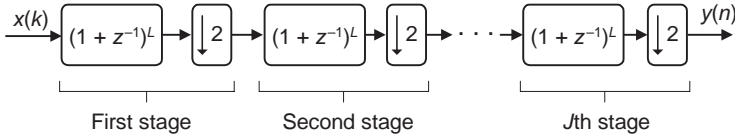
**Figure 6-6**   Multistage $L$th-order nonrecursive CIC structure.

$$H_{cic}(z) = \left(\frac{1-z^{-R}}{1-z^{-1}}\right)^L = \left[\sum_{n=0}^{R-1} z^{-n}\right]^L = (1+z^{-1}+z^{-2}+\ldots+z^{-R+1})^L. \qquad (6\text{--}3)$$

Now if the sample rate change factor $R$ is an integer power of two, then $R = 2^J$ where $J$ is a positive integer, and the $L$th-order nonrecursive polynomial form of $H_{cic}(z)$ in (6–3) can be factored as

$$H_{cic}(z) = (1+z^{-1})^L(1+z^{-2})^L(1+z^{-4})^L\ldots(1+z^{-2^{J-1}})^L. \qquad (6\text{--}4)$$

The benefit of the factoring given in (6–4) is that the CIC decimation filter can then be implemented with $J$ nonrecursive stages as shown for the multistage CIC filter in Figure 6-6. This implementation trick eliminates filter feedback loops with their unpleasant binary word-width growth. The data word widths increase by $L$ bits per stage, while the sampling rate is reduced by a factor of two for each stage.

This nonrecursive structure has been shown to consume less power than the Figure 6-2(a) recursive implementation for filter orders greater than three and decimation factors larger than eight. Thus the power savings from sample rate reduction is greater than the power consumption increase due to data word-width growth. By the way, the cascade of nonrecursive subfilters in Figure 6-6 are still called CIC filters even though they have no integrators!

Lucky for us, further improvements are possible with each stage of this nonrecursive structure [2]–[4]. For example, assume $L = 5$ for the first stage in Figure 6-6. In that case the first stage's transfer function is

$$\begin{aligned}
H(z) &= (1+z^{-1})^5 = 1+5z^{-1}+10z^{-2}+10z^{-3}+5z^{-4}+z^{-5} \\
&= 1+10z^{-2}+5z^{-4}+(5+10z^{-2}+z^{-4})z^{-1} = H_1(z)+H_2(z)z^{-1}.
\end{aligned} \qquad (6\text{--}5)$$

The last step in (6–5), known as polyphase decomposition, enables a polyphase implementation having two parallel paths as shown in Figure 6-7(a). Because we implement decimation by 2 before the filtering, the new polyphase components are $H_{1'}(z) = 1+10z^{-1}+5z^{-2}$ and $H_{2'}(z) = 5+10z^{-1}+z^{-2}$ implemented at half the data rate into the stage. (Reducing data rates as early as possible is a key design goal in the implementation of CIC decimation filters.) The initial delay element and the dual decimation by two operations can implemented by routing the odd-index input samples to $H_{1'}(z)$, and the even-index samples to $H_{2'}(z)$ as shown in Figure 6-7(b). Of course the $H_{1'}(z)$ and $H_{2'}(z)$ polyphase components are implemented in a tapped-delay line fashion.

Fortunately, we can further simplify the $H_{1'}(z)$ and $H_{2'}(z)$ polyphase components. Let's consider the $H_{1'}(z)$ polyphase filter component (implemented in a tapped-delay
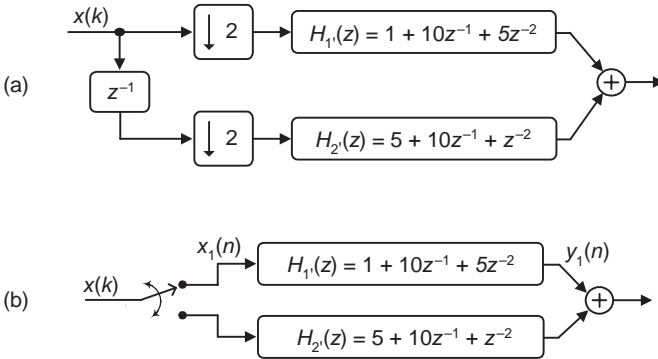
**Figure 6-7**   Polyphase structure of a single nonrecursive fifth-order CIC stage.
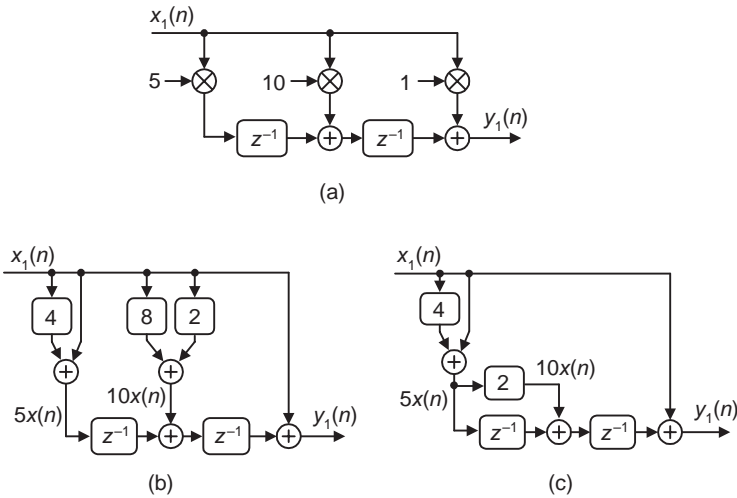


**Figure 6-8**   Filter component $H_1(z)$: (a) delay line structure; (b) transposed structure; (c) simplified multiplication; (d) substructure sharing.

line configuration) shown in Figure 6-7(b). The transposed version of this filter is presented in Figure 6-8(a) with its flipped coefficient sequence. The adder used in the standard tapped-delay implementation to implement $H_{1'}(z)$ in Figure 6-7(b) must perform two additions per output data sample, while in the transposed structure no adder need perform more than one addition per output sample. So the transposed structure can operate at a higher speed.

The next improvement, proposed by Gao *et al.* [4], uses simplified multiplication, as shown in Figure 6-8(b), by means of arithmetic left-shifts and adds. Thus a factor of 5 is implemented as $2^2 + 1$, eliminating all multiplications. Finally, because of the transposed structure, we can use the technique of *substructure sharing* in Figure 6-8(c) to reduce the hardware component count.

The nonrecursive CIC decimation filters described above have the restriction that the *R* decimation factor must be an integer power of two. That constraint is loosened due to a clever scheme assuming *R* can be factored into the product of prime numbers. Details of that process, called *prime factorization*, are available in [2] and [5].

## 6.4    CONCLUSIONS

Here we showed CIC filter tricks to: (1) eliminate one stage in a multistage CIC interpolation filter, (2) perform computationally efficient linear interpolation using a CIC interpolation filter, (3) use nonrecursive structures to eliminate the integrators (along with their unpleasant data word-width growth problems) from CIC decimation filters, (4) use polyphase decomposition and substructure sharing to eliminate multipliers in the nonrecursive CIC decimation filters.

## 6.5    REFERENCES

[1]  S. ORFANIDIS, *Introduction to Signal Processing*. Prentice Hall, Upper Saddle River, NJ, 1996.

[2]  R. LYONS, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 2004.

[3]  L. ASCARI, et al., "Low Power Implementation of a Sigma Delta Decimation Filter for Cardiac Applications," *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21–23, 2001, pp. 750–755.

[4]  Y. GAO, et al., "Low-Power Implementation of a Fifth-Order Comb Decimation Filter for Multi-Standard Transceiver Applications," *Int. Conf. on Signal Proc. Applications and Technology (ICSPAT)*, Orlando, FL, October 1999. [Online: http://www.pcc.lth.se/events/workshops/1999/posters/Gao. pdf.]

[5]  Y. JANG, and S. YANG, "Non-Recursive Cascaded Integrator-Comb Decimation Filters with Integer Multiple Factors," *44th IEEE Midwest Symposium on Circuits and Systems (MWSCAS)*, Dayton, OH, August 2001, pp. 130–133.

# Chapter 7

# Precise Filter Design

**Greg Berchin**

*Consultant in Signal Processing*

You have just been assigned to a new project at work, in which the objective is to replace an existing analog system with a functionally equivalent digital system. Your job is to design a digital filter that matches the magnitude and phase response of the existing system's analog filter over a broad frequency range.

You are running out of ideas. The bilinear transform and impulse invariance methods provide poor matches to the analog filter response, particularly at high frequencies. Fast convolution requires more computational resources than you have and creates more input/output latency than you can tolerate. What will you do?

This chapter describes an obscure but simple and powerful method for designing a digital filter that approximates an arbitrary magnitude and phase response. If applied to the problem above, it can create a filter roughly comparable in computational burden and latency to the bilinear transform method, with fidelity approaching that of fast convolution. In addition, the method presented here can be also applied to a wide variety of other system identification tasks.

## 7.1 PROBLEM BACKGROUND

Filter specifications are commonly expressed in terms of passband width and flatness, transition bandwidth, and stopband attenuation. There may also be some general specifications about phase response or time-domain performance, but the exact magnitude and phase responses are usually left to the designer's discretion.

An important exception occurs, however, when a digital filter is to be used to emulate an analog filter. This is traditionally a very difficult problem, because analog systems are described by Laplace transforms, using integration and differentiation,

whereas digital systems are described by Z-transforms, using delay. Since the conversion between them is nonlinear, the response of an analog system can be only approximated by a digital system, and vice-versa.

## 7.2   TEXTBOOK APPROXIMATIONS

A common method used to create digital filters from analog prototypes is the *bilinear transform*. This technique can be very effective when specifications are given as passband, transition band, and stopband parameters, as described earlier. And implementation can be very efficient, because the number of coefficients in the digital filter is comparable to the number in its analog prototype. But the bilinear transform suffers from two problems that make a close match to an analog frequency response impossible:

- It squeezes the entire frequency range from zero to infinity in the analog system into the range from DC to half the sampling frequency, inducing *frequency warping*, in the digital system.
- It can match a prototype frequency response at only three frequencies. At all other frequencies the response falls where it may, though its behavior is predictable.

Another design method, called *impulse invariance*, matches the impulse response of the digital filter to the sampled impulse response of the prototype analog filter. Since there is a one-to-one mapping between impulse response and frequency response in the continuous-time case, one might assume that matching the digital impulse response to the analog impulse response will cause the digital frequency response to match the analog frequency response. Unfortunately, aliasing causes large frequency response errors unless the analog filter rolls off steeply at high frequencies.

A popular method for realizing arbitrary-response filters, called *fast convolution*, is implemented in the frequency domain by multiplying the FFT of the signal by samples of the analog filter's frequency response, computing the inverse FFT, and so on. While it can be very effective, it is computationally intensive and suffers from high input-output latency.

## 7.3   AN APPROXIMATION THAT YOU WON'T FIND IN ANY TEXTBOOK

The filter approximation method we present here is called *frequency-domain least-squares* (FDLS). I developed FDLS while I was a graduate student [1] and described it in some conference papers [2], [3], but the technique was all but forgotten after I left school. The FDLS algorithm produces a transfer function that approximates an arbitrary frequency response. The input to the algorithm is a set of magnitude and phase values at a large number (typically thousands) of arbitrary frequencies between zero Hz and half the sampling rate. The algorithm's output is a set of transfer

function coefficients. The technique is quite flexible in that it can create transfer functions containing poles and zeros (IIR), only zeros (FIR), or only poles (autoregressive). Before we can see how the technique works, we need to review some linear algebra and matrix concepts. The FDLS algorithm uses nothing more esoteric than basic linear algebra.

## 7.4  LEAST SQUARES SOLUTION

Hopefully the reader remembers that in order to uniquely solve a system of equations we need as many equations as unknowns. For example, the single equation with one unknown, $5x = 7$, has the unique solution $x = 7/5$. But the single equation with two unknowns, $5x + 2y = 7$, has multiple solutions for $x$ that depend on the unspecified value of $y$: $x = (7 - 2y)/5$.

If another equation is added, such as

$$5x + 2y = 7$$
$$-6x + 4y = 9,$$

then there are unique solutions for both $x$ and $y$ that can be found algebraically or by matrix inversion (denoted in the following by a "$-1$" superscript):

$$\begin{bmatrix} 5 & 2 \\ -6 & 4 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \end{bmatrix}.$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ -6 & 4 \end{bmatrix}^{-1}\begin{bmatrix} 7 \\ 9 \end{bmatrix} = \begin{bmatrix} \dfrac{1}{8} & -\dfrac{1}{16} \\ \dfrac{3}{16} & \dfrac{5}{32} \end{bmatrix}\begin{bmatrix} 7 \\ 9 \end{bmatrix} = \begin{bmatrix} \left(\dfrac{7}{8} - \dfrac{9}{16}\right) \\ \left(\dfrac{21}{16} + \dfrac{45}{32}\right) \end{bmatrix}.$$

Now let us consider what happens if we add another equation to the pair that we already have (we will see later why we might want to do this), such as

$$5x + 2y = 7$$
$$-6x + 4y = 9$$
$$x + y = 5.$$

There are no values of $x$ and $y$ that satisfy all three equations simultaneously. Well, matrix algebra provides something called the *pseudoinverse* to deal with this situation. It determines the values of $x$ and $y$ that come *as close as possible*, in the least-squares sense, to satisfying all three equations.

Without going into the derivation of the pseudoinverse or the definition of least-squares, let us simply jump to the solution to this new problem:

$$\begin{bmatrix} 5 & 2 \\ -6 & 4 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \\ 5 \end{bmatrix} \quad \text{or}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \approx \left( \begin{bmatrix} 5 & 2 \\ -6 & 4 \\ 1 & 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} 5 & 2 \\ -6 & 4 \\ 1 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 5 & 2 \\ -6 & 4 \\ 1 & 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} 7 \\ 9 \\ 5 \end{bmatrix} \approx \begin{bmatrix} 0.3716 \\ 2.8491 \end{bmatrix}.$$

(The "T" superscript denotes the matrix transpose.) The pseudoinverse always computes the set of values that comes as close as possible to solving *all* of the equations, when there are more equations than unknowns.

As promised, everything that we have discussed is plain-vanilla linear algebra. The mathematical derivation of the matrix inverse and pseudoinverse, and the definition of least-squares, can be found in any basic linear algebra text. And our more mathematically inclined readers will point out that there are better ways than this to compute the pseudoinverse, but this method is adequate for our example.

Now that we remember how to solve simultaneous equations, we need to figure out how to get the equations in the first place. To do that, we first need to review a little DSP.

We will start with the transfer function, which is a mathematical description of the relationship between a system's input and its output. We will assume that the transfer function of our digital filter is in a standard textbook form:

$$\frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + \ldots + a_D z^{-D}}$$

where $U(z)$ is the z-transform of the input signal and $Y(z)$ is the z-transform of the output signal. Furthermore, we assume that the filter is causal, meaning that its response to an input does not begin until after the input is applied. (The filter cannot see into the future.) Under these circumstances the time-domain difference equation that implements our filter is:

$$y(k) = -a_1 y(k-1) - \ldots - a_D y(k-D) + b_0 u(k) + \ldots + b_N u(k-N)$$

where the *a* and *b* coefficients are exactly the same as in the transfer function above, *k* is the time index, $u(k)$ and $y(k)$ are the current values of the input and output, respectively, $u(k-N)$ was the input value *N* samples in the past, and $y(k-D)$ was the output value *D* samples in the past. We can write the equation above in matrix form as

$$y(k) =$$

$$[-y(k-1) \quad \ldots \quad -y(k-D) \, u(k) \quad \ldots \quad u(k-N)] \begin{bmatrix} a_1 \\ \vdots \\ a_D \\ b_0 \\ \vdots \\ b_N \end{bmatrix}.$$

We conclude our review of DSP with a consideration of exactly what a frequency response value means. If, for example, the frequency response of a system

at a frequency $\omega_1$ is given in magnitude/phase form to be $A_1\angle\phi_1$, it means that the output amplitude will be $A_1$ times the input amplitude, and the output phase will be shifted an angle $\phi_1$ relative to the input phase when a steady-state sine wave of frequency $\omega_1$ is applied to the system.

Let's look at an example. If the input to the system described above, at time $k$, is

$$u_1(k) = \cos(k\omega_1 t_s)$$

where $t_s$ is the sampling period (equal to one over the sampling frequency), then the output will be

$$y_1(k) = A_1 \cos(k\omega_1 t_s + \phi_1).$$

The input and output values at *any* sample time can be determined in a similar manner. For example, the input sample value $N$ samples in the past was

$$u_1(k - N) = \cos((k - N)\omega_1 t_s)$$

and the output sample value $D$ samples in the past was

$$y_1(k - D) = A_1 \cos((k - D)\omega_1 t_s + \phi_1).$$

That's all there is to it. For our purposes, since $k$ represents the current sample time its value can be conveniently set to zero.

That is the end of our review of frequency response, transfer function, and pseudoinverse. Now we will put it all together into a filter design technique. We have just demonstrated that the relationship between input $u$ and output $y$ at any sample time can be inferred from the frequency response value $A\angle\phi$ at frequency $\omega$. We know from our discussion of transfer function that the output is a combination of present and past input and output values, each scaled by a set of $b$ or $a$ coefficients, respectively, the values of which are not yet known. Combining the two, our frequency response value $A_1\angle\phi_1$ at $\omega_1$, in the example above, provides us with one equation in $D+N+1$ unknowns:

$$y_1(0) =$$

$$[-y_1(-1) \quad \dots \quad -y_1(-D) \, u_1(0) \quad \dots \quad u_1(-N)] \begin{bmatrix} a_1 \\ \vdots \\ a_D \\ b_0 \\ \vdots \\ b_N \end{bmatrix}.$$

(Note that $k$, the current-sample index, has been set to zero.) If we do exactly the same thing again, except this time using frequency response $A_2\angle\phi_2$ at a *different* frequency $\omega_2$, we obtain a second equation in $D+N+1$ unknowns:

$$
\begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} =
$$

$$
\begin{bmatrix} -y_1(-1) & \cdots & -y_1(-D) & u_1(0) & \cdots & u_1(-N) \\ -y_2(-1) & \cdots & -y_2(-D) & u_2(0) & \cdots & u_2(-N) \end{bmatrix}
\begin{bmatrix} a_1 \\ \vdots \\ a_D \\ b_0 \\ \vdots \\ b_N \end{bmatrix}.
$$

And if we keep doing this at many more *different* frequencies $M$ than we have unknowns $D+N+1$, we know from our review of linear algebra that the pseudoinverse will compute values for the set of coefficients $a_1 \ldots a_D$ and $b_0 \ldots b_N$ that come as close as possible to solving *all* of the equations, *which is exactly what we need to design our filter.* So now we can write:

$$
\begin{bmatrix} y_1(0) \\ y_2(0) \\ \vdots \\ y_M(0) \end{bmatrix} =
$$

$$
\begin{bmatrix} -y_1(-1) & \cdots & -y_1(-D) & u_1(0) & \cdots & u_1(-N) \\ -y_2(-1) & \cdots & -y_2(-D) & u_2(0) & \cdots & u_2(-N) \\ \vdots & & \vdots & \vdots & & \vdots \\ -y_M(-1) & \cdots & -y_M(-D) & u_M(0) & \cdots & u_M(-N) \end{bmatrix}
\begin{bmatrix} a_1 \\ \vdots \\ a_D \\ b_0 \\ \vdots \\ b_N \end{bmatrix}.
$$

We can employ shortcut matrix notation by defining the following vectors and matrix:

$$
Y = \begin{bmatrix} y_1(0) \\ y_2(0) \\ \vdots \\ y_M(0) \end{bmatrix}, \Theta = \begin{bmatrix} a_1 \\ \vdots \\ a_D \\ b_0 \\ \vdots \\ b_N \end{bmatrix}, \quad \text{and}
$$

$$
X = \begin{bmatrix} -y_1(-1) & \cdots & -y_1(-D) & u_1(0) & \cdots & u_1(-N) \\ -y_2(-1) & \cdots & -y_2(-D) & u_2(0) & \cdots & u_2(-N) \\ \vdots & & \vdots & \vdots & & \vdots \\ -y_M(-1) & \cdots & -y_M(-D) & u_M(0) & \cdots & u_M(-N) \end{bmatrix};
$$
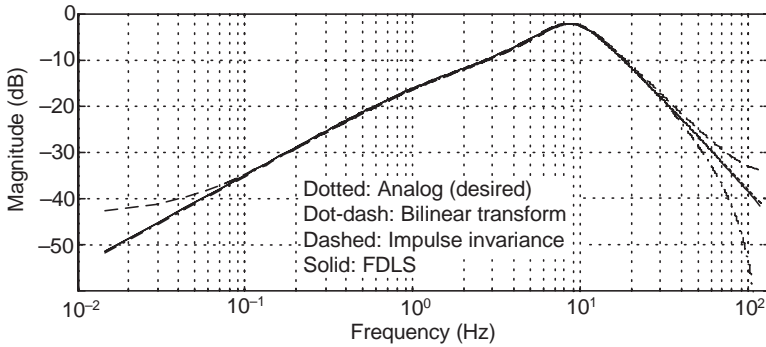
**Figure 7-1**   Magnitude responses.

then $Y=X\Theta$, and the pseudoinverse is $(X^TX)^{-1}X^TY \approx \Theta$, where vector $\Theta$ contains the desired filter coefficients.

That's it. That is the entire algorithm. We can now describe our filter design trick as follows:

1. Choose the numerator order $N$ and the denominator order $D$, where $N$ and $D$ do not have to be equal and either one (but not both) may be zero.

2. Define the $M$ separate input $u_m$ cosine sequences, each of length $(N+1)$.

3. Compute the $M$ separate output $y_m$ cosine sequences, each of length $D$ (based on $A_m \angle \phi_m$).

4. Fill the $X$ matrix with the input $u_m$ and output $y_m$ cosine sequences.

5. Fill the $Y$ vector with the $M$ output cosine values, $y_m(0)=A_m \cos(\phi_m)$.

6. Compute the pseudoinverse, and the $\Theta$ vector contains the filter coefficients.

Figures 7-1 and 7-2 show the magnitude and phase, respectively, of a real-world example analog system (dotted), and of the associated bilinear transform (dot-dash), impulse invariance (dashed), and FDLS (solid) approximations. The sampling rate is 240 Hz, and $D=N=12$. The dotted analog system curves are almost completely obscured by the solid FDLS curves. In this example, the FDLS errors are often 3 to 4 orders of magnitude smaller than those of the other methods. (In Figure 7-2, the bilinear transform curve is obscured by the FDLS curve at low frequencies and by the impulse invariance curve at high frequencies.)

## 7.5  IMPLEMENTATION NOTES

- I have found no rule of thumb for defining $N$ and $D$. They are best determined experimentally.
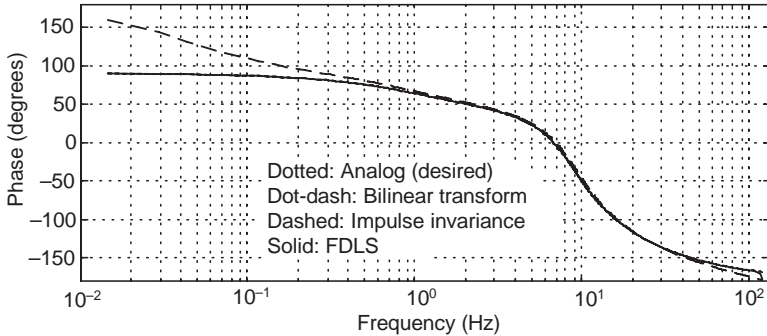
**Figure 7-2**    Phase responses.

- The selection of a cosine input, rather than sine, was *not* arbitrary. A sine formulation suffers from *zero-crossing* problems at frequencies near half the sampling frequency [1].

- For similar reasons, the algorithm can have some difficulties modeling a system whose phase response approaches odd multiples of 90° near half the sampling frequency. Adding a few samples of artificial delay to the frequency response data, in the form of a linear phase shift, solves this problem. "$\delta$" samples of delay result in a phase shift of "$-\delta\omega_m t_s$," so each equation for $y_m(k)$,

$$y_m(k) = A_m \cos(k\omega_m t_s + \phi_m),$$

  becomes:

$$y_m(k) = A_m \cos(k\omega_m t_s + \phi_m - \delta\omega_m t_s).$$

- The algorithm can be used to design 1-, 2-, or even 3-dimensional beam-formers [2]. (*Hint:* Making the delay value $p$, in equations of the general form $u(k-p) = \cos((k-p)\omega t_s)$, an integer in the range $(0, 1, \ldots, N)$ is overly restrictive. Variable $p$ does not have to be an integer and there are some very interesting designs that can be achieved if the integer restriction upon $p$ is removed.)

- A complex-number form of the algorithm exists, in which the inputs and outputs are complex sinusoids $e^{jk\omega t_s}$, the filter coefficients can be complex, and the frequency response can be asymmetrical about 0 Hz (or the beam pattern can be asymmetrical about array-normal).

- A *total-least-squares* formulation exists that concentrates all estimator errors into a single diagonal submatrix for convenient analysis [3].

In terms of the computational complexity of an FDLS-designed filter, the number of feedback and feedforward coefficients are determined by the variables $D$ and $N$, respectively. As such, an FDLS-designed filter requires $(N+D+1)$ multiplies and $(N+D)$ additions per filter output sample.

## 7.6   CONCLUSIONS

FDLS is a powerful method for designing digital filters. As is the case with all approximation techniques, there are circumstances in which the FDLS method works well, and others in which it does not. It does not replace other filter design methods; it provides one more method from which to choose. It is up to the designer to determine whether to use it in any given situation.

## 7.7   REFERENCES

[1]  G. BERCHIN, "A New Algorithm for System Identification from Frequency Response Information," Master's Thesis, University of California—Davis, 1988.

[2]  G. BERCHIN and M. SODERSTRAND, "A Transform-Domain Least-Squares Beamforming Technique," *Proceedings of the IEEE Oceans '90 Conference*, Arlington VA, September 1990.

[3]  G. BERCHIN and M. SODERSTRAND, "A Total Least Squares Approach to Frequency Domain System Identification," *Proceedings of the 32nd Midwest Symposium on Circuits and Systems*, Urbana, IL, August 1989.

## EDITOR COMMENTS

Here we present additional examples to illustrate the use of the FDLS algorithm.

### Algebraic Example

Recall the FDLS matrix expression

$$
\begin{bmatrix} y_1(0) \\ y_2(0) \\ \vdots \\ y_M(0) \end{bmatrix} = \begin{bmatrix} -y_1(-1) & \dots & -y_1(-D) & u_1(0) & \dots & u_1(-N) \\ -y_2(-1) & \dots & -y_2(-D) & u_2(0) & \dots & u_2(-N) \\ \vdots & & \vdots & \vdots & & \vdots \\ -y_M(-1) & \dots & -y_M(-D) & u_M(0) & \dots & u_M(-N) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_D \\ b_0 \\ \vdots \\ b_N \end{bmatrix}
$$

which we wrote as $Y = X\Theta$.

Each individual element in the $Y$ column vector is of the form $A_m\cos(\phi_m)$, and each element in the $X$ matrix is of the form $A_1\cos(k\omega_1 t_s + \phi_1)$ or $\cos(k\omega_1 t_s)$. Because all of these elements are of the form of a product (Amplitude)[cos(angle)], each element in $Y$ and $X$ is equal to a constant.

Now if, say, $D = 10$ and $N = 9$, then

$$
y_1(0) = A_1 \cos(\phi_1)
$$

is the first element of the $Y$ column vector and

$$
[-y_1(-1) \dots - y_1(-10)\, u_1(0) \dots u_1(-9)]
$$

is the top row of the $X$ matrix expression where

$$-y_1(-1) = -A_1\cos[(-1)\omega_1 t_s + \phi_1] = -A_1\cos(-\omega_1 t_s + \phi_1)$$
$$-y_2(-2) = -A_1\cos[(-2)\omega_1 t_s + \phi_1] = -A_1\cos(-2\omega_1 t_s + \phi_1)$$

$$\cdots$$

$$-y_1(-10) = -A_1\cos[(-10)\omega_1 t_s + \phi_1] = -A_1\cos(-10\omega_1 t_s + \phi_1)$$

and

$$u_1(0) = \cos[(0)\omega_1 t_s] = 1$$
$$u_1(-1) = \cos[(-1)\omega_1 t_s] = \cos(-\omega_1 t_s)$$
$$u_1(-2) = \cos[(-2)\omega_1 t_s] = \cos(-2\omega_1 t_s)$$

$$\cdots$$

$$u_1(-9) = \cos[(-9)\omega_1 t_s] = \cos(-9\omega_1 t_s).$$

So the top row of the $X$ matrix looks like:

$$[-A_1\cos(-\omega_1 t_s + \phi_1) \quad -A_1\cos(-2\omega_1 t_s + \phi_1)\ldots -A_1\cos(-10\omega_1 t_s + \phi_1) \quad 1$$
$$\cos(-\omega_1 t_s) \quad \cos(-2\omega_1 t_s)\ldots\cos(-9\omega_1 t_s)].$$

The second row of the $X$ matrix looks like:

$$[-A_2\cos(-\omega_2 t_s + \phi_2) \quad -A_2\cos(-2\omega_2 t_s + \phi_2)\ldots -A_2\cos(-10\omega_2 t_s + \phi_2) \quad 1$$
$$\cos(-\omega_2 t_s) \quad \cos(-2\omega_2 t_s)\ldots\cos(-9\omega_2 t_s)],$$

and so on.

## Numerical Example

Here is an example of the above expressions using actual numbers. Suppose we need to approximate the transfer function coefficients for the system whose frequency magnitude and phase response are as shown in Figure 7-3. Assume that our discrete-system sample rate is 1000 Hz, thus $t_s = 10^{-3}$ seconds, and $N = D = 2$. (The $N = D = 2$ values mean that our final filter will be a second-order filter.) Also assume $M = 8$ and we have the eight $A_1$-to-$A_8$ magnitude sample values and the eight $\phi_1$-to-$\phi_8$ phase samples, shown as dots in Figure 7-3, available to us as input values to the FDLS algorithm.

In matrix form, the target analog system parameters are

$$f_m = \begin{bmatrix} 0.0 \\ 19.6850 \\ 35.4331 \\ 51.1811 \\ 59.0551 \\ 66.9291 \\ 106.299 \\ 389.764 \end{bmatrix} \quad \omega_m t_s = \begin{bmatrix} 0.0 \\ 0.1237 \\ 0.2226 \\ 0.3216 \\ 0.3711 \\ 0.4205 \\ 0.6679 \\ 2.449 \end{bmatrix} \quad A_m = \begin{bmatrix} 0.2172 \\ 0.2065 \\ 0.1696 \\ 0.0164 \\ 1.3959 \\ 0.6734 \\ 0.3490 \\ 0.3095 \end{bmatrix} \quad \phi_m = \begin{bmatrix} 0.0 \\ -0.0156 \\ -0.0383 \\ 3.0125 \\ 2.3087 \\ 0.955 \\ 0.0343 \\ 0.0031 \end{bmatrix}$$

$\omega_m = 2\pi f_m:$
$\omega_1 = 2\pi(0.0)$
$\omega_2 = 2\pi(19.69)$
$\omega_3 = 2\pi(35.43)$
$\omega_4 = 2\pi(51.18)$
$\omega_5 = 2\pi(59.05)$
$\omega_6 = 2\pi(66.93)$
$\omega_7 = 2\pi(106.30)$
$\omega_8 = 2\pi(389.76)$
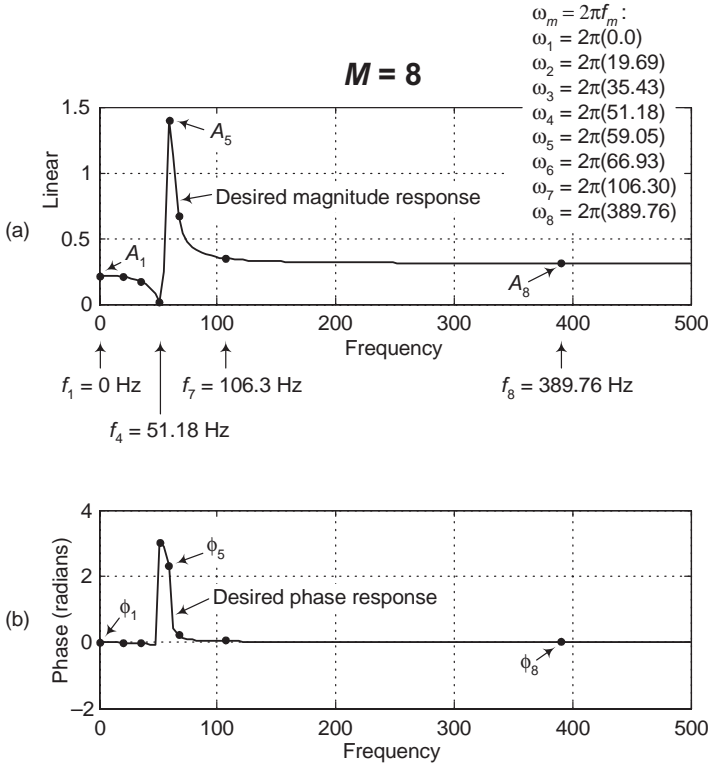
**Figure 7-3**    Desired magnitude and phase responses.

where the $f_m$ vector is in Hz, the $\omega_m t_s$ vector is in radians, and $1 \le m \le 8$. The first two elements of the $Y$ vector are:

$$y_1(0) = A_1 \cos(\phi_1) = 0.2172 \cos(0) = 0.2172,$$
$$y_2(0) = A_2 \cos(\phi_2) = 0.2065 \cos(-0.0156) = 0.2065.$$

The complete $Y$ vector is:

$$Y = \begin{bmatrix} A_1 \cos(\phi_1) \\ A_2 \cos(\phi_2) \\ A_3 \cos(\phi_3) \\ A_4 \cos(\phi_4) \\ A_5 \cos(\phi_5) \\ A_6 \cos(\phi_6) \\ A_7 \cos(\phi_7) \\ A_8 \cos(\phi_8) \end{bmatrix} = \begin{bmatrix} 0.2172 \\ 0.2065 \\ 0.1695 \\ -0.0162 \\ -0.9390 \\ 0.6605 \\ 0.3488 \\ 0.3095 \end{bmatrix}.$$

The two elements of the "$y_1$" part of the first row of the $X$ vector are:

$$-y_1(-1) = -A_1\cos(-\omega_1 t_s + \phi_1) = -0.2172\cos(-0+0) = -0.2172,$$
$$-y_1(-2) = -A_1\cos(-2\omega_1 t_s + \phi_1) = -0.2172\cos(-0+0) = -0.2172.$$

The two elements of the "$y_8$" part of the eighth row of the $X$ vector are:

$$-y_8(-1) = -A_8\cos(-\omega_8 t_s + \phi_8) = -0.3095\cos(-2.449 + 0.0031) = 0.2376,$$
$$-y_8(-2) = -A_8\cos(-2\omega_8 t_s + \phi_8) = -0.3095\cos(-4.898 + 0.0031) = -0.562.$$

The three elements of the "$u_1$" part of the first row of the $X$ matrix are:

$$u_1(0) = \cos(0) = 1$$
$$u_1(-1) = \cos(-\omega_1 t_s) = \cos(-0) = 1$$
$$u_1(-2) = \cos(-2\omega_1 t_s) = \cos(-0) = 1.$$

The three elements of the "$u_8$" part of the eighth row of the $X$ matrix are:

$$u_8(0) = \cos(0) = 1$$
$$u_8(-1) = \cos(-\omega_8 t_s) = \cos(-2.449) = -0.7696$$
$$u_8(-2) = \cos(-2\omega_8 t_s) = \cos(-4.898) = 0.1845.$$

The complete $X$ matrix is:

$$
X =
\begin{bmatrix}
-A_1\cos(-1\alpha_1 + \phi_1) & -A_1\cos(-2\alpha_1 + \phi_1) & \cos(0) & \cos(-1\alpha_1) & \cos(-2\alpha_1) \\
-A_2\cos(-1\alpha_2 + \phi_2) & -A_2\cos(-2\alpha_2 + \phi_2) & \cos(0) & \cos(-1\alpha_2) & \cos(-2\alpha_2) \\
-A_3\cos(-1\alpha_3 + \phi_3) & -A_3\cos(-2\alpha_3 + \phi_3) & \cos(0) & \cos(-1\alpha_3) & \cos(-2\alpha_3) \\
-A_4\cos(-1\alpha_4 + \phi_4) & -A_4\cos(-2\alpha_4 + \phi_4) & \cos(0) & \cos(-1\alpha_4) & \cos(-2\alpha_4) \\
-A_5\cos(-1\alpha_5 + \phi_5) & -A_5\cos(-2\alpha_5 + \phi_5) & \cos(0) & \cos(-1\alpha_5) & \cos(-2\alpha_5) \\
-A_6\cos(-1\alpha_6 + \phi_6) & -A_6\cos(-2\alpha_6 + \phi_6) & \cos(0) & \cos(-1\alpha_6) & \cos(-3\alpha_6) \\
-A_7\cos(-1\alpha_7 + \phi_7) & -A_7\cos(-2\alpha_7 + \phi_7) & \cos(0) & \cos(-1\alpha_7) & \cos(-3\alpha_7) \\
-A_8\cos(-1\alpha_8 + \phi_8) & -A_8\cos(-2\alpha_8 + \phi_8) & \cos(0) & \cos(-1\alpha_8) & \cos(-3\alpha_8)
\end{bmatrix}
$$

$$
=
\begin{bmatrix}
-0.2172 & -0.2172 & 1.0 & 1.0 & 1.0 \\
-0.2045 & -0.994 & 1.0 & 0.9924 & 0.9696 \\
-0.1639 & -0.1502 & 1.0 & 0.9753 & 0.9025 \\
0.0147 & 0.0117 & 1.0 & 0.9487 & 0.8002 \\
0.5007 & -0.0059 & 1.0 & 0.939 & 0.7370 \\
-0.6564 & -0.5378 & 1.0 & 0.9129 & 0.6667 \\
-0.2812 & -0.0928 & 1.0 & 0.7851 & 0.2328 \\
0.2376 & -0.0562 & 1.0 & -0.7696 & 0.1845
\end{bmatrix}
$$

where $\alpha_1 = \omega_1 t_s$, $\alpha_2 = \omega_2 t_s$, ... $\alpha_8 = \omega_8 t_s$. Given the above $Y$ vector and the $X$ matrix, the FDLS algorithm computes the second-order ($N=D=2$) transfer function coefficients vector $\theta_{M=8}$ as
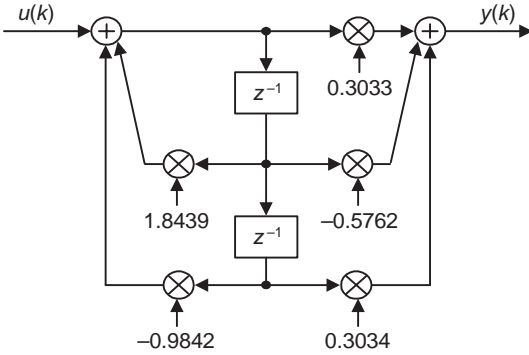
**Figure 7-4**    Second-order filter.
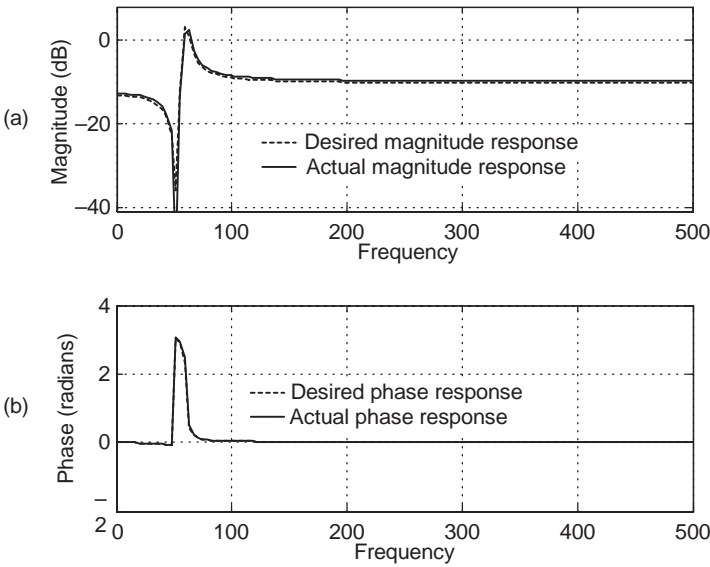


**Figure 7-5**    Actual versus desired filter magnitude and phase responses.

$$\theta_{M=8} = \begin{bmatrix} -1.8439 \\ 0.9842 \\ 0.3033 \\ -0.5762 \\ 0.3034 \end{bmatrix}.$$

Treated as filter coefficients, we can write vector $\theta_{M=8}$ as

$$a_0 = 1$$
$$a_1 = -1.8439$$

$$a_2 = 0.9842$$
$$b_0 = 0.3033$$
$$b_1 = -0.5762$$
$$b_2 = 0.3034$$

implemented as the recursive filter network shown in Figure 7-4.

The frequency-domain performance of the filter is in the solid curves shown in Figure 7-5. There we see that the $\theta_{M=8}$ coefficients provide an accurate approximation to the desired frequency response in Figure 7-3.

# Chapter 8

# Turbocharging Interpolated FIR Filters

**Richard Lyons**
*Besser Associates*

Interpolated finite impulse response (IFIR) filters—a significant innovation in the field of digital filtering—drastically improve the computational efficiency of traditional Parks-McClellan-designed lowpass FIR filters. IFIR filters is indeed a topic with which every digital filter designer needs to be familiar and, fortunately, tutorial IFIR filter material is available [1]–[5]. This chapter presents two techniques that improve the computational efficiency of IFIR filters.

Before we present the IFIR filter enhancement schemes, let's review the behavior and implementation of standard vanilla-flavored IFIR filters.

## 8.1 TRADITIONAL IFIR FILTERS

Traditional IFIR filters comprise a *band-edge shaping* subfilter in cascade with a lowpass *masking* subfilter as shown in Figure 8-1, where both subfilters are traditionally implemented as linear-phase tapped-delay FIR filters. The band-edge shaping subfilter has a sparse $h_{be}(k)$ impulse response, with all but every $L$th sample being zero, that shapes the final IFIR filter's passband, transition band, and stopband responses. (Integer $L$ is called the *expansion* or *stretch* factor of the band-edge shaping subfilter.) Because the band-edge shaping subfilter's frequency response contains $L–1$ unwanted periodic passband images, the masking subfilter is used to attenuate those images and can be implemented with few arithmetic operations.

To further describe IFIR filter behavior, consider the $h_{pr}(k)$ impulse response (the coefficients) of a tapped-delay line lowpass FIR *prototype* filter shown in Figure 8-2(a). That prototype filter has the frequency magnitude response shown in Figure
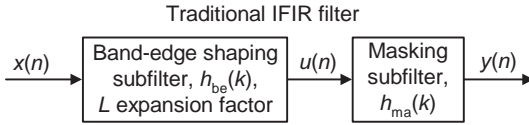
**Figure 8-1**   Traditional IFIR filter structure.
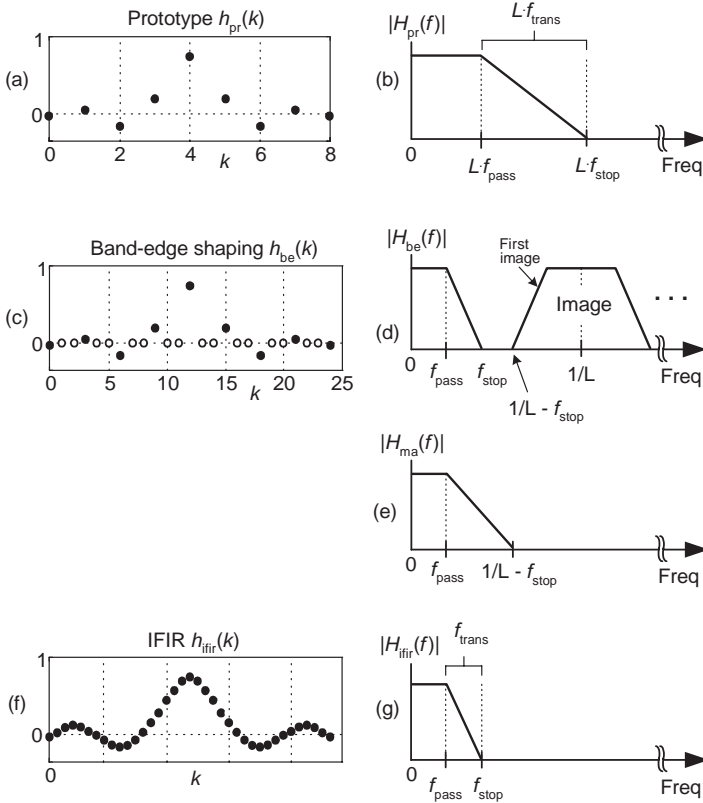


**Figure 8-2**   IFIR subfilters' impulse and frequency magnitude responses.

8-2(b). (The $f_{pass}$ and $f_{stop}$ frequency values are normalized to the filter input sample rate, $f_s$, in hertz. For example, $f_{pass}=0.1$ is equivalent to a cyclical frequency of $f_{pass}=f_s/10$ Hz.) To create a band-edge shaping subfilter, each of the prototype filter's unit delays are replaced with $L$-unit delays, with the expansion factor $L$ being an integer. If the $h_{pr}(k)$ impulse response of a 9-tap FIR prototype filter is that shown in Figure 8-2(a), the impulse response of the band-edge shaping subfilter, where, for example, $L=3$, is the $h_{be}(k)$ in Figure 8-2(c). The variable $k$ is merely an integer time-domain index where $0 \leq k \leq N-1$.

As we should expect, an $L$-fold expansion of a time-domain filter impulse response causes an $L$-fold compression (and repetition) of the frequency-domain

$|H_{pr}(f)|$ magnitude response as in Figure 8-2(d). Those repetitive passbands in $|H_{be}(f)|$ centered at integer multiples of $1/L$ ($f_s/L$ Hz)—for simplicity only one passband is shown in Figure 8-2(d)—are called *images*, and those images must be eliminated.

If we follow the band-edge shaping subfilter with a lowpass *masking* subfilter, having the frequency response shown in Figure 8-2(e), whose task is to attenuate the image passbands, we can realize a multistage filter whose $|H_{ifir}(f)|$ frequency magnitude response is shown in Figure 8-2(g). The cascaded $|H_{ifir}(f)|$ frequency magnitude response that we originally set out to achieve is

$$|H_{ifir}(f)| = |H_{be}(f)| \cdot |H_{ma}(f)|. \tag{8–1}$$

The cascade of the two subfilters is the so-called IFIR filter shown in Figure 8-1, with its cascaded impulse response given in Figure 8-2(f). Keep in mind, now, the $h_{ifir}(k)$ sequence in Figure 8-2(f) does not represent the coefficients used in an FIR filter. Sequence $h_{ifir}(k)$ is the convolution of the $h_{be}(k)$ and $h_{ma}(k)$ impulse responses (coefficients).

The goal in IFIR filter design is to find the optimum value for $L$, denoted as $L_{opt}$, that minimizes the total number of non-zero coefficients in the band-edge shaping and masking subfilters. Although design curves are available for estimating expansion factor $L_{opt}$ to minimize IFIR filter computational workload [1], [3], the following expression due to Mehrnia and Willson [6] enables $L_{opt}$ to be computed directly using:

$$L_{opt} = \frac{1}{f_{pass} + f_{stop} + \sqrt{f_{stop} - f_{pass}}}. \tag{8–2}$$

Note that once $L_{opt}$ is computed using (8-2) its value is then rounded to the nearest integer.

## 8.2  TWO-STAGE-MASKING IFIR FILTERS

Of this chapter's two techniques for improving the computational efficiency of IFIR filters, the first technique is a scheme called *two-stage masking*, where the masking subfilter in Figure 8-1 is itself implemented as an IFIR filter using the cascaded arrangement shown in Figure 8-3. This two-stage-masking method reduces the computational workload of traditional IFIR filters by roughly 20–30%.

The major design issue with the two-stage-masking method is that the expansion factors $L_1$ and $L_2$ are not directly related to a traditional IFIR filter's optimum $L_{opt}$ value. Fortunately, Reference [6] also provides a technique to determine the optimum values for $L_1$ and $L_2$ for the two-stage-masking design method.

## 8.3  RECURSIVE IFIR FILTERS

Our second technique to improve the computational efficiency of IFIR filters replaces Figure 8-1's masking subfilter with a cascade of subfilters as shown in Figure 8-4(a).
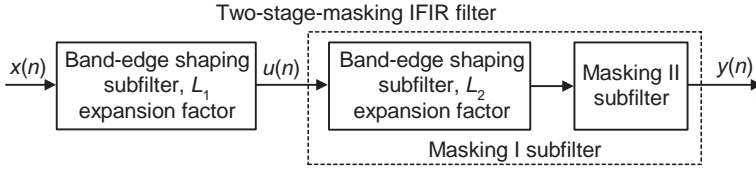
**Figure 8-3**    Two-stage-masking IFIR filter implementation.
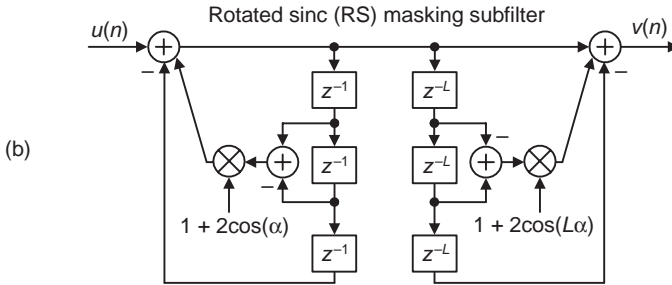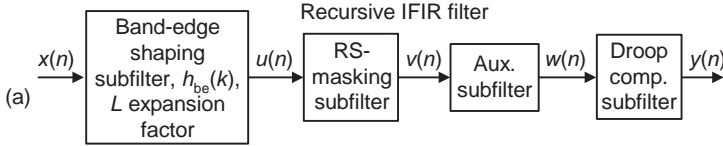


**Figure 8-4**    Recursive IFIR filter: (a) subfilters; (b) RS-masking subfilter structure.

The detailed structure of the RS-masking subfilter is shown in Figure 8-4(b), where $L$ is the band-edge shaping subfilter's expansion factor and a $z^{-L}$ block represents a cascade of $L$ unit-delay elements.

The RS-masking subfilter is called a *rotated sinc* (RS) filter and was originally proposed for use with sigma-delta A/D converters [7]. Factor $\alpha$ is the angular positions (in radians) of the subfilter's poles/zeros on the $z$-plane near $z=1$ as shown in Figure 8-5(a). The RS-masking subfilter's $z$-domain transfer function is

$$H_{RS}(z) = \frac{1 - Az^{-L} + Az^{-2L} - z^{-3L}}{1 - Bz^{-1} + Bz^{-2} - z^{-3}} \qquad (8\text{--}3)$$

where $A = 1 + 2\cos(L\alpha)$ and $B = 1 + 2\cos(\alpha)$. There are multiple ways to implement (8-3); however, the structure in Figure 8-4(b) is the most computationally efficient.

An RS-masking subfilter with $L=7$, for example, has transfer function zeros and poles as shown in Figure 8-5(a) with a total of $L$ triplets of zeros evenly spaced around the unit circle. The triplet of zeros near $z=1$ are overlaid with three poles where pole-zero cancellation makes our masking subfilter a lowpass filter,
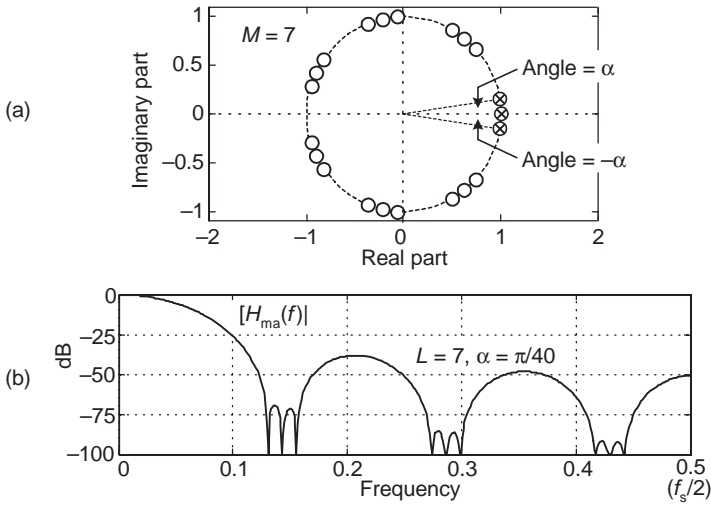
**Figure 8-5**    Masking subfilter: (a) pole/zero locations; (b) frequency magnitude response.

as illustrated by the $|H_{ma}|$ magnitude response curve in Figure 8-5(b) when $\alpha = \pi/40$, for example.

The expansion factor $L$ in (8-1) determines how many RS-masking subfilter response notches (triplets of zeros) are distributed around the unit circle, and angle $\alpha$ determines the width of those notches. For proper operation, angle $\alpha$ must be less than $\pi/L$. As it turns out, thankfully, the RS-masking subfilter's impulse response is both finite in duration and symmetrical so the subfilter exhibits linear phase.

## 8.4  MASKING ATTENUATION IMPROVEMENT

Because the RS-masking subfilter often does not provide sufficient notch attenuation, we can improve that attenuation, at a minimal computational cost, by applying the masking subfilter's $v(n)$ output to an *auxiliary* subfilter as shown in Figure 8-4(a). Using auxiliary subfilter I in Figure 8-6(a) we can achieve an additional 15–25 dB of notch attenuation. That auxiliary subfilter, when two's complement fixed-point math is used, is a guaranteed-stable cascaded integrator-comb (CIC) filter that places an additional $z$-domain transfer function zero at the center of each of the band-edge shaping subfilter's passband image frequencies.

To achieve greater than 90 dB of masking notch attenuation the auxiliary subfilter II in Figure 8-6(b) may be used. That subfilter places a pair of $z$-domain zeros within each triplet of zeros of the RS-masking subfilter (except near $z = 1$).

The RS-masking and auxiliary subfilters have some *droop* in their passband magnitude responses. If their cascaded passband droop is intolerable in your application, then some sort of passband-droop compensation must be employed as indicated in Figure 8-4(a). We could incorporate droop compensation in the design of the
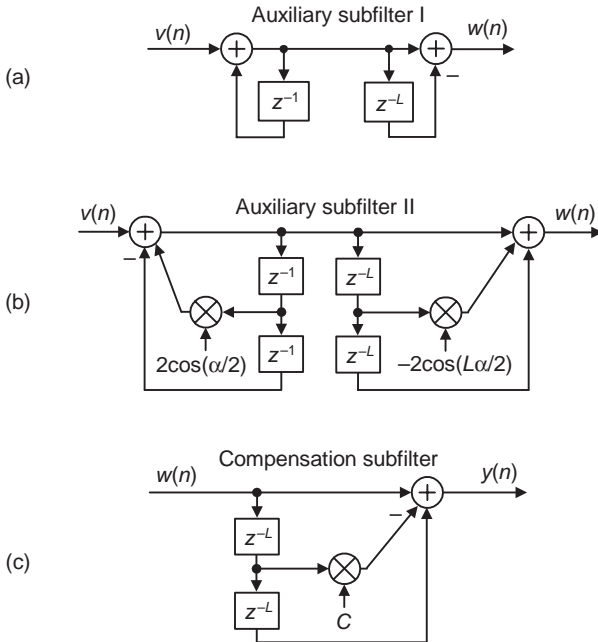
**Figure 8-6**   Recursive IFIR subfilters: (a) auxiliary subfilter I; (b) auxiliary subfilter II; (c) compensation subfilter.

band-edge shaping subfilter, but doing so drastically increases the order (the computational workload) of that subfilter. We could employ the *filter sharpening* technique (described in Chapter 1) to flatten the recursive IFIR filter's drooping passband response. However, that droop-compensation scheme significantly increases the computational workload and the number of delay elements needed in the recursive IFIR filter, as well as restricting the value of $L$ that can be used because we want integer-only delay line lengths. Rather, we suggest using the compensation subfilter shown in Figure 8-6(c), which was originally proposed for use with high-order CIC lowpass filters [8]. That compensation subfilter has a monotonically rising magnitude response beginning at zero Hz—just what we need for passband-droop compensation. The coefficient $C$, typically in the range of 4 to 10, is determined empirically.

## 8.5   RECURSIVE IFIR FILTER DESIGN EXAMPLE

We can further understand the behavior of a recursive IFIR filter by considering the design scenario where we desire a narrowband lowpass IFIR filter with $f_{pass}=0.01f_s$, a peak–peak passband ripple of 0.2 dB, a transition region bandwidth of $f_{trans}=0.005f_s$, and 90 dB of stopband attenuation. (A traditional Parks-McClellan-designed FIR lowpass filter satisfying these very demanding design specifications would require a 739-tap filter.) Designing a recursive IFIR filter to meet our design requirements
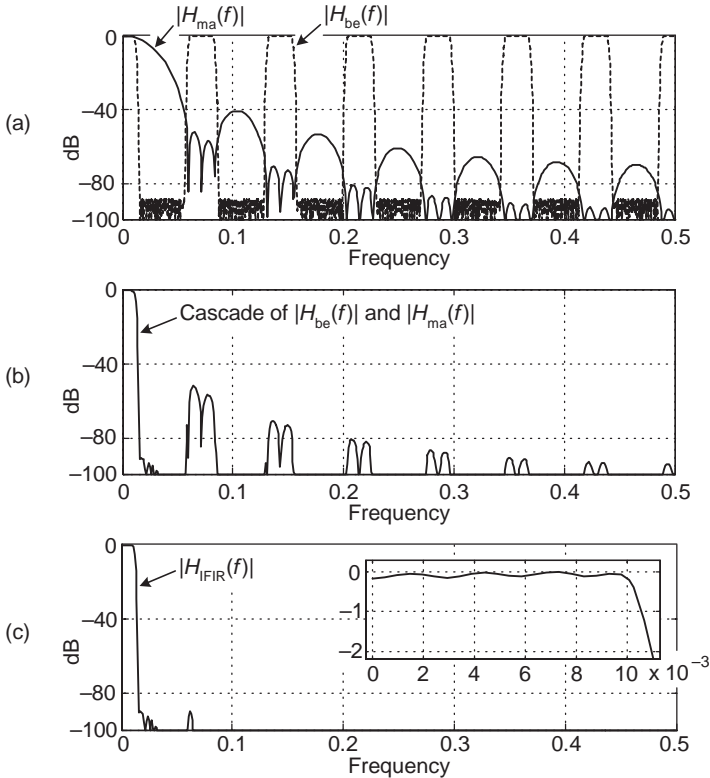
**Figure 8-7**    Recursive IFIR filter performance: (a) band-edge shaping and RS-masking subfilters'
responses; (b) response with no auxiliary filter; (c) response using cascaded auxiliary subfilter II and a
compensation subfilter.

yields an $L=14$ band-edge shaping subfilter having 52-taps, and an RS-masking
subfilter with $\alpha=\pi/42$ radians.

The dashed curve in Figure 8-7(a) shows the design example band-edge shaping
subfilter's $|H_{be}(f)|$ frequency magnitude response with its periodically spaced image
passbands that must be removed by the RS-masking subfilter. Selecting $\alpha=\pi/42$ for
the RS-masking subfilter yields the $|H_{ma}(f)|$ magnitude response shown by the solid
curve. The magnitude response of the cascade of just the band-edge shaping and the
RS-masking subfilters is shown in Figure 8-7(b). The magnitude response of our
final recursive IFIR filter, using auxiliary subfilter II and a $C=6$ compensation sub-
filter, is shown in Figure 8-7(c). The inset in Figure 8-7(c) shows the final recursive
IFIR filter's passband flatness measured in dB.

For comparison purposes, Table 8-1 lists the design example computational
workload per filter output sample for the various filter implementation options
discussed above. The "PM FIR" table entry [P1] means a single Parks-McClellan-
designed, tapped-delay line FIR filter. The "Efficiency gain" column indicates the
percent reduction in additions plus multiplications with respect to a standard IFIR

**Table 8-1**    Recursive IFIR Filter Design Example Computational Workload

| Lowpass filter: | Adds | Mults | Efficiency gain |
|---|---|---|---|
| PM FIR [Order=738] | 738 | 739 | — |
| Standard IFIR [L=10] | 129 | 130 | — |
| Two-stage-masking IFIR [L1=18, L2=4] | 95 | 96 | 26% |
| Recursive, RS-masking IFIR [L=14] | 63 | 57 | 54% |

filter. (Again, reference [6] proposed the smart idea of implementing a traditional IFIR filter's masking subfilter as, itself, a separate IFIR filter in order to reduce overall computational workload. The results of applying that *two-stage-masking* scheme to our IFIR filter design example are also shown in Table 8-1.)

As Table 8-1 shows, for cutting the computational workload of traditional IFIR filters the recursive IFIR filter is indeed a sharp knife.

## 8.6  RECURSIVE IFIR FILTER IMPLEMENTATION ISSUES

The practical issues to keep in mind when using a recursive IFIR filter are:

- Due to its coefficient symmetry, the band-edge shaping subfilter can be implemented with a *folded delay line* structure to reduce its number of multipliers by a factor of two.

- While the above two-stage-masking IFIR filter method can be applied to any desired linear-phase lowpass filter, the recursive IFIR filter scheme is more applicable to lowpass filters whose passbands are very narrow with respect to the filter's input sample rate. To optimize an IFIR filter, it's prudent to implement both the two-stage-masking and recursive IFIR filter schemes to evaluate their relative effectiveness.

- The gain of a recursive IFIR filter can be very large (in the hundreds or thousands), particularly for large $L$ and small $\alpha$, depending on which auxiliary subfilter is used. As such, the recursive IFIR filter scheme is best suited for floating-point numerical implementations. Two options exist that may enable a fixed-point recursive IFIR filter implementation: (1) when filter gain scaling methods are employed; and (2) swapping the internal feedback and feedforward sections of a subfilter to minimize data word growth; then (3) reduce the gains of auxiliary subfilter II and the compensation subfilter by a factor of $Q$ by changing their coefficients from [1, $-2\cos(L\alpha/2)$, 1] and [1, $-C$, 1] to [1/$Q$, $-2\cos(L\alpha/2)/Q$, 1/$Q$] and [1/$Q$, $-C/Q$, 1/$Q$]. If $Q$ is an integer power of two, then the subfilters' multiply by 1/$Q$ can be implemented with binary arithmetic right-shifts.

- Whenever we see filter poles lying on the $z$-domain's unit circle, as in Figure 8-5(a), we should follow Veronica's warning in the movie *The Fly*, and "Be

Figure 8-8 structure (a) Guaranteed stable RS-masking subfilter and (b) Guaranteed stable auxiliary subfilter II
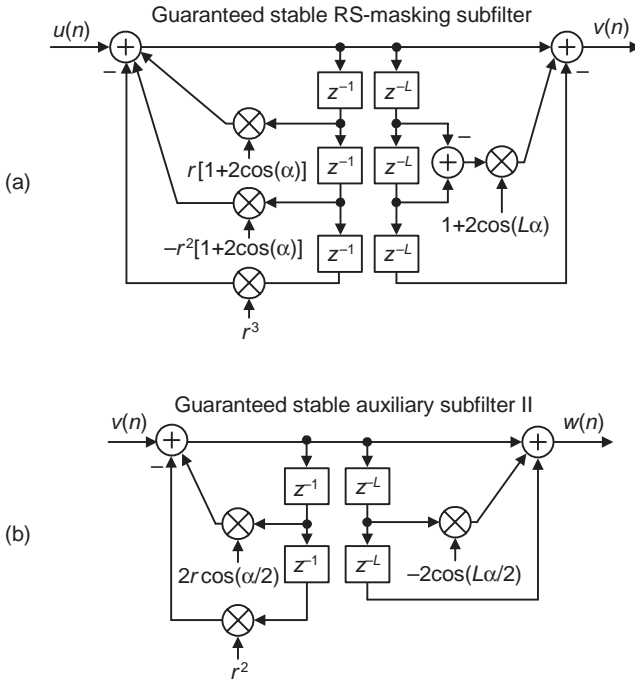
**Figure 8-8**  Guaranteeing stability: (a) stable RS-masking subfilter; (b) stable auxiliary subfilter II.

afraid. Be very afraid." Such filters run the risk of being unstable should our finite-precision filter coefficients cause a pole to lie just slightly outside the unit circle. If we find that our quantized-coefficient filter implementation is unstable, at the expense of a few additional multiplications per filter output sample we can use the subfilters shown in Figure 8-8 to guarantee stability while maintaining linear phase. The stability factor $r$ is a constant just as close to, but less than, one as our number format allows.

## 8.7  RECURSIVE IFIR FILTER DESIGN

Designing a recursive IFIR filter comprises the following steps:

1. Based on the desired lowpass filter's $f_{pass}$ and $f_{stop}$ frequencies, use (8-2) to determine a preliminary value for the band-edge shaping subfilter's integer expansion factor $L$.

2. Choose an initial value for the RS-masking subfilter's $\alpha$ using $\alpha = 2\pi f_{pass}$. Adjust $\alpha$ to maximize the attenuation of the band-edge shaping subfilter's passband images.

3. If the RS-masking subfilter does not provide sufficient passband image attenuation, employ one of the auxiliary filters in Figure 8-6.

4. Choose an initial value for $C$ (starting with $4<C<10$) for the compensation subfilter. Adjust $C$ to provide the desired passband flatness.

5. Continue by increasing L by one (larger values of L yield lower-order band-edge shaping subfilters) and repeat steps 2 through 4 until the either the RS-masking/auxiliary subfilter combination no longer supplies sufficient passband image attenuation or the compensation subfilter no longer can achieve acceptable passband flatness.

6. Sit back and enjoy a job well done.

## 8.8 RECURSIVE IFIR FILTERS FOR DECIMATION

If our lowpass filtering application requires the $y(n)$ output to be decimated, fortunately the RS-masking subfilter lends itself well to such a sample rate change process. To decimate $y(n)$ by $L$, we merely rearrange the order of the subfilters' elements so that all feedforward paths and the band-edge shaping subfilter follow the downsample-by-$L$ process as shown in Figure 8-9. Doing so has two advantages: First, the zero-valued coefficients in the band-edge shaping subfilter are eliminated, reducing that subfilter's order by a factor of $L$. (This converts the band-edge shaping subfilter back to the original prototype filter.) Second, the $z^{-L}$ delay lines in the other subfilters become $z^{-1}$ unit-delays, which reduces signal data storage requirements.
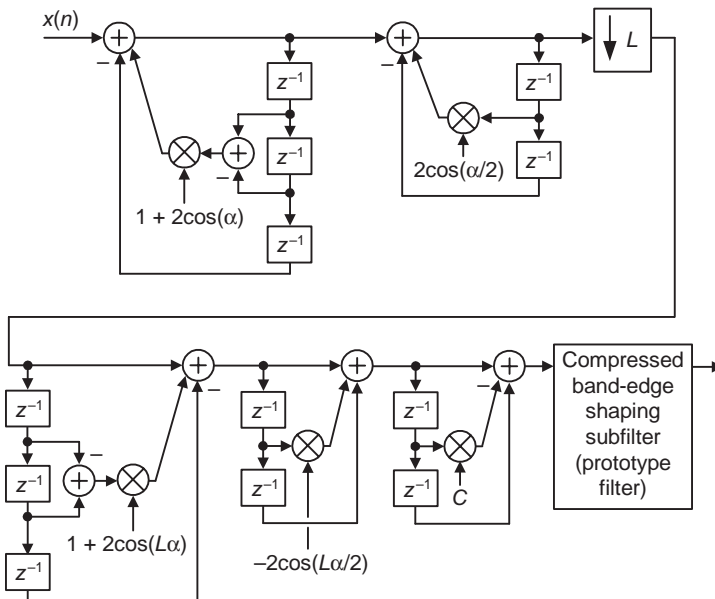


**Figure 8-9** RS-masking subfilter cascaded with the auxiliary II and compensation subfilters for decimation by $L$.

Modified CIC (MCIC) masking subfilter
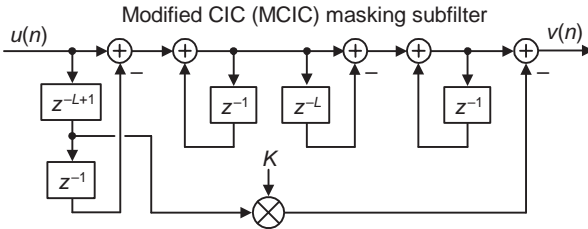


**Figure 8-10**   MCIC-masking subfilter.

In this decimation scenario the feedback multiplies by the $2\cos(\alpha)$ and $2\cos(\alpha/2)$ coefficients must be performed at the high input sample rate. Reference [7] discusses the possibility of replacing those coefficients with $2-2k$ in order to implement the multiplies with simple high-speed binary shifts and a subtraction.

## 8.9   AN ALTERNATE MASKING SUBFILTER

For completeness we introduce an alternate structure, shown in Figure 8-10, that can be used in place of the RS-masking subfilter in Figure 8-2(a). This *modified CIC* (MCIC) masking subfilter was inspired by, but is an optimized version of, a cascaded integrator-comb (CIC) filter proposed in reference [9].

The MCIC-masking subfilter, when two's complement fixed-point math is used, is a guaranteed-stable linear phase filter whose frequency response has an infinite-attenuation notch at the center of each of the band-edge shaping subfilter's passband image frequencies. Coefficient $K$ controls the width of the notches, and this subfilter's $z$-domain transfer function is

$$H_{\mathrm{MCIC}}(z) = \left[ \frac{1-z^{-L}}{1-z^{-1}} \right]^2 - Kz^{-L+1} \qquad (8\text{--}4)$$

When used in place of the RS-masking subfilter, a $K=6$ MCIC-masking subfilter also meets the stringent requirements of the above design example. While the MCIC-masking subfilter requires one fewer multiplier and fewer delay elements than the RS-masking subfilter, sadly the number of delay elements of an MCIC-masking subfilter is not reduced in decimation applications as in an RS-masking subfilter.

## 8.10   REFERENCES

[1]  R. LYONS, "Interpolated Narrowband Lowpass FIR Filters," *IEEE Signal Processing Magazine*: DSP Tips and Tricks Column, vol. 20, no. 1, January 2003, pp. 50–57.

[2]  R. LYONS, "Interpolated FIR Filters," *GlobalDSP On-line Magazine*, June 2003. [Online: http://www.globaldsp.com/index.asp?ID=8.]

[3]  R. LYONS, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 2004, pp. 319–331.

[4] F. Harris, *Multirate Signal Processing For Communication Systems*. Prentice Hall, Upper Saddle River, NJ, 2004, pp. 370–375.

[5] P. Vaidyanathan, *Multirate Systems and Filter Banks*. Prentice Hall PTR, Upper Saddle River, NJ, 1992, pp. 134–143.

[6] A. Mehrnia and A. Willson Jr., "On Optimal IFIR Filter Design," *Proc. of the 2004 International Symp. on Circuits and Systems (ISCAS)*, vol. 3, 23–26 May 2004, pp. 133–136.

[7] L. Lo Presti, "Efficient Modified-Sinc Filters for Sigma-Delta A/D Converters," *IEEE Trans. on Circuits and Systems—II: Analog and Digital Signal Proc.*, vol. 47, no. 11, November 2000, pp. 1204–1213.

[8] H. Oh, S. Kim, G. Choi, and Y. Lee, "On the Use of Interpolated Second-Order Polynomials for Efficient Filter Design in Programmable Downconversion," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 4, April 1999, pp. 551–560.

[9] T. Saramäki, Y. Neuvo, and S. Mitra, "Design of Computationally Efficient Interpolated FIR Filters," *IEEE Trans. on Circuits and Systems*, vol. 35, no. 1, January 1988, pp. 70–88.

# Chapter 9

# A Most Efficient Digital Filter: The Two-Path Recursive All-Pass Filter

**Fred Harris**
*San Diego State University*

**M**any of us are introduced to digital recursive filters as mappings from analog prototype structures mapped to the sample data domain by the bilinear Z-Transform. These digital filters are normally implemented as a cascade of canonic second-order filters that independently form its two poles and two zeros with two feedback and two feedforward coefficients respectively. In this chapter we discuss an efficient alternative recursive filter structure based on simple recursive all-pass filters that use a single coefficient to form both a pole and a zero or to form two poles and two zeros.

An all-pass filter has unity gain at all frequencies and otherwise exhibits a frequency-dependent phase shift. We might then wonder, if the filter has unity gain at all frequencies, how it can form a stopband. We accomplish this by adjusting the phase in each path of a two-path filter to obtain destructive cancellation of signals occupying specific spectral bands. Thus the stopband zeros are formed by the destructive cancellation of components in the multiple paths rather than as explicit polynomial zeros. This approach leads to a wide class of very efficient digital filters that require only 25% to 50% of the computational workload of the standard cascade of canonic second-order filters. These filters also permit the interchange of the resampling and filtering to obtain further workload reductions.

---

## 9.1    ALL-PASS NETWORK BACKGROUND

All-pass networks are the building blocks of every digital filter [1]. All-pass networks exhibit unity gain at all frequencies and a phase shift that varies as a function of frequency. All-pass networks have poles and zeros that occur in (conjugate) reciprocal pairs. Since all-pass networks have reciprocal pole-zero pairs, the numerator and denominator are seen to be reciprocal polynomials. If the denominator is an $N$th-order polynomial $P_N(Z)$, the reciprocal polynomial in the numerator is $Z^N P_N(Z^{-1})$. It is easily seen that the vector of coefficients that represent the denominator polynomial is reversed to form the vector representing the numerator polynomial. A cascade of all-pass filters is also seen to be an all-pass filter. A sum of all-pass networks is not all-pass, and we use these two properties to build our class of filters. Every all-pass network can be decomposed into a product of first- and second-order all-pass networks; thus it is sufficient to limit our discussion to first- and second-order filters, which we refer to as Type I and Type II respectively. Here we limit our discussion to first- and second-order polynomials in $Z$ and $Z^2$. The transfer functions of Type-I and Type-II all-pass networks are shown in (9–1) with the corresponding pole-zero diagrams shown in Figure 9-1.

$$H_1(Z) = \frac{(1+\alpha Z)}{(Z+\alpha)}, \qquad H_1(Z^2) = \frac{(1+\alpha Z^2)}{(Z^2+\alpha)}: \qquad \text{Type I}$$

$$H_2(Z) = \frac{(1+\alpha_1 Z + \alpha_2 Z^2)}{(Z^2 + \alpha_1 Z + \alpha_2)}, \qquad H_2(Z^2) = \frac{(1+\alpha_1 Z^2 + \alpha_2 Z^4)}{(Z^4 + \alpha_1 Z^2 + \alpha_2)}: \quad \text{Type II}$$

$$(9–1)$$

Note that the single sample delay with $Z$-transform $Z^{-1}$ (or $1/Z$) is a special case of the Type-I all-pass structure obtained by setting the coefficient $\alpha$ to zero. Linear
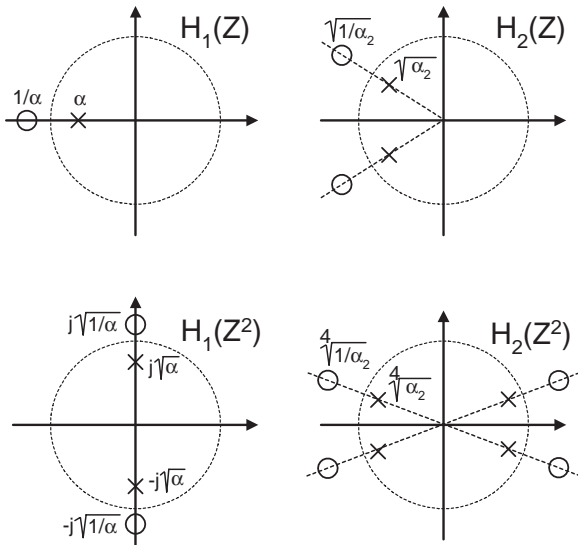


**Figure 9-1**    Pole-zero structure of Type-I and Type-II all-pass filters of degrees 1 and 2.

phase delay is all that remains as the pole of this structure approaches the origin while its zero simultaneously approaches infinity. We use the fundamental all-pass filter ($Z^{-1}$) as the starting point of our design process and then develop the more general case of the Type-I and Type-II all-pass networks to form our class of filters.

A closed-form expression for the phase function of the Type-I transfer function is obtained by evaluating the transfer function on the unit circle. The result of this exercise is shown in (9–2).

$$\phi = -2\operatorname{atan}\left[\frac{(1+\alpha)}{(1-\alpha)}\tan\left(M\frac{\theta}{2}\right)\right], M = 1, 2.\tag{9–2}$$

The phase response of the first- and second-order all-pass Type-I structures is shown in Figures 9-2(a) and (b).
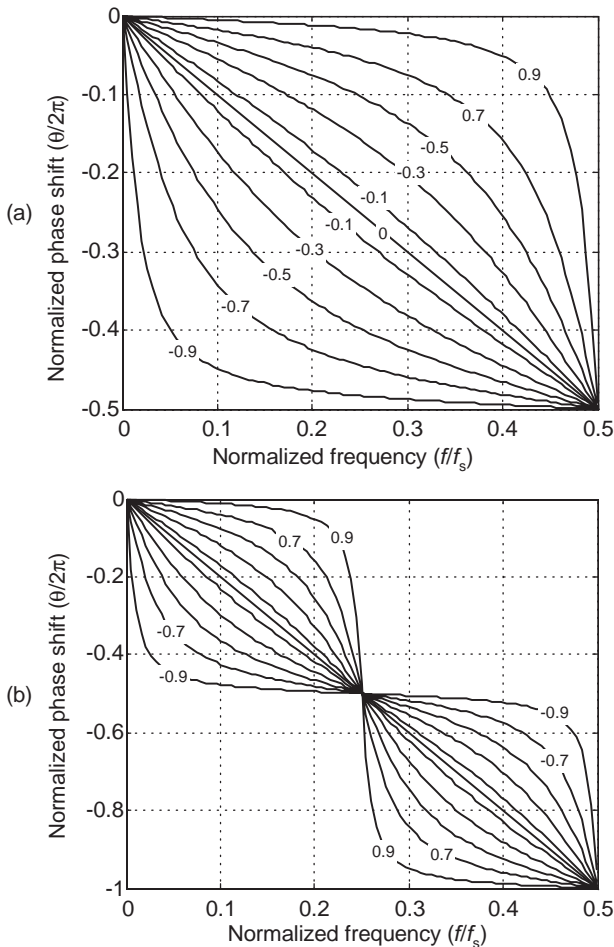


**Figure 9-2** (a) Phase response of Type I all-pass filter, first-order polynomial in $Z^{-1}$, as function of coefficient $\alpha$, ($\alpha$=0.9, 0.7, . . . , −0.7, −0.9); (b) phase response of Type II all-pass filter, first-order polynomial in $Z^{-2}$, as function of coefficient $\alpha$, ($\alpha$=0.9, 0.7, . . . , −0.7, −0.9).

Note that for the first-order polynomial in $Z^{-1}$, the transfer function for $\alpha=0$ defaults to the pure delay, and as expected, its phase function is linear with frequency. The phase function will vary with $\alpha$ and this network can be thought of, and will be used as, the generalized delay element. We observe that the phase function is anchored at its end points (0° at zero frequency and 180° at the half sample rate) and that it warps with variation in $\alpha$. It bows upward (less phase) for positive $\alpha$ and bows downward (more phase) for negative $\alpha$. The bowing phase function permits us to use the generalized delay to obtain a specified phase shift angle at any frequency. For instance, we note that when $\alpha=0$, the frequency for which we realize the 90-degree phase shift is 0.25 (the quarter sample rate). We can determine a value of $\alpha$ for which the 90-degree phase shift is obtained at any normalized frequency such as at normalized frequency 0.45 ($\alpha=0.8$) or at normalized frequency 0.05 ($\alpha=-0.73$).

## 9.2  IMPLEMENTING AND COMBINING ALL-PASS NETWORKS

While the Type-I all-pass network can be implemented in a number of architectures we limit the discussion to the one shown in Figure 9-3. This structure has a number of desirable implementation attributes that are useful when multiple stages are placed in cascade. We observe that the single multiplier resides in the feedback loop of the lower delay register to form the denominator of the transfer function. The single multiplier also resides in the feedforward path of the upper delay register to form the numerator of the transfer function. The single multiplier thus forms all the poles and zeros of this all-pass network and we call attention to this in the equivalent processing block to the right of the filter block diagram.

## 9.3  TWO-PATH FILTERS

While the $M$th-order all-pass filter finds general use in an $M$-path polyphase structure, we restrict this discussion to two-path filters. We first develop an understanding of the simplest two-path structure and then expand the class by invoking a simple set of frequency transformations. The structure of the two-path filter is presented in Figure 9-4. Each path is formed as a cascade of all-pass filters in powers of $Z^2$. The
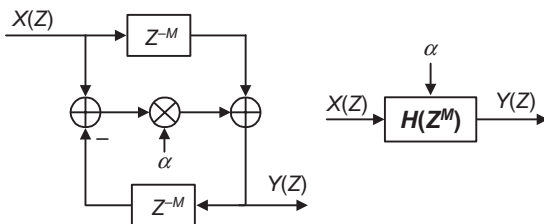


**Figure 9-3**    Single coefficient Type-I all-pass filter structure.
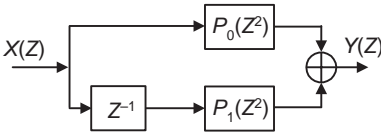
**Figure 9-4**    Two-path polyphase filter.
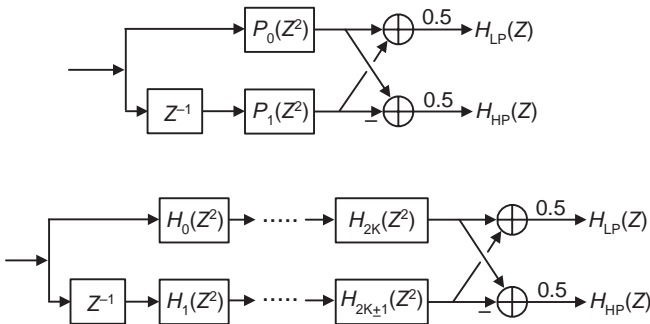


**Figure 9-5**    Two-path all-pass filter.

delay in the lower path can be placed on either side of the all-pass network. When the filter is implemented as a multirate device, the delay is positioned on the side of the filter operating at the higher of the two rates where it is absorbed by the input (or output) commutator.

This deceivingly simple filter structure offers a surprisingly rich class of filter responses. The two-path all-pass structure can implement halfband lowpass and highpass filters, as well as Hilbert transform filters that exhibit minimum or non-minimum phase response. The two-path filter implements standard recursive filters such as the *Butterworth* and the *elliptical* filters. A MATLAB routine, *tony_des2*, that computes the coefficients of the two-path filter and a number of its frequency-transformed variants is available from the author via an e-mail request. Also, as suggested earlier, the halfband filters can be configured to embed a 1-to-2 upsampling or a 2-to-1 downsampling operation within the filtering process.

The prototype halfband filters have their 3 dB band edge at the quarter sample rate. All-pass frequency transformations applied to the two-path prototype form arbitrary-bandwidth lowpass and highpass complementary filters, and arbitrary center frequency passband and stopband complementary filters. Zero packing the time response of the two-path filter, another trivial all-pass transformation, causes spectral scaling and replication. The zero-packed structure is used in cascade with other filters in iterative filter designs to achieve composite spectral responses exhibiting narrow-transition bandwidths with low-order filters.

The specific form of the prototype halfband two-path filter is shown in Figure 9-5. The number of poles (or order of the polynomials) in the two paths differ by precisely one, a condition assured when the number of filter segments in the lower leg is equal to or is one less than the number of filter segments in the upper leg. The

structure forms complementary lowpass and highpass filters as the scaled sum and difference of the two paths.

The transfer function of the two-path filter shown in Figure 9-5 is shown in (9–3).

$$H(Z) = P_0(Z^2) \pm Z^{-1} P_1(Z^2)$$

$$P_i(Z^2) = \prod_{k=0}^{K_i} H_{i,k}(Z^2), \quad i = 0, 1$$

$$H_{i,k}(Z^2) = \frac{1 + \alpha(i, k)Z^2}{Z^2 + \alpha(i, k)}$$

(9–3)

In particular, we can examine the simple case of two all-pass filters in each path. The transfer function for this case is shown in (9–4).

$$H(Z) = \frac{1 + \alpha_0 Z^2}{Z^2 + \alpha_0} \frac{1 + \alpha_2 Z^2}{Z^2 + \alpha_2} \pm \frac{1}{Z} \frac{1 + \alpha_1 Z^2}{Z^2 + \alpha_1} \frac{1 + \alpha_3 Z^2}{Z^2 + \alpha_3}$$

$$= \frac{b_0 Z^9 \pm b_1 Z^8 + b_2 Z^7 \pm b_3 Z^6 + b_4 Z^5 \pm b_4 Z^4 + b_3 Z^3 \pm b_2 Z^2 + b_1 Z^1 \pm b_0}{Z(Z^2 + \alpha_0)(Z^2 + \alpha_1)(Z^2 + \alpha_2)(Z^2 + \alpha_3)}$$

(9–4)

We note a number of interesting properties of this transfer function, applicable to all the two-path prototype filters. The denominator roots are on the imaginary axis restricted to the interval ±1 to assure stability. The numerator is a linear-phase FIR filter with a symmetric weight vector. As such the numerator roots must appear either on the unit circle, or if off and real, in reciprocal pairs, and if off and complex, in reciprocal conjugate quads. Thus for appropriate choice of the filter weights, the zeros of the transfer function can be placed on the unit circle, and can be distributed to obtain an equal-ripple stopband response. In addition, due to the one-pole difference between the two paths, the numerator must have a zero at ±1. When the two paths are added, the numerator roots are located in the left half-plane, and when subtracted, the numerator roots are mirror imaged to the right half-plane forming lowpass and highpass filters respectively.

The attraction of this class of filters is the unusual manner in which the transfer-function zeros are formed. The zeros of the all-pass subfilters reside outside the unit circle (at the reciprocal of the stable pole positions) but migrate to the unit circle as a result of the sum or difference of the two paths. The zeros appear on the unit circle because of destructive cancellation of spectral components delivered to the summing junction via the two distinct paths, as opposed to being formed by numerator weights in the feedforward path of standard biquadratic filters. The stopband zeros are a windfall. They start as the maximum phase all-pass zeros formed concurrently with the all-pass denominator roots by a single shared coefficient and migrate to the unit circle in response to addition of the path signals. This is the reason that the two-path filter requires less than half the multiplies of the standard biquadratic filter.

Figure 9-6 presents the pole-zero diagram for this filter. The composite filter contains nine poles and nine zeros and requires two coefficients for path-0 and two coefficients for path-1. The *tony_des2* design routine was used to compute weights
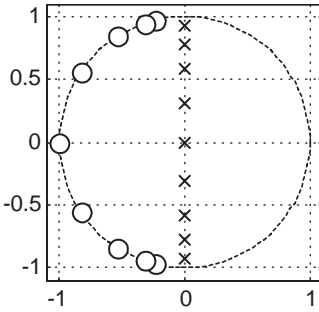
**Figure 9-6**    Pole-zero diagram of two-path, nine-pole, four-multiplier filter.

for the ninth-order filter with −60 dB equal ripple stopband. The passband edge is located at a normalized frequency of 0.25 and the stopband edge that achieved the desired 60 dB stopband attenuation is located at a normalized frequency of 0.284. This is an elliptical filter with constraints on the pole positions. The denominator coefficient vectors of the filter are listed here in decreasing powers of Z:

**Path-0 Polynomial Coefficients:**

Filter-0 [1    0    0.101467517]

Filter-2 [1    0    0.612422841]

**Path-1 Polynomial Coefficients:**

Filter-1 [1    0    0.342095596]

Filter-3 [1    0    0.867647439]

The roots presented here represent the lowpass filter formed from the two-path filter. Figure 9-7 presents the phase slopes of the two paths of this filter as well as the filter frequency response. We note that the zeros of the spectrum correspond to the zero locations on the unit circle in the pole-zero diagram.

The first subplot in Figure 9-7 presents two sets of phase responses for each path of the two-path filter. The dashed lines represent the phase response of the two paths when the filter coefficients are set to zero. In this case, the two paths default to two delays in the top path and three delays in the bottom path. Since the two paths differ by one delay, the phase shift difference is precisely 180° at the half-sample rate. When the filter coefficients in each path are adjusted to their design values, the phase response of both paths assumes the bowed "lazy S" curve described earlier in Figure 9-2(b). Note that at low frequencies, the two phase curves exhibit the same phase profile, and that at high frequencies, the two-phase curves maintain the same 180-degree phase difference. Thus the addition of the signals from the two paths will lead to a gain of 2 in the band of frequencies with the same phase and will result in destructive cancellation in the band of frequencies with 180-degree phase difference. These two bands of course are the passband and stopband respectively. We note that the two phase curves differ by exactly 180° at four distinct frequencies as well as the half-sample rate: those frequencies corresponding to the spectral zeros
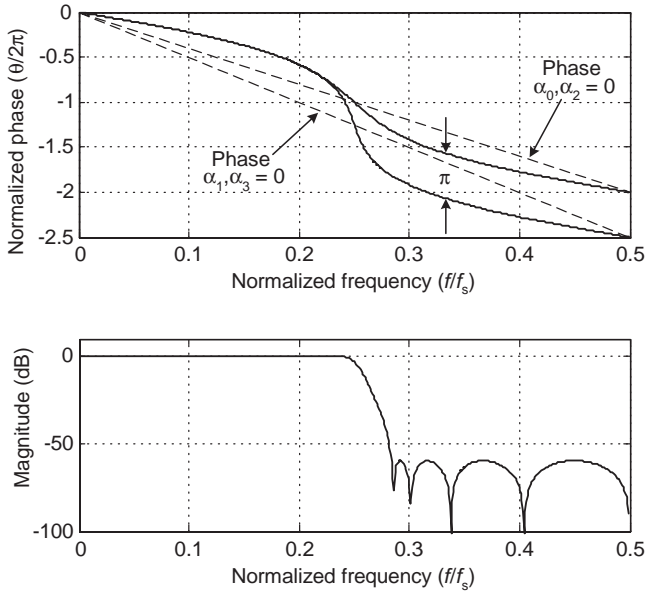
**Figure 9-7**    Phase slopes and frequency response of two-path, nine-pole, four-multiplier filter.

of the filter. Between these zeros, the filter exhibits stopband side lobes that, by design, are equal ripple.

## 9.4    LINEAR-PHASE TWO-PATH HALFBAND FILTERS

We can modify the structure of the two-path filter to form filters with approximately linear phase response by restricting one of the paths to be pure delay. We accomplish this by setting all the filter coefficients in the upper leg to zero. This sets the all-pass filters in this leg to their default responses of pure delay with poles at the origin. As we pursue the solution to the phase-matching problem in the equal-ripple approximation we find that the all-pass poles must move off the imaginary axis. In order to keep real coefficients for the all-pass filters, we call on the Type-II all-pass filter structure. The lower path then contains first- and second-order filters in $Z^2$. We lose a design degree of freedom when we set the phase slope in one path to be a constant. Consequently, when we design an equal-ripple group delay approximation to a specified performance we require additional all-pass sections. To meet the same out-of-band attenuation and the same stopband band-edge as the nonlinear phase design of the previous section, our design routine, *lineardesign*, determined that we require two first-order filters in $Z^2$ and three second-order filters in $Z^2$. This means that eight-coefficients are required to meet the specifications that in the nonlinear phase design required only four coefficients. Path-0 (the delay-only path) requires 16 units of

delay while the all-pass denominator coefficient vector list is presented below in decreasing powers of Z, which, along with its single delay element, form a seventeenth-order denominator.

**Path-0 Polynomial Coefficients:**

Delay   [zeros(1,16)   1]

**Path-1 Polynomial Coefficients:**

Filter-0 [1   0     0.832280776]

Filter-1 [1   0   −0.421241137]

Filter-2 [1   0     0.67623706   0   0.23192313]

Filter-3 [1   0     0.00359228   0   0.19159423]

Filter-4 [1   0   −0.59689082   0   0.18016931]

Figure 9-8 presents the pole-zero diagram of the linear-phase all-pass filter structure that meets the same spectral characteristics as those outlined in the previous section. We first note that the filter is nonminimum phase due to the zeros outside the unit circle. We also note the near cancellation of the right half-plane pole cluster with the reciprocal zeros of the nonminimum phase zeros. Figure 9-9 presents the phase slopes of the two filter paths and the filter frequency response. We first note that the phase of the two paths is linear; consequently, the group delay is constant over the filter passband. The constant group delay matches the time delay to the peak of the impulse response, which corresponds to the 16-sample time delay of the top path. Of course, the spectral zeros of the frequency response coincide with the transfer-function zeros on the unit circle.
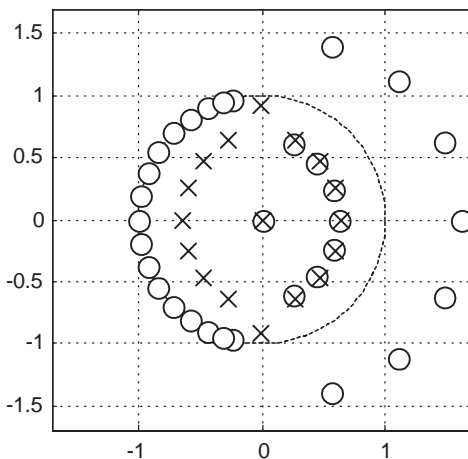


**Figure 9-8**    Pole-zero diagram of two-path, 33-pole, 8-multiplier filter.
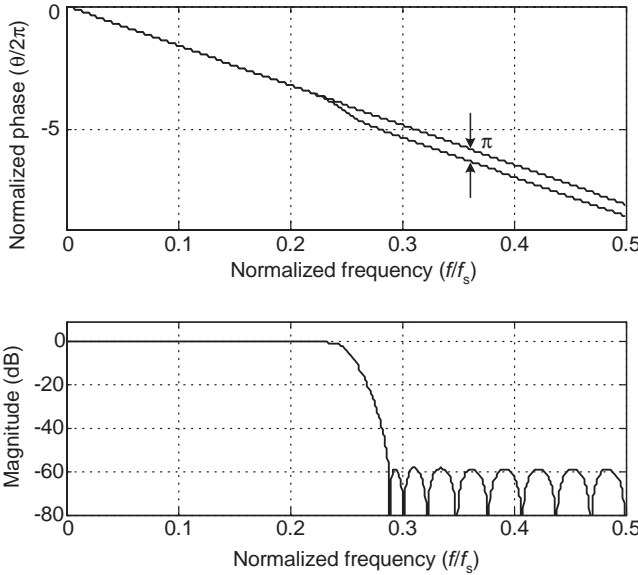
**Figure 9-9**    Two-path, 33-pole, 8-multiplier filter: phase slopes; and frequency response.
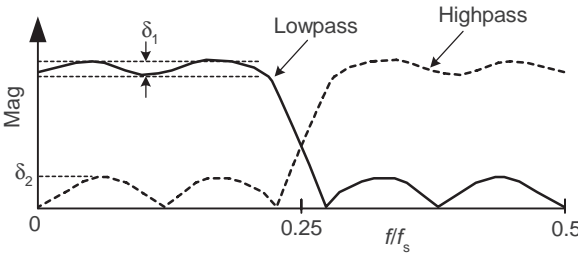


**Figure 9-10**    Magnitude response of lowpass and highpass halfband filter.

## 9.5  PASSBAND RESPONSE IN TWO-PATH HALFBAND FILTERS

The all-pass networks that formed the halfband filter exhibit unity gain at all frequencies. These are lossless filters affecting only the phase response of signal they process. This leads to an interesting relationship between the passband and stopband ripple response of the halfband filter and in fact for any of the two-path filters discussed in this chapter. We noted earlier that the two-path filter presents complementary lowpass and highpass versions of the halfband filter, the frequency responses of which are shown in Figure 9-10, where passband and stopband ripples have been denoted by $\delta_1$ and $\delta_2$ respectively.

The transfer functions of the lowpass and of the highpass filters are shown in (9–5), where $P_0(Z)$ and $P_1(Z)$ are the transfer functions of the all-pass filters in each

of the two paths. The power gain of the lowpass and highpass filters is shown in (9–6). When we form the sum of the power gains, the cross terms in the pair of products cancel and we obtain the results shown in (9–7).

$$H_{LOW}(Z) = 0.5 \cdot [P_0(Z) + Z^{-1}P_1(Z)]$$
$$H_{HIGH}(Z) = 0.5 \cdot [P_0(Z) - Z^{-1}P_1(Z)] \tag{9–5}$$

$$
\begin{aligned}
|H_{LOW}(Z)|^2 &= H_{LOW}(Z)H_{LOW}(Z^{-1}) \\
&= 0.25 \cdot [P_0(Z) + Z^{-1}P_1(Z)] \cdot [P_0(Z^{-1}) + ZP_1(Z^{-1})]
\end{aligned}
$$

$$
\begin{aligned}
|H_{HIGH}(Z)|^2 &= H_{HIGH}(Z)H_{HIGH}(Z^{-1}) \\
&= 0.25 \cdot [P_0(Z) - Z^{-1}P_1(Z)] \cdot [P_0(Z^{-1}) - ZP_1(Z^{-1})]
\end{aligned} \tag{9–6}
$$

$$|H_{LOW}(Z)|^2 + |H_{HIGH}(Z)|^2 = 0.25 \cdot [2 \cdot |P_0(Z)|^2 + 2 \cdot |P_1(Z)|^2] = 1 \tag{9–7}$$

Equation (9–7) tells us that at any frequency, the squared magnitude of the lowpass gain and the squared magnitude of the highpass gain sum to unity. This is a consequence of the filters being lossless. Energy that enters the filter is never dissipated; a fraction of it is available at the lowpass output and the rest of it is available at the highpass output. This property is the reason the complementary lowpass and highpass filters cross at their 3-dB points. If we substitute the gains at peak ripple of the lowpass and highpass filters into (9–7), we obtain (9–8), which we can rearrange and solve for the relationship between $\delta_1$ and $\delta_2$. The result is interesting. We learn here that the peak-to-peak in-band ripple is approximately half the square of the out-of-band peak ripple. Thus, if the out-of-band ripple is −60 dB or 1 part in 1000, then the in-band peak-to-peak ripple is half of 1 part in one million, which is on the order of 5 μ-dB (4.34 μ-dB). The halfband recursive all-pass filter exhibits an extremely small in-band ripple. The in-band ripple response of the two-path nine-pole filter is seen in Figure 9-11.

$$[1 - \delta_1]^2 + [\delta_2]^2 = 1$$
$$[1 - \delta_1] = \sqrt{1 - \delta_2^2} \cong 1 - 0.5 \cdot \delta_2^2$$
$$\delta_1 \cong 0.5 \cdot \delta_2^2 \tag{9–8}$$
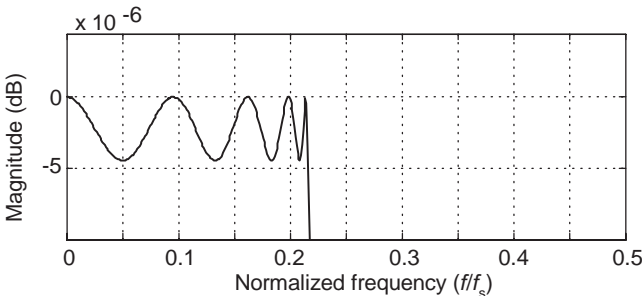


**Figure 9-11**    In-band ripple level of two-path, nine-pole recursive filter.

## 9.6  TRANSFORMING THE PROTOTYPE HALFBAND TO ARBITRARY BANDWIDTH

In the previous section we examined the design of two-path halfband filters formed from recursive all-pass first-order filters in the variable $Z^2$. We did this because we have easy access to the weights of this simple constrained filter, the constraint being stated in (9–5). If we include a requirement that the stopband be equal ripple, the halfband filters we examine are elliptical filters that can be designed from standard design routines. Our program *tony_des2* essentially does this in addition to the frequency transformations we are about to examine. The prototype halfband filter can be transformed to form other filters with specified (arbitrary) bandwidth and center frequency. In this section, elementary frequency transformations are introduced and their impact on the prototype architecture as well as on the system response is reviewed. In particular, the frequency transformation that permits bandwidth tuning of the prototype is introduced first. Additional transformations that permit tuning of the center frequency of the prototype filter are also discussed.

## 9.7  LOWPASS-TO-LOWPASS TRANSFORMATION

We now address the transformation that converts a lowpass halfband filter to a lowpass arbitrary-bandwidth filter. Frequency transformations occur when an existing all-pass subnetwork in a filter is replaced by another all-pass subnetwork. In particular, we present the transformation shown in (9–9).

$$\frac{1}{Z} \Rightarrow \frac{1+bZ}{Z+b}; \quad b = \frac{1-\tan(\theta_b/2)}{1+\tan(\theta_b/2)}; \quad \theta_b = 2\pi\frac{f_b}{f_S} \tag{9–9}$$

This is the generalized delay element we introduced in the initial discussion of first-order all-pass networks. We can physically replace each delay in the prototype filter with the all-pass network and then tune the prototype by adjusting the parameter "$b$". We have fielded many designs in which we perform this substitution. Some of these designs are cited in the bibliography. For the purpose of this discussion, we perform the substitution algebraically in the all-pass filters comprising the two-path halfband filter, and in doing so generate a second structure for which we will develop and present an appropriate architecture.

 We substitute (9–9) into the first-order in $Z^2$ all-pass filter introduced in (9–2) and rewritten in (9–10).

$$G(Z) = H(Z^2)\big|_{Z \Rightarrow \frac{Z+b}{1+bZ}}$$

$$G(Z) = \frac{1+\alpha Z^2}{Z^2+\alpha}\bigg|_{Z \Rightarrow \frac{Z+b}{1+bZ}} \tag{9–10}$$

After performing the indicated substitution and gathering terms, we find the form of the transformed transfer function is as shown in (9–11).

$$G(Z) = \frac{1 + c_1 Z + c_2 Z^2}{Z^2 + c_1 Z + c_2}; \quad c_1 = \frac{2b(1+\alpha)}{1+\alpha b^2}; \quad c_2 = \frac{b^2 + \alpha}{1+\alpha b^2} \tag{9-11}$$

As expected, when $b \to 0$, $c_1 \to 0$, and $c_2 \to a$, the transformed all-pass filter reverts back to the original first-order filter in $Z^2$. The architecture of the transformed filter, which permits one multiplier to form the matching numerator and denominator coefficient simultaneously, is shown in Figure 9-12. Also shown is a processing block $G(Z)$ that uses two coefficients $c_1$ and $c_2$. This is seen to be an extension of the one-multiply structure presented in (9–3). The primary difference in the two architectures is the presence of the coefficient and multiplier $c_1$ associated with the power of $Z^{-1}$. This term, formerly zero, is the sum of the polynomial roots, and hence is minus twice the real part of the roots. With this coefficient being non-zero, the roots of the polynomial are no longer restricted to the imaginary axis.

The root locations of the transformed, or generalized, second-order all-pass filter are arbitrary except that they appear as conjugates inside the unit circle, and the poles and zeros appear in reciprocal sets as indicated in Figure 9-13.

The two-path prototype filter contained one or more one-multiply first-order recursive filters in $Z^2$ and a single delay. We effect a frequency transformation on the prototype filter by applying the lowpass-to-lowpass transformation shown in (9–10). Doing so converts the one-multiply first-order in $Z^2$ all-pass filter to the



**Figure 9-12** Block diagram of general second-order all-pass filter.
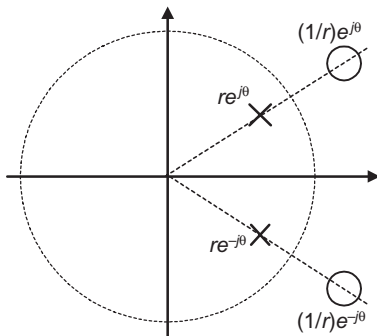


**Figure 9-13** Pole-zero diagram of generalized second-order all-pass filter.
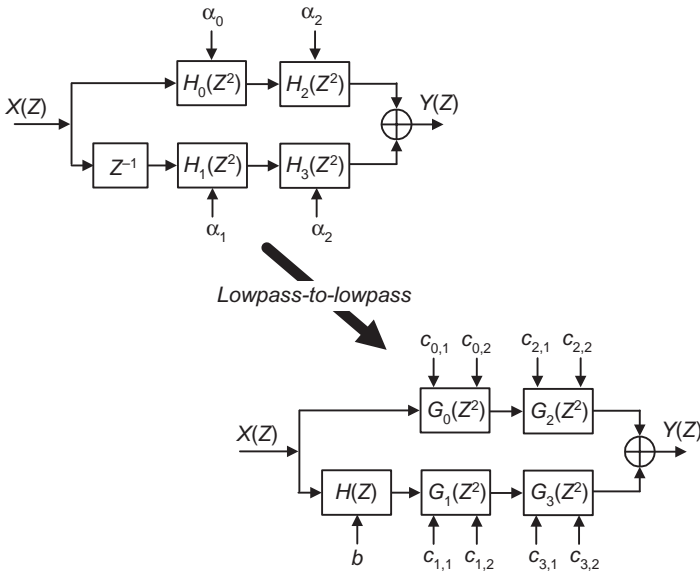
**Figure 9-14** Effect on architecture of frequency transformation applied to two-path halfband all-pass filter.
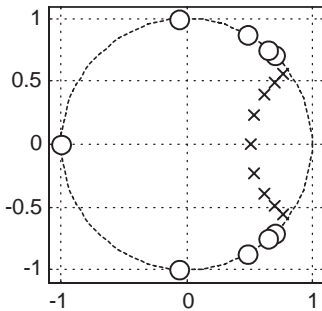


**Figure 9-15** Pole-zero diagram obtained by frequency transforming halfband filter to normalized frequency 0.1.

generalized two-multiply second-order all-pass filter and converts the delay, a zero-multiply all-pass filter, to the generalized one-multiply first-order in $Z$ all-pass filter. Figure 9-14 shows how applying the frequency transformation affects the structure of the prototype. Note that the nine-pole, nine-zero halfband filter, which is implemented with only four multipliers, now requires nine multipliers to form the same nine poles and nine zeros for the arbitrary-bandwidth version of the two-path network. This is still significantly less than the standard cascade of first- and second-order canonic filters for which the same nine-pole, nine-zero filter would require 23 multipliers.

Figure 9-15 presents the pole-zero diagram of the frequency-transformed prototype filter. The nine poles have been pulled off the imaginary axis, and the nine
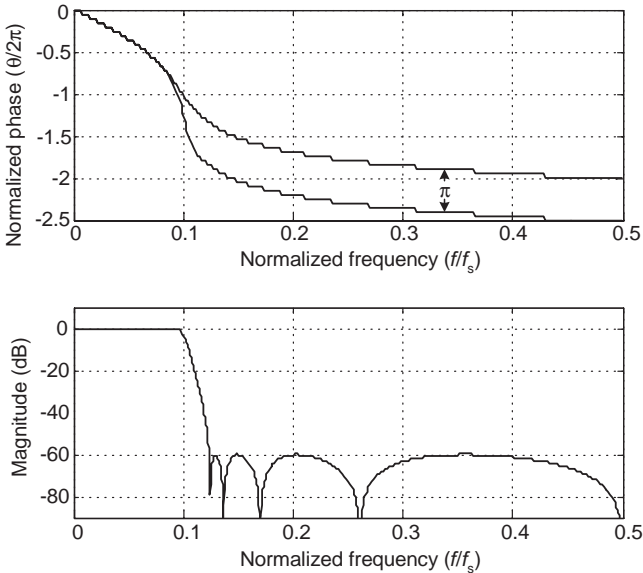
**Figure 9-16**    Two-path halfband filter: phase response of two paths; and frequency response of filter frequency transformed to 0.1 normalized bandwidth.

zeros have migrated around the unit circle to form the reduced-bandwidth version of the prototype.

Figure 9-16 presents the phase response of the two paths and the frequency response obtained by applying the lowpass-to-lowpass frequency transformation to the prototype two-path, four-multiply, halfband filter presented in Figure 9-7. The lowpass-to-lowpass transformation moved passband-edge from normalized frequency 0.25 to normalized frequency 0.1.

## 9.8   LOWPASS-TO-BANDPASS TRANSFORMATION

In the previous section, we examined the design of two-path, arbitrary-bandwidth lowpass filters formed from recursive all-pass first- and second-order filters as shown in Figure 9-14. We formed this filter by a transformation of a prototype halfband filter. We now address the second transformation, one that performs the lowpass-to-bandpass transformation. As in the previous section we invoke a frequency transformation wherein an existing all-pass subnetwork in a filter is replaced by another all-pass subnetwork. In particular, we now examine the transformation shown in (9–12).

$$\frac{1}{Z} \Rightarrow -\frac{1}{Z}\frac{1-cZ}{Z-c}; \quad c = \cos(\theta_C); \quad \theta_C = 2\pi\frac{f_C}{f_S} \tag{9–12}$$

This, except for the sign, is a cascade of a delay element with the generalized delay element we introduced in the initial discussion of first-order all-pass networks.

We can physically replace each delay in the prototype filter with this all-pass network and then tune the center frequency of the low-pass prototype by adjusting the parameter $c$. For our purposes, we perform the substitution algebraically in the all-pass filters comprising the two-path predistorted arbitrary-bandwidth filter, and in doing so generate yet a third structure for which we will develop and present an appropriate architecture. We substitute (9–12) into the second-order all-pass filter derived in (9–11) and rewritten in (9–13).

$$F(Z) = G(Z)|_{Z \Rightarrow \frac{Z(Z-c)}{(cZ-1)}}$$

$$= \frac{(b^2 + \alpha) + 2b(1+\alpha)Z + (1+\alpha b^2)Z^2}{(1+\alpha b^2) + 2b(1+\alpha)Z + (b^2 + \alpha)Z^2}\bigg|_{Z \Rightarrow \frac{Z(Z-c)}{(cZ-1)}} \tag{9–13}$$

After performing the indicated substitution and gathering up terms, we find the form of the transformed transfer function is as shown in (9–14).

$$F(Z) = \frac{1 + d_1 Z + d_2 Z^2 + d_3 Z^3 + d_4 Z^4}{Z^4 + d_1 Z^3 + d_2 Z^2 + d_3 Z + d_4}$$

$$d_1 = \frac{-2c(1+b)(1+\alpha b)}{1+\alpha b^2} \quad d_2 = \frac{(1+\alpha)(c^2(1+b)^2 + 2b)}{1+\alpha b^2} \tag{9–14}$$

$$d_3 = \frac{-2c(1+b)(1+\alpha b)}{1+\alpha b^2} \quad d_4 = \frac{\alpha + b^2}{1+\alpha b^2}$$

As expected, when we let $c \to 0$, $d_1$ and $d_3 \to 0$, while $d_2 \to c_1$ and $d_4 \to c_2$, the weights default to those of the prototype (arbitrary-bandwidth) filter. The transformation from lowpass to bandpass generates two spectral copies of the original spectrum, one each at the positive- and negative-tuned center frequency. The architecture of the transformed filter, which permits one multiplier to simultaneously form the matching numerator and denominator coefficients, is shown in Figure 9-17. Also shown is a processing block F(Z), which uses four coefficients $d_1$, $d_2$, $d_3$, and $d_4$. This is seen to be an extension of the two-multiply structure presented in Figure 9-14.

We have just described the lowpass-to-bandpass transformation that is applied to the second-order all-pass networks of the two-path filter. One additional transformation that requires attention is the lowpass-to-bandpass transformation that must be applied to the generalized delay or bandwidth-transformed delay from the prototype halfband filter. We substitute (9–12) into the first-order all-pass filter derived in (9–9) and rewritten in (9–15).

$$E(Z) = \frac{1 + bZ}{Z + b}\bigg|_{Z \Rightarrow \frac{Z(Z-c)}{(cZ-1)}}$$

$$= \frac{(cZ-1) + bZ(Z-c)}{Z(Z-c) + b(cZ-1)} = \frac{-1 + c(1-b)Z + bZ^2}{Z^2 - c(1-b)Z - b} \tag{9–15}$$

As expected, when $c \to 1$, the denominator goes to $(Z+b)(Z-1)$ while the numerator goes to $(1+bZ)(Z-1)$ so that the transformed all-pass filter reverts back to the
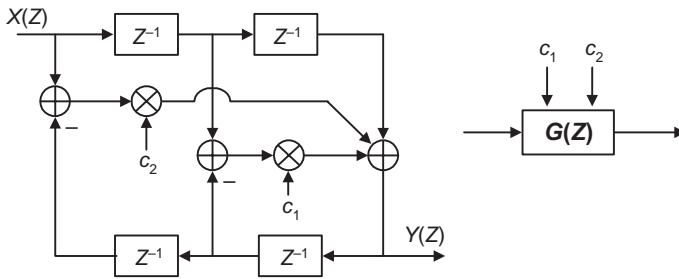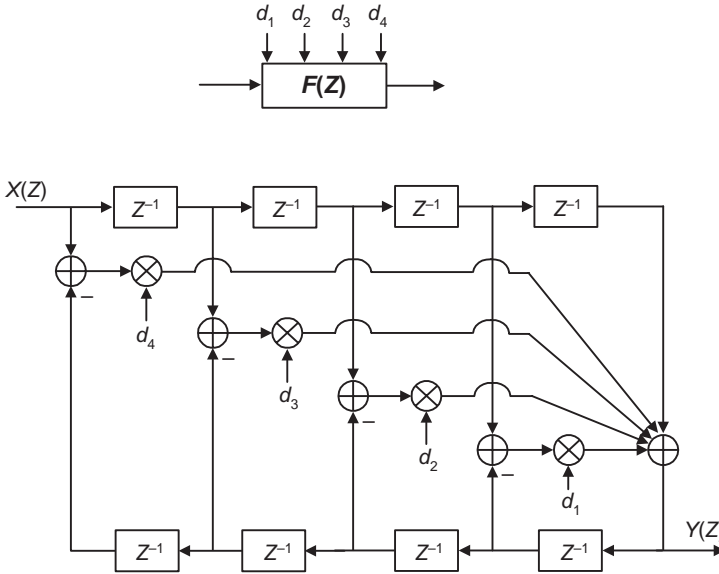
**Figure 9-17** Block diagram of general fourth-order all-pass filter.
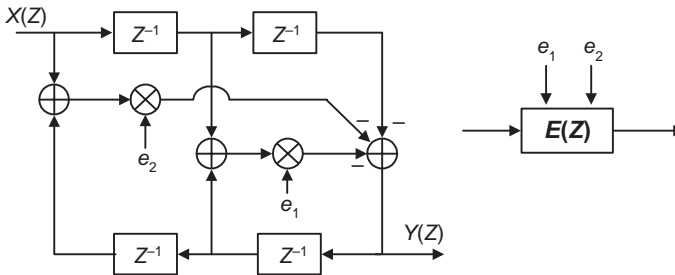


**Figure 9-18** Block diagram of lowpass-to-bandpass-transformation applied to lowpass-to-lowpass transformed delay element.

original first-order filter. The distributed minus sign in the numerator modifies the architecture of the transformed second-order filter by shuffling signs in Figure 9-13 to form the filter shown in Figure 9-18. Also shown is a processing block $E(Z)$, which uses two coefficients $e_1$ and $e_2$.

In the process of transforming the lowpass filter to a bandpass filter we convert the two-multiply second-order all-pass filter to a four-multiply fourth-order all-pass filter, and convert the one-multiply lowpass-to-lowpass filter to a two-multiply all-pass filter. The doubling of the number of multiplies is the consequence of replicating the spectral response at two spectral centers of the real bandpass system. Note that the 9-pole, 9-zero arbitrary lowpass filter now requires 18 multipliers to form the 18 poles and 18 zeros for the bandpass version of the two-path network. This is still significantly less than the standard cascade of first- and second-order canonic filters
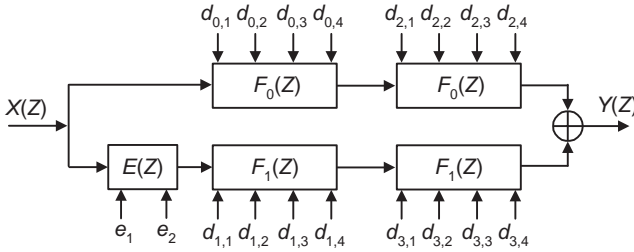
**Figure 9-19**    Effect on architecture of lowpass-to-bandpass frequency transformation applied to two-path arbitrary-bandwidth all-pass filter.
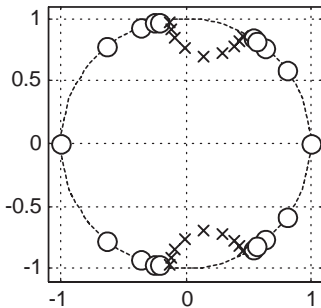


**Figure 9-20**    Pole-zero plot of two-path all-pass halfband filter subjected to lowpass-to-lowpass and then lowpass-to-bandpass transformations.

for which the same 18-pole, 18-zero filter would require 45 multipliers. Figure 9-19 shows how the structure of the prototype is affected by applying the lowpass-to-bandpass frequency transformation.

Figure 9-20 presents the pole-zero diagram of the frequency-transformed prototype filter. The nine poles defining the lowpass filter have been pulled to the neighborhood of the bandpass center frequency. The nine zeros have also replicated, appearing both below and above the passband frequency.

Figure 9-21 presents the phase response of the two paths and the frequency response obtained by applying the lowpass-to-bandpass frequency transformation to the prototype two-path, nine-multiply, lowpass filter presented in Figure 9-14. The one-sided bandwidth was originally adjusted to a normalized frequency of 0.1, and is now translated to a center frequency of 0.22.

## 9.9  CONCLUSIONS

We have presented a class of particularly efficient recursive filters based on two-path recursive all-pass filters. The numerator and denominator of an all-pass filter have reciprocal polynomials with the coefficient sequence of the denominator reversed in the numerator. The all-pass filters described in this chapter fold and align the numerator registers with the denominator registers so that the common coefficients
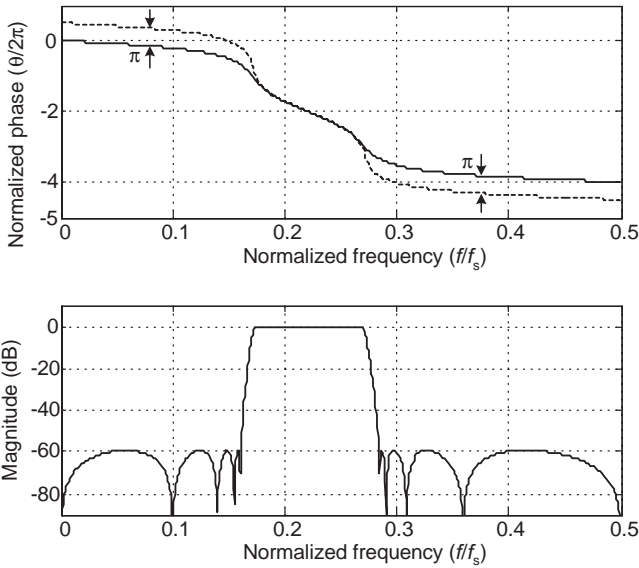
**Figure 9-21**    Frequency response of two-path all-pass filter subjected to lowpass-to-lowpass and then lowpass-to-bandpass transformations.

can be shared and thus form a pole and reciprocal zero with a single multiply. Coefficients are selected via a design algorithm to obtain matching phase profiles in specific spectral intervals with 180-degree phase offsets in other spectral intervals. When the time series from the two paths are added, the signals residing in the spectral intervals with 180-degree phase offsets are destructively canceled.

From the transfer-function perspective, the nonminimum phase zeros in the separate numerators migrate to the unit circle as a result of the interaction of the numerator-denominator cross-products resulting when forming the common denominator from the separate transfer functions of the two paths. The migration to the unit circle of the essentially free reciprocal zeros, formed while building the system poles, is the reason this class of filters requires less than half the multiplies of a standard recursive filter. The destructive cancellation of the spectral regions defined for the two-path halfband filter is preserved when the filter is subjected to transformations that enable arbitrary-bandwidth and arbitrary-center frequencies. The one characteristic not preserved under the transformation is the ability to embed 1-to-2 or 2-to-1 multirate processing in the two-path filter. The extension of the two-path filter structure to an *M*-path structure with similar computational efficiency is the topic of a second presentation. (The abovementioned MATLAB routines may be obtained by request to fharris@kahuna.sdsu.edu.)

## 9.10  REFERENCE

[1]  F. Harris, *Multirate Signal Processing for Communication Systems*. Prentice-Hall, Englewood Cliff, NJ, 2004.

## EDITOR COMMENTS

To facilitate the reader's software modeling, below we provide the polynomial coefficients for (9–4). Expanding its denominator, we may write (9–4) as:

$$H(Z) = \frac{b_0 Z^9 \pm b_1 Z^8 + b_2 Z^7 \pm b_3 Z^6 + b_4 Z^5 \pm b_4 Z^4 + b_3 Z^3 \pm b_2 Z^2 + b_1 Z^1 \pm b_0}{Z^9 + a_2 Z^7 + a_4 Z^5 + a_6 Z^3 + a_8 Z^1}. \qquad (9\text{-}16)$$

The individual lowpass and highpass paths are then described by

$$H_{\text{Lowpass}}(Z) = \frac{b_0 Z^9 + b_1 Z^8 + b_2 Z^7 + b_3 Z^6 + b_4 Z^5 + b_4 Z^4 + b_3 Z^3 + b_2 Z^2 + b_1 Z^1 + b_0}{Z^9 + a_2 Z^7 + a_4 Z^5 + a_6 Z^3 + a_8 Z^1}$$

$$(9\text{-}16')$$

and

$$H_{\text{Highpass}}(Z) = \frac{b_0 Z^9 - b_1 Z^8 + b_2 Z^7 - b_3 Z^6 + b_4 Z^5 - b_4 Z^4 + b_3 Z^3 - b_2 Z^2 + b_1 Z^1 - b_0}{Z^9 + a_2 Z^7 + a_4 Z^5 + a_6 Z^3 + a_8 Z^1}.$$

$$(9\text{-}16'')$$

The $b_k$ and $a_i$ coefficients in (9–16) are:

$$b_0 = \alpha_0 \alpha_2,$$
$$b_1 = \alpha_1 \alpha_3,$$
$$b_2 = \alpha_0 + \alpha_2 + \alpha_0 \alpha_1 \alpha_2 + \alpha_0 \alpha_2 \alpha_3,$$
$$b_3 = \alpha_1 + \alpha_3 + \alpha_0 \alpha_1 \alpha_3 + \alpha_1 \alpha_2 \alpha_3,$$
$$b_4 = \alpha_0 \alpha_1 + \alpha_0 \alpha_3 + \alpha_1 \alpha_2 + \alpha_2 \alpha_3 + \alpha_0 \alpha_1 \alpha_2 \alpha_3 + 1,$$
$$a_2 = \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3,$$
$$a_4 = \alpha_0 \alpha_1 + \alpha_0 \alpha_2 + \alpha_0 \alpha_3 + \alpha_1 \alpha_2 + \alpha_1 \alpha_3 + \alpha_2 \alpha_3,$$
$$a_6 = \alpha_0 \alpha_1 \alpha_2 + \alpha_0 \alpha_1 \alpha_3 + \alpha_0 \alpha_2 \alpha_3 + \alpha_1 \alpha_2 \alpha_3,$$
$$a_8 = \alpha_0 \alpha_1 \alpha_2 \alpha_3.$$

# Part Two

# Signal and Spectrum Analysis Tricks

# Chapter 10

# Fast, Accurate Frequency Estimators

**Eric Jacobsen**

*Abineau Communications*

**Peter Kootsookos**

*UTC Fire & Security Co.*

**T**he problem of estimating the frequency of a tone, contaminated with noise, appears in communications, audio, medical, instrumentation, and a host of other applications. Naturally, the fundamental tool for such analysis is the discrete Fourier transform (DFT) or its efficient cousin the fast Fourier transform (FFT) [1]. A well-known trade-off exists between the amount of time needed to collect data, the number of data points collected, the type of time-domain window used, and the resolution that can be achieved in the frequency domain. This chapter presents computationally simple estimators that provide substantial refinement of the frequency estimation of tones based on discrete Fourier transform DFT samples without the need for increasing the DFT size.

## 10.1 SPECTRAL PEAK LOCATION ALGORITHMS

An important distinction between the "resolution" in the frequency domain and the accuracy of frequency estimation should be clarified. Typically when the term *resolution* is used in the context of frequency estimation the intent is to describe the 3 dB width of the $\sin(x)/x$ response in the frequency domain. The resolution is affected by $N$, the number of data points collected; the type of window used; and the sample rate. One of the primary uses of the *resolution metric* is the ability to resolve closely spaced tones within an estimate. The frequency estimation problem, while it can be

affected by the resolution, seeks only to find, as accurately as possible, the location of the peak value of the $\sin(x)/x$ spectral envelope, regardless of its width or other characteristics. This distinction is important since improving the resolution is often a common avenue taken by someone wishing to improve the frequency estimation capability of a system. Improving the resolution is typically costly in computational burden or latency and this burden is often unnecessary if the only goal is to improve the frequency estimate of an isolated tone.

The general concept in using spectral peak location estimators is to estimate the frequency of the spectral peak, $k_{\text{peak}}$ in Figure 10-1, based on the three $X_{k-1}$, $X_k$, and $X_{k+1}$ DFT samples. If we estimated $k_{\text{peak}}$ to be equal to the $k$ index of the largest DFT magnitude sample, the maximum error in $k_{\text{peak}}$ would be half the width of the DFT bin. Using the frequency-domain peak sample, $X_k$, and one or two adjacent samples allows some simple best- or approximate-fit estimators to be used to improve the estimate of the peak location. In this material, each estimator provides a fractional correction term, $\delta$, which is added to the integer peak index, $k$, to determine a *fine* $k_{\text{peak}}$ estimate of the $\sin(x)/x$ main lobe peak location using

$$k_{\text{peak}} = k + \delta \qquad\qquad (10\text{--}1)$$
$$f_{\text{tone}} = k_{\text{peak}} f_{\text{s}} / N \qquad\qquad (10\text{--}1')$$

where $f_{\text{s}}$ is the time data sample rate in Hz and $N$ is the DFT size. Note that $\delta$ can be positive or negative, and that $k_{\text{peak}}$ need not be an integer. This refinement of the original bin-location estimate can be surprisingly accurate even in low signal-to-noise-ratio (SNR) conditions.

Figure 10-1 shows the basic concept where the main lobe of the $\sin(x)/x$ response in the frequency domain traverses three samples. These three samples can be used with simple curve-fit techniques to provide an estimate of the peak location between bins, $\delta$, which can then be added to the peak index, $k$, to provide a fine estimate of the tone frequency. An example of such an estimator is

$$\delta = (|X_{k+1}| - |X_{k-1}|)/(4|X_k| - 2|X_{k-1}| - 2|X_{k+1}|) \qquad\qquad (10\text{--}2)$$
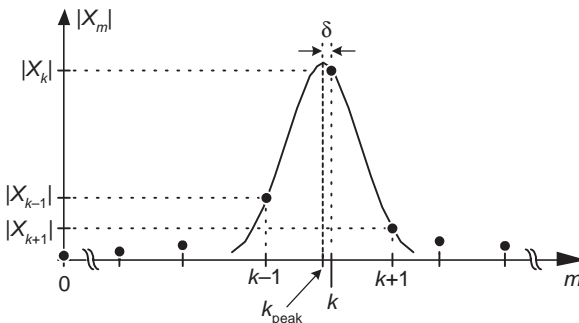


**Figure 10-1**    DFT magnitude samples of a spectral tone.

using three DFT magnitude samples [2], [3]. This estimator is simple, but it is statistically biased, and performs poorly in the presence of noise.

Jacobsen suggested some simple changes that improve the performance of estimator (10–2) [4]. For example, (10–3) uses the complex DFT values of $X_k$, $X_{k+1}$, and $X_{k-1}$. Taking the real part of the computed result provides significantly improved spectral peak location estimation accuracy and eliminates the statistical bias of (10–2).

$$\delta = -\operatorname{Re}[(X_{k+1} - X_{k-1})/(2X_k - X_{k-1} - X_{k+1})] \qquad (10\text{–}3)$$

Because the magnitude calculations in (10–2) are nontrivial, (10–3) provides a potential for computation reduction as well.

Quinn contributed two efficient estimators with good performance [5], [6]. Quinn's first estimator is:

$$\alpha_1 = \operatorname{Re}(X_{k-1}/X_k) \qquad (10\text{–}4)$$
$$\alpha_2 = \operatorname{Re}(X_{k+1}/X_k) \qquad (10\text{–}5)$$
$$\delta_1 = \alpha_1/(1-\alpha_1) \qquad (10\text{–}6)$$
$$\delta_2 = -\alpha_2/(1-\alpha_2) \qquad (10\text{–}7)$$

If $\delta_1 > 0$ and $\delta_2 > 0$, then $\delta = \delta_2$, otherwise $\delta = \delta_1$.

Quinn's second estimator [6] performs better but includes a number of transcendental function computations that take it out of the realm of computationally efficient estimators.

An estimator provided by MacLeod [7] requires the computation of a square root but is otherwise fairly simple. That estimator begins with

$$r = X_k \qquad (10\text{–}8)$$
$$R_n = \operatorname{Re}(X_n r^*) \qquad (10\text{–}9)$$

That is, create a real-valued vector $R_n$ that is the real part of the result of the Fourier coefficient vector $X_n$ made up of coefficients $X_{k-1}$ through $X_{k+1}$, times the conjugate of the spectral peak coefficient, $X_k$. This phase-aligns the terms relative to the peak so that the result gets a strong contribution in the real part from each of the coefficients. Then

$$d = (R_{k-1} - R_{k+1})/(2 * R_k + R_{k-1} + R_{k+1}) \qquad (10\text{–}10)$$
$$\delta = (\operatorname{sqrt}(1 + 8d^2) - 1)/4d \qquad (10\text{–}11)$$

While the above estimators work well for analysis when a rectangular time-domain window is applied to the DFT input samples, it is often beneficial or necessary to use nonrectangular windowing. Estimators that allow nonrectangular time-domain windows include those of Grandke [8], Hawkes [9], and Lyons [10].

Grandke proposed a simple estimator assuming the use of a Hanning window, using only the peak sample, $X_k$, and its largest-magnitude adjacent sample. Letting $X_k$ be the peak sample, the estimator is then simply:

**Table 10-1**    Correction Scaling Values for Four Common
Window Types for the (10–14) and (10–15) Estimators

| Window | $P$ | $Q$ |
| --- | --- | --- |
| Hamming | 1.22 | 0.60 |
| Hanning | 1.36 | 0.55 |
| Blackman | 1.75 | 0.55 |
| Blackman-Harris (3-term) | 1.72 | 0.56 |

When $|X_{k-1}| > |X_{k+1}|$,

$$\alpha = |X_k|/|X_{k-1}| \tag{10–12}$$

$$\delta = (\alpha - 2)/(\alpha + 1). \tag{10–12'}$$

When $|X_{k-1}| < |X_{k+1}|$,

$$\alpha = |X_{k+1}|/|X_k| \tag{10–13}$$

$$\delta = (2\alpha - 1)/(\alpha + 1) \tag{10–13'}$$

Hawkes proposed the estimator (10–14), similar to (10–2) with a scaling term, $P$, which can be adjusted for different window applications.

$$\delta = P(|X_{k+1}| - |X_{k-1}|)/(|X_k| + |X_{k-1}| + |X_{k+1}|) \tag{10–14}$$

Inspired by (10–3) and (10–14), Lyons has suggested estimator (10–15) with a window-specific scaling term $Q$ [10].

$$\delta = \text{Re}[Q(X_{k-1} - X_{k+1})/(2X_k + X_{k-1} + X_{k+1})] \tag{10–15}$$

Table 10-1 shows the scaling values for the (10–14) and (10–15) estimators for certain common windowing functions.

## 10.2   ALGORITHM PERFORMANCE

Simulation of the above estimators yields some interesting results. While the estimators differ in computational efficiency, their performance also varies and some performance metrics vary with the tone offset within the bin. Thus the menu of estimators available that would perform best in a certain application depends on the nature of the system. In general the estimators used with nonrectangular windows provide less accuracy and have more highly biased outputs than those used with rectangular windows [8], [9], [10], but their performance is better than that with the rectangular-windowed estimators when applied to DFT samples from nonrectangular-windowed data.

The performance of the best estimators is captured in Figures 10-2 and 10-3. A tone was simulated with peak location varying from bin 9 to bin 10 at small increments. The root-mean-squared error (RMSE) of the estimators for tones at bin 9.0

Estimator RMS error vs. SNR at $f = 9.00$

(a)

Estimator RMS error vs. SNR at $f = 9.50$

(b)
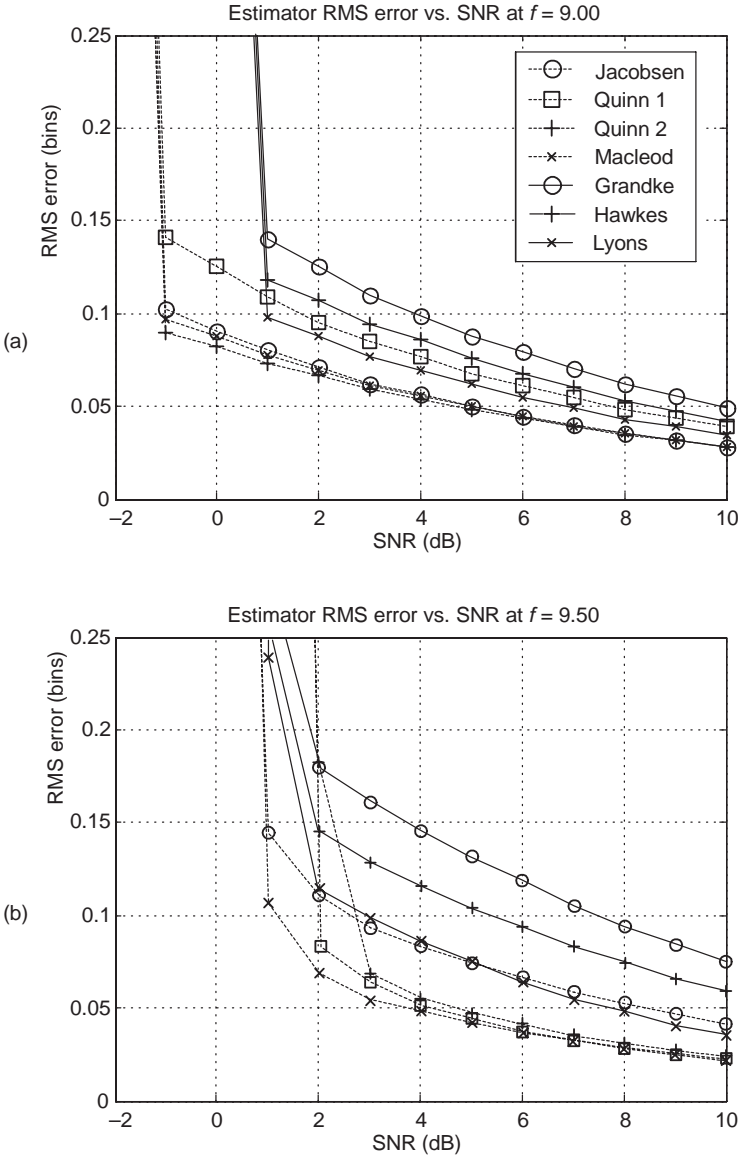
**Figure 10-2**  RMS error performance in AWGN for the indicated estimators: (a) tone at bin 9; (b) tone centered between bin 9 and bin 10.

and bin 9.5 are shown in Figure 10-2. Some thresholding of the FFT peak location detection is seen in Figure 10-2(b) at low SNRs, which is independent of the fine estimator used. The estimators of Grandke, Hawkes, and Lyons were tested with a Hanning window applied while the remainder were tested with rectangular windowing.
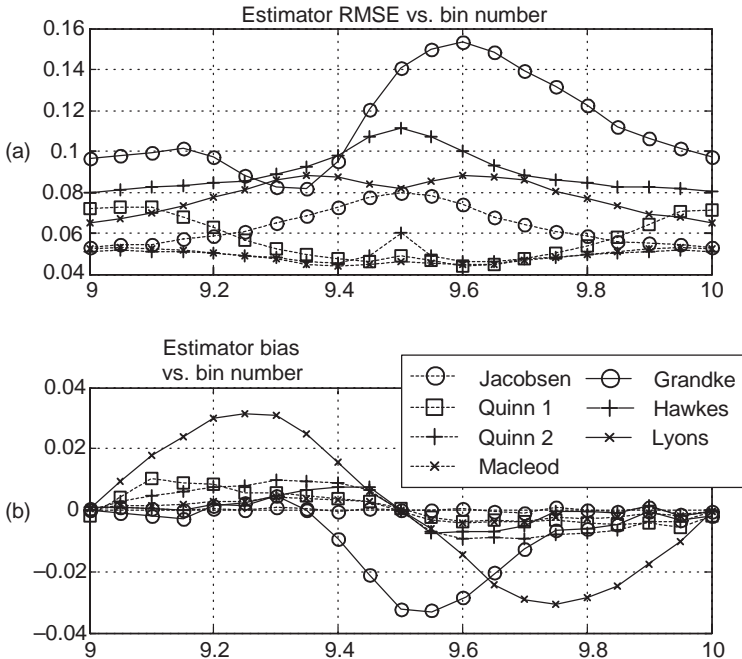
**Figure 10-3**    Interbin performance in AWGN with SNR = 1.4 dB: (a) RMSE; (b) bias.

It can be seen that the relative locations of some of the estimator's performance curves change depending on the tone offset within the bin. This is further seen in Figure 10-3, where the RMSE and bias at ≈1.4 dB SNR demonstrates that for some estimators the error is worse when the tone is aligned on a bin and for others it is worse when the tone is between bins. In Figure 10-4 the estimator performances are shown in no noise. All of the estimators shown are unbiased except for Grandke (10–12) and (10–13), Hawkes (10–14), and Lyons (10–15).

The relative merits of the various estimators are captured in Table 10-2. All of the estimators use a number of adds or subtracts, but these are typically trivial to implement in logic or software and so are not considered significant for computational complexity comparison. The relative complexity of multiplies, divides, magnitude computations, or transcendental functions via computation or lookup table depends highly on the specific implementation and they are enumerated here for consideration.

## 10.3   CONCLUSIONS

Interbin tone peak location estimation for isolated peaks can be performed with a number of effective estimators in the presence or absence of noise or nonrectangular windowing. Which estimator is most appropriate for a particular application depends
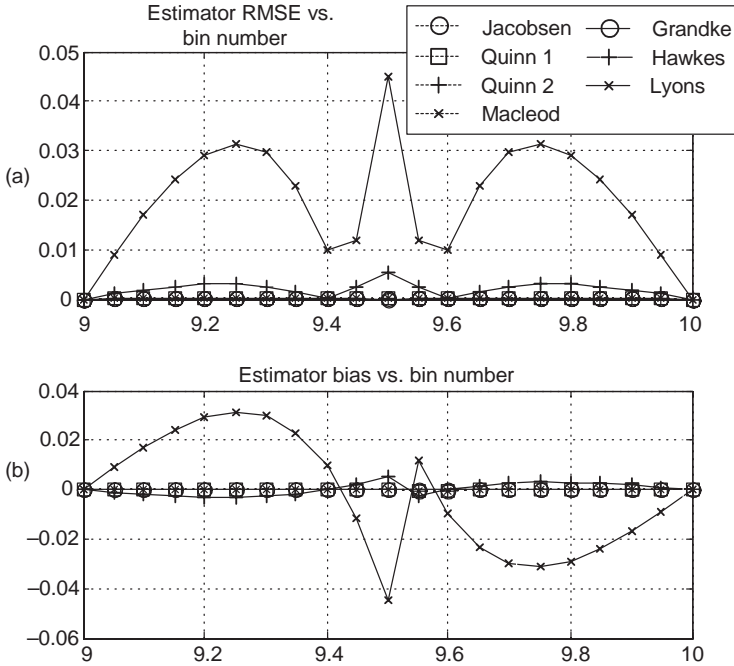
**Figure 10-4**    Interbin performance in no noise: (a) RMSE; (b) bias.

**Table 10-2**    Algorithm Computational and Performance Comparison

| Equ. | Mult. | Div. | Mag. | RMSE | Bias? | Remarks |
|---|---|---|---|---|---|---|
| (10–2) | 0 | 1 | 3 | High at low SNR | Yes | Not considered due to poor performance |
| (10–3) | 4 | 1 | 0 | Medium, increasing with bin offset | No | Very good bias performance |
| (10–4), (10–5) | 4 | 0 | 0 | Medium, decreasing with bin offset | No | |
| n/a | 8 | 7 | 0 | Low | No | Quinn's second estimator [6].* |
| n/a | 1(3) | 2 | 0 | Very low | No | MacLeod's estimator [7].* * |
| (10–12), (10–13) | 0 | 2 | 2 | High, increasing with bin offset | No | Worst RMSE, Hanning only |
| (10–14) | 1 | 1 | 3 | High | Yes | Good for windowed applications |
| (10–15) | 5 | 1 | 0 | High | Yes | Good for windowed applications |

*Requires four logarithmic computations.
**Requires one square root computation.

on the type of windowing used, system sensitivity to bias, location-dependent RMS error, and computational complexity. Macleod's and Quinn's second estimator perform especially well but may be computationally burdensome in complexity-sensitive applications. The Jacobsen estimator provides a good complexity–performance trade-off, requiring only a single divide but at the expense of a small performance reduction, especially for tones between bins. Lyons' estimator is similar for windowed applications, requiring only an additional multiplication compared to Jacobsen's but at the expense of some bias. Since estimator bias is predictable it often can be removed simply with an additional arithmetic operation.

## 10.4 REFERENCES

[1] D. RIFE and R. BOORSTYN, "Single-Tone Parameter Estimation from Discrete-Time Observations," *IEEE Trans. Information Theory*, vol. IT-20, September 1974, pp. 591–598.

[2] W. PRESS, et al., *Numerical Recipes in C*. Cambridge University Press, Cambridge, 1992, Chapter 10.

[3] P. VOGLEWEDE, "Parabola Approximation for Peak Determination," *Global DSP Magazine*, May 2004.

[4] E. JACOBSEN, "On Local Interpolation of DFT Outputs," [Online: http://www.ericjacobsen.org/FTinterp.pdf.]

[5] B. G. QUINN, "Estimating Frequency by Interpolation Using Fourier Coefficients," *IEEE Trans. Signal Processing*, vol. 42, no. 5, May 1994, pp. 1264–1268.

[6] B. QUINN, "Estimation of Frequency, Amplitude and Phase from the DFT of a Time Series," *IEEE Trans. Signal Processing*, vol. 45, no. 3, March 1997, pp. 814–817.

[7] M. MACLEOD, "Fast Nearly ML Estimation of the Parameters of Real or Complex Single Tones or Resolved Multiple Tones," *IEEE Trans. Signal Processing*, vol. 46, no. 1, January 1998, pp. 141–148.

[8] T. GRANDKE, "Interpolation Algorithms for Discrete Fourier Transforms of Weighted Signals," *IEEE Trans. Instrumentation and Measurement*, vol. IM-32, pp. 350–355, June 1983.

[9] K. HAWKES, "Bin Interpolation," *Technically Speaking*. ESL Inc., January 1990, pp. 17–30.

[10] R. LYONS, *private communication*, August 30, 2006.

## EDITOR COMMENTS

Estimators (10–3) and (10–15) have the advantage that DFT magnitude calculations, with their computationally costly square root operations, are not required as is necessary with some other spectral peak location estimators described in this chapter. However, the question arises, "How do we determine the index $k$ of the largest-magnitude DFT sample, $|X_k|$, in Figure 10-1 without computing square roots to obtain DFT magnitudes?" The answer is we can use the complex sample magnitude estimation algorithms, requiring no square root computations, described in Chapter 16.

The following shows the (10–3) and (10–15) estimators in real-only terms. If we express the complex spectral samples, whose magnitudes are shown in Figure 10-1, in rectangular form as:

$$X_{k-1} = X_{k-1,\text{real}} + jX_{k-1,\text{imag}}$$
$$X_k = X_{k,\text{real}} + jX_{k,\text{imag}}$$
$$X_{k+1} = X_{k+1,\text{real}} + jX_{k+1,\text{imag}},$$

we can express estimator (10–3) using real-only values and eliminate its minus sign, as

$$\delta_{(3)\text{ real-only}} = \frac{R_{\text{num}}R_{\text{den}} + I_{\text{num}}I_{\text{den}}}{R_{\text{den}}^2 + I_{\text{den}}^2} \tag{10–16}$$

where

$$R_{\text{num}} = X_{k-1,\text{real}} - X_{k+1,\text{real}}$$
$$I_{\text{num}} = X_{k-1,\text{imag}} - X_{k+1,\text{imag}}$$
$$R_{\text{den}} = 2X_{k,\text{real}} - X_{k-1,\text{real}} - X_{k+1,\text{real}}$$
$$I_{\text{den}} = 2X_{k,\text{imag}} - X_{k-1,\text{imag}} - X_{k+1,\text{imag}}.$$

Thus (10–16) is the actual real-valued arithmetic needed to compute (10–3). In a similar manner we can express estimator (10–15) using real-only values as

$$\delta_{(15)\text{ real-only}} = \frac{Q(R_{\text{num}}R_{\text{den}} + I_{\text{num}}I_{\text{den}})}{R_{\text{num}}^2 R_{\text{den}}^2}$$

where

$$R_{\text{num}} = X_{k-1,\text{real}} - X_{k+1,\text{real}}$$
$$I_{\text{num}} = X_{k-1,\text{imag}} - X_{k+1,\text{imag}}$$
$$R_{\text{den}} = 2X_{k,\text{real}} + X_{k-1,\text{real}} + X_{k+1,\text{real}}$$
$$I_{\text{den}} = 2X_{k,\text{imag}} + X_{k-1,\text{imag}} - X_{k+1,\text{imag}}.$$

# Chapter 11

# Fast Algorithms for Computing Similarity Measures in Signals

**James McNames**
*Portland State University*

**T**his chapter describes fast algorithms that compute *similarity measures* between contiguous segments of a discrete-time one-dimensional signal $x(n)$ and a single template or pattern $p(n)$, where $N_x$ is the duration of the observed signal $x(n)$ in units of samples and $N_p$ is the duration of the template $p(n)$ in units of samples. Similarity measures are used to detect occurrences of significant intermittent events that occur in the observed signal. The occurrences are detected by finding the sample times in which a segment of the signal is similar to the template. For example, Figure 11-1(a) shows the output sequence of a noisy microelectrode recording and Figure 11-1(b) shows the cross-correlation of that sequence with template matching. It is much easier to detect spikes as the peaks above the noise floor (the dots) in the cross-correlation than it is in the original recorded sequence.

This type of event detection is called *template matching*, and it is used in many signal and image processing applications. Some examples of similarity measure applications are QRS and arrhythmia detection in electrocardiograms [1], labeling of speech [2], object detection in radar [3], matching of patterns in text strings [4], and detection of action potentials in extracellular recordings of neuronal activity [5].

Direct calculation of similarity measures normally requires computation that scales (the number of computations increases) linearly with the duration of the
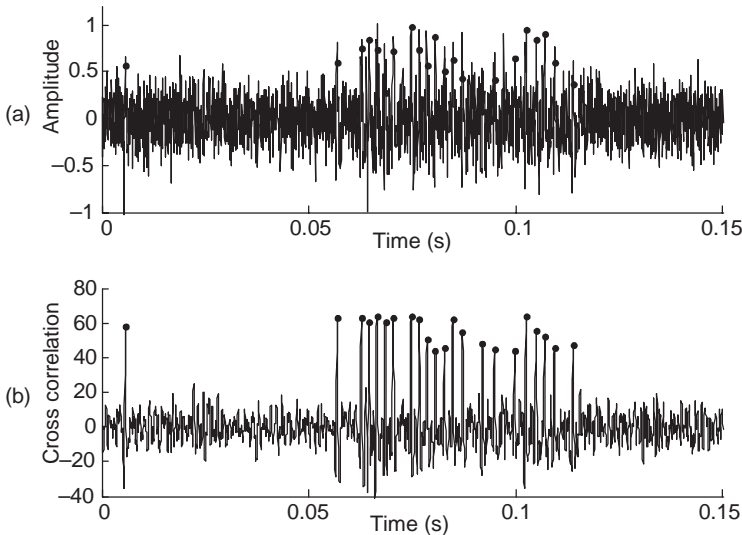
**Figure 11-1**    Microelectrode recording: (a) original sequence; (b) cross-correlation result.

template, $N_p$. Although fast algorithms for computing similarity measures in signals have been discovered independently by many signal processing engineers, there is no publication that describes these algorithms in a common framework. Most of the similarity measures used in practice are based on second-order statistics, which include averages, products, and squares of the terms. These statistics include sample means, cross-correlations, and squared differences. The aims of this chapter are to demonstrate how nonlinear similarity measures based on these second-order statistics can be computed efficiently using simple components and to estimate the template size $N_p$ at which the fast algorithms outperform direct implementations.

Although no one has published fast algorithms for one-dimensional signals, many publications describe fast algorithms for template matching in images. Image template matching differs fundamentally from signal template matching in several important respects. The size of the template, in units of pixels, usually representing an object that one wishes to locate within the image, is often on the same scale as the size of the image, which limits the efficiency of fast filtering algorithms based on block processing that can be used in one-dimensional signals. In image processing applications the template is often not just shifted (in the $x$–$y$ directions), but rotated, scaled in size, and scaled in amplitude. Rotation has no analogous operation in one-dimensional signal processing. In both image and signal processing applications, the detection of templates with various scales or amplitudes is usually treated by selecting a similarity measure that is invariant to changes in scale.

Most of the similarity measures of interest in template matching applications are nonlinear and based on second-order statistics of the template and the signal segment. Direct implementations of these measures requires computation that scales linearly with the duration of the template. If the signal length $N_x$ is much greater than the template length ($N_x \gg N_p$), the similarity measure must be calculated over the entire possible range ($0 \leq n \leq N_x - N_p$), and the similarity measures can be expressed in terms of second-order components, then it is possible to calculate these measures much more quickly with running sums and fast finite impulse response (FIR) filtering algorithms than it would be with direct calculation.

## 11.1 SIMILARITY MEASURE COMPONENTS

In order to describe our similarity measure algorithms in a common framework, Table 11-1 lists the *components* that can be computed to evaluate various similarity measures. Later we will define these components and present algorithms for computing these components in the most efficient way possible. The range of each sum in this table is $i = 0$ to $N_p$. The weighting function $w(i)^2$ permits the user to control the relative influence of each term in the template.

## 11.2 SIMILARITY MEASURES

Four similarity measures are discussed in the following sections. Each section defines the measure, describes its relevant properties, cites some example applications, and gives a mathematically equivalent expression in terms of the components listed in Table 11-1. Generalizations to other measures based on second-order statistics, such as normalized cross-correlation, cross-covariance, and normalized mean squared error, are straightforward.

**Table 11.1**  Similarity Measure Components

| | |
|---|---|
| $s_x(n) = \sum_i x(n+i)$ | $s_{x^2}(n) = \sum_i x(n+i)^2$ |
| $s_{x^2 w^2}(n) = \sum_i x(n+i)^2 w(i)^2$ | $s_{xp}(n) = \sum_i x(n+i) p(i)$ |
| $s_{xpw^2}(n) = \sum_i x(n+i) p(i) w(i)^2$ | $s_p(n) = \sum_i p(i)$ |
| $s_{p^2}(n) = \sum_i p(i)^2$ | $s_{p^2 w^2}(n) = \sum_i p(i)^2 w(i)^2$ |

## 11.2.1   Cross-Correlation

The sample cross-correlation of the template and the signal is defined as

$$r(n) = \frac{1}{N_p} \sum_{i=0}^{N_p-1} x(n+i)p(i) \tag{11-1}$$

The primary advantage of the cross-correlation is that it can be computed efficiently [6]. It is widely used for matched filter and eigenfilter applications [7].

Cross-correlation is the most popular similarity measure in signal processing applications because it can be calculated with traditional linear filter architectures by treating the template as the impulse response of an anti-causal tapped-delay line FIR filter. The cross-correlation can be expressed in terms of the Table 11-1 second-order components as

$$r(n) = \frac{1}{N_p} s_{xp}(n) \tag{11-2}$$

## 11.2.2   Mean Squared Error

The mean squared error is defined as

$$\zeta(n) = \frac{1}{N_p} \sum_{i=0}^{N_p-1} [x(n+i) - p(i)]^2 \tag{11-3}$$

This similarity measure is popular for many applications because it has several useful properties. It is invariant to identical shifts in the amplitude of the signal and template means, is related to the maximum likelihood estimate of the template location, and maximizes the peak signal-to-noise ratio [8]–[12].

The mean squared error can be expressed in terms of the Table 11-1 second-order components as

$$\zeta(n) = \frac{1}{N_p} \sum_{i=0}^{N_p-1} x(n+i)^2 - 2x(n+i)p(i) + p(i)^2 \tag{11-4}$$

$$\zeta(n) = \frac{1}{N_p}[s_{x^2}(n) - 2s_{xp}(n) + s_{p^2}] \tag{11-5}$$

The same decomposition in terms of the components in Table 11-1 has been used in image processing applications for both analysis and fast computation [6], [9], [12].

Some signal analysts prefer to use the *root mean squared error* (RMSE), the square root of the above mean squared error (MSE), as a similarity measure. In most applications the MSE conveys the same similarity information as the RMSE, but the

MSE does not require the computationally costly square root operation demanded by RMSE calculations.

## 11.2.3   Weighted Mean Squared Error

The mean squared error in the previous section can be generalized to facilitate relative weighting of each term in the sum,

$$\zeta_w(n) = \frac{1}{N_p} \sum_{i=0}^{N_p-1} (w(i)[x(n+i) - p(i)])^2 \tag{11-6}$$

where $w$ is a vector of weights. This similarity measure may be appropriate if the occurrence of the template in the signal also affects the signal variance and for detecting patterns that vary. Under some general assumptions, this similarity measure can also be used for maximum likelihood estimation of the template locations, as has been done for images [11]. The weighted mean squared error can be expressed as

$$\zeta_w(n) = \frac{1}{N_p} \sum_{i=0}^{N_p-1} [w(i)x(n+i) - w(i)p(i)]^2 \tag{11-7}$$

and then expanded in the same manner as (11–3),

$$\zeta_w(n) = s_{x^2 w^2}(n) - 2s_{xpw^2}(n) + s_{p^2 w^2} \tag{11-8}$$

## 11.2.4   Centered Correlation Coefficient

The centered correlation coefficient is defined as

$$\rho_C(n) = \frac{\displaystyle\sum_{i=0}^{N_p-1} [x(n+i) - \bar{x}(n)][p(i) - \bar{p}]}{\left(\displaystyle\sum_{i=0}^{N_p-1} [x(n+i) - \bar{x}(n)]^2\right)\left(\displaystyle\sum_{i=0}^{N_p-1} [p(i) - \bar{p}]^2\right)} \tag{11-9}$$

where $\bar{x}(n)$ and $\bar{p}$ are the sample means of the signal segment and template,

$$\bar{x}(n) = \frac{1}{N_p} \sum_{i=0}^{N_p-1} x(n+i) \qquad \bar{p} = \frac{1}{N_p} \sum_{i=0}^{N_p-1} p(i) \tag{11-10}$$

The first term in the denominator of (11–9) is the scaled sample variance of the signal segment, $(N_p - 1)\sigma_x^2(n)$. It can be expressed as

$$(N_p - 1)\sigma_x^2(n) = \sum_{i=0}^{N_p - 1} [x(n+i) - \bar{x}(n)]^2 \qquad (11\text{--}11)$$

$$= s_{x^2}(n) - \frac{1}{N_p} s_x(n)^2 \qquad (11\text{--}12)$$

In the image processing literature, this similarity measure is sometimes called the *normalized cross-correlation* [10], [13]. It is invariant to changes in the mean and scale of both the signal segment and template. It is also normalized and bounded, $-1 \le \rho_c(n) \le 1$. It can be expressed in terms of the Table 11-1 second-order components as

$$\rho_c(n) = \frac{s_{x\tilde{p}}(n)}{\sqrt{s_{x^2}(n) - \frac{1}{N_p} s_x(n)^2}\, s_{\tilde{p}^2}} \qquad (11\text{--}13)$$

where the centered template is defined as

$$\tilde{p}(i) = p(i) - \bar{p} \qquad (11\text{--}14)$$

and

$$s_{\tilde{p}^2} = \sum_{i=0}^{N_p - 1} \tilde{p}(i)^2 \qquad (11\text{--}15)$$

## 11.3  FAST CALCULATION OF SIMILARITY MEASURE COMPONENTS

Since the last three similarity measure components in Table 11-1, $s_p$, $s_{p^2}$, and $s_{p^2 w^2}$, do not scale with the signal length $N_x$, the computational cost of computing these components is insignificant compared to the first five components. The following sections describe fast techniques for computing the first five Table 11-1 components.

### 11.3.1  Running Sums

The components $s_x(n)$ and $s_{x^2}(n)$ are the same operation applied to $x(n)$ and $x(n)^2$, respectively. Because these are simple sums, they can be computed with a running sum,

$$s_x(n) = s_x(n-1) + x(n + N_p - 1) - x(n-1) \qquad (11\text{--}16)$$
$$s_{x^2}(n) = s_{x^2}(n-1) + x(n + N_p - 1)^2 - x(n-1)^2 \qquad (11\text{--}17)$$

where we define $x(n) = 0$ for $n < 0$. The computational cost of this operation is two additions per output sample.

Similar schemes have been used in image processing based on precomputed summed area and summed squares tables [12]–[14]. It is also possible to calculate these components with fast FIR filters, which compute the convolution as multiplica-

tion in the frequency domain using the FFT, by convolving $x(n)$ and $x(n)^2$ with an impulse response of ones [15], but this is less efficient than running sums.

A disadvantage of running sums is that the quantization errors accumulate in the same manner as a random walk, and the quantization error variance scales linearly with $N_x$. This effect can be reduced by computing the sum directly every $N_r$ samples. This increases the computational cost to $N_p + 2(N_r - 1)/N_r$ per sample. Compared with a direct computation of $s_x(n)$ and $s_{x^2}(n)$, the computational cost of a running sum is reduced by nearly a factor of $N_r$ as compared with a direct computation of the sum at every sample time.

## 11.3.2 Fast FIR Filters

The components $s_{x^2w^2}(n)$, $s_{xp}(n)$, and $s_{xpw^2}(n)$ defined in Table 11-1 can each be expressed as a convolution of a signal with a tapped-delay line FIR filter impulse response,

$$s_{x^2w^2}(n) = x(n)^2 * w(-n)^2 \qquad (11\text{–}18)$$
$$s_{xp}(n) = x(n) * p(-n) \qquad (11\text{–}19)$$
$$s_{xpw^2}(n) = x(n) * p(-n) w(-n)^2 \qquad (11\text{–}20)$$

where * denotes convolution. Each impulse response has a finite duration of $N_p$ and is anti-causal with a range of $-(N_p - 1) \le n \le 0$.

Expressing these components as a convolution with an FIR filter makes it possible to apply any of the fast filtering algorithms that have been developed over the past 40 years for a variety of hardware architectures. For large pattern lengths, say $N_p > 100$, the fastest algorithms take advantage of the well-known convolution property of the discrete Fourier transform (DFT) and the efficiency of the FFT to minimize computation.

## 11.4  RESULTS

## 11.4.1  Floating Point Operations

Table 11-2 lists the number of multiplication/division operations ($M$), addition/subtraction operations ($A$), and square root operations ($S$) per sample for direct implementations of each of the four similarity measures.

Table 11-3 lists the number of FIR filtering operations ($F$) and other arithmetic operations per sample for fast implementations of each of the four similarity measures.

These tables do not include the cost of computing $s_p$, $s_{p^2}$ and $s_{p^2w^2}$, since $N_x \gg N_p$ and the cost of computing these components does not scale with $N_x$. These tables assume three additions per sample for each running sum, which corresponds to a reset interval of approximately $N_r = N_p$. Note that these tables include only basic arithmetic operations. Other operations that are not accounted for here, such as

**Table 11.2**    Computational Cost of a Direct
Implementation of Each Similarity Measure

| Similarity measure | Equation | $M$ | $A$ | $S$ |
|---|---|---|---|---|
| $r(n)$ | (11–1) | $N_p+1$ | $N_p-1$ | — |
| $\xi(n)$ | (11–3) | $N_p+1$ | $2N_p-1$ | — |
| $\xi_w(n)$ | (11–6) | $2N_p$ | $2N_p-1$ | — |
| $\rho_c(n)$ | (11–9), (11–10) | $2N_p+3$ | $5N_p-3$ | 1 |

**Table 11.3**    Computational Cost of a Fast Implementation
of Each Similarity Measure

| Similarity measure | Equation | $F$ | $M$ | $A$ | $S$ |
|---|---|---|---|---|---|
| $r(n)$ | (11–2) | 1 | — | — | — |
| $\xi(n)$ | (11–5) | 1 | 3 | 5 | — |
| $\xi_w(n)$ | (11–8) | 2 | 2 | 2 | — |
| $\rho_c(n)$ | (11–13) | 1 | 5 | 7 | — |

memory access and conditional branching, can have a large impact on performance
[16]–[17].

## 11.4.2    Simulation Example

To demonstrate an example of the possible reduction in computational cost achiev-
able by using running sums and fast FIR filters, we compared the direct and fast
implementations of each similarity measure on a personal computer (Pentium 4,
2 GHz processor, 512 MB of RAM). The direct implementations and running sums
were written in C and compiled with the Visual C/C++ optimizing compiler
(Microsoft, version 12.00.8168) for 80×86. The fast FIR filtering was performed
by the overlap-add method implemented in the fftfilt() function in MATLAB
(MathWorks, version 7.0.0.19920). The block length was chosen automatically by
MATLAB to minimize the total number of floating-point operations (flops). The
source code used to generate these results is available at http://bsp.pdx.edu.

Figure 11-2 shows the average execution time required per sample for each of
the four measures. The signal length was $N_x=2^{18}=262,144$ samples. The execution
time for each measure and template length was averaged over 20 runs. The range
of execution times (max-min) in a set of 20 runs never exceeded 0.24 µs/sample. In
each case, the mean absolute error between the direct and fast implementations was
never greater than 5.4ε, where ε is the distance from a floating-point representation
of 1.0 to the next larger floating-point number. Thus, quantization did not signifi-
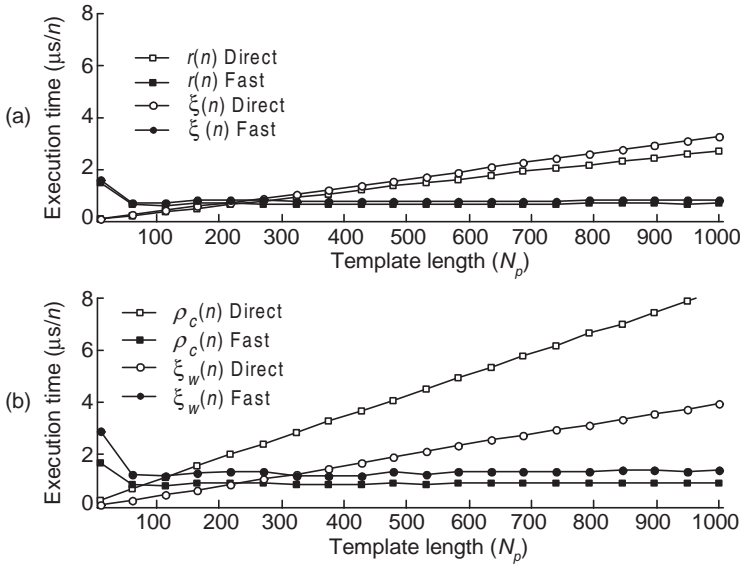
**Figure 11-2** Similarity measures' average execution times required per sample (µs/*n*).

cantly reduce the accuracy of the similarity measures computed with the fast implementations.

## 11.5 CONCLUSIONS

Popular similarity measures based on second-order statistics can be expressed in terms of basic components and computed with fast algorithms. These techniques are most suitable for detection applications in which the similarity measure must be calculated for every sample in a given signal segment [1]–[5]. For template lengths of greater than 200, these fast algorithms are much more computationally efficient than direct implementations.

## 11.6 REFERENCES

[1] A. BOLLMANN, K. SONNE, H. ESPERER, I. TOEPFFER, J. LANGBERG, and H. KLEIN, "Non-invasive Assessment of Fibrillatory Activity in Patients with Paroxysmal and Persistent Atrial Fibrillation Using the Holter ECG," *Cardiovascular Research*, vol. 44, 1999, pp. 60–66.

[2] S. HANNA and A. CONSTANTINIDES, "An Automatic Segmentation and Labelling Algorithm for Continuous Speech Signals," *Digital Signal Processing*, vol. 87, 1987, pp. 543–546.

[3] A. OLVER and L. CUTHBERT, "FMCW Radar for Hidden Object Detection," *IEE Proceedings—F Radar and Signal Processing*, vol. 135, no. 4, August 1988, pp. 354–361.

[4] M. ATALLAH, F. CHYZAK, and P. DUMAS, "A Randomized Algorithm for Approximate String Matching," *Algorithmica*, vol. 29, 2001, pp. 468–486.

[5] I. Bankman, K. Johnson, and W. Schneider, "Optimal Detection, Classification, and Superposition Resolution in Neural Waveform Recordings," *IEEE Transactions on Biomedical Engineering*, vol. 40, no. 8, August 1993, pp. 836–841.

[6] L. Brown, "A Survey of Image Registration Techniques," *ACM Computing Surveys*, vol. 24, no. 4, December 1992, pp. 325–376.

[7] E. Hall, R. Kruger, S. Dwyer, III, D. Hall, R. McLaren, and G. Lodwick, "A Survey of Prepocessing and Feature Extraction Techniques for Radiographic Images," *IEEE Transactions on Computers*, vol. 20, no. 9, September 1971, pp. 1032–1044.

[8] K. McGill and L. Dorfman, "High-Resolution Alignment of Sampled Waveforms," *IEEE Transactions on Biomedical Engineering*, vol. 31, no. 6, June 1984, pp. 462–468.

[9] W. Pratt, *Digital Image Processing*. 2nd Ed. John Wiley & Sons, Inc., 1991.

[10] R. Gonzalez and R. Woods, *Digital Image Processing*, Addison-Wesley, 1992.

[11] C. Olson, "Maximum-Likelihood Image Matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 6, June 2002, pp. 853–857.

[12] S. Kilthau, M. Drew, and T. Möller, "Full Search Content Independent Block Matching Based on the Fast Fourier Transform," *2002 International Conference on Image Processing*, vol. 1, 2002, pp. 669–672.

[13] K. Briechle and U. Hanebeck, "Template Matching Using Fast Normalized Cross Correlation," Proceedings of SPIE—The International Society for Optical Engineering, vol. 4387, 2001, pp. 95–102.

[14] H. Schweitzer, J. Bell, and F. Wu, "Very Fast Template Matching," *Computer Vision—ECCV 2002*, vol. 2353, 2002, pp. 358–372.

[15] M. Uenohara and T. Kanade, "Use of Fourier and Karhunen-Loeve Decomposition for Fast Pattern Matching with a Large Set of Templates," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 8, August 1997, pp. 891–898.

[16] Z. Mou and P. Duhamel, "Short-Length FIR Filters and Their Use in Fast Nonrecursive Filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 6, June 1991, pp. 1322–1332.

[17] A. Gacic, M. Püschel, and J. Moura, "Fast Automatic Software Implementations of FIR Filters," *International Conference on Acoustics, Speech and Signal Processing (ICASSP'03)*, vol. 2, April 2003, pp. 541–544.

# Chapter 12

# Efficient Multi-tone Detection

**Vladimir Vassilevsky**
*Abvolt Ltd.*

**T**his chapter presents the DSP tricks employed to build a computationally efficient multi-tone detection system implemented without multiplications, and with minimal data memory requirements. More specifically, we describe the detection of incoming dial tones, the validity checking to differentiate valid tones from noise signals, and the efficient implementation of the detection system. While our discussion focuses on dual-tone multifrequency (DTMF) telephone dial tone detection, the processing tricks presented may be employed in other multi-tone detection systems.

## 12.1  MULTI-TONE DETECTION

Multi-tone detection is the process of detecting the presence of spectral tones, each of which has the frequencies $\omega_1$, $\omega_2$, . . . $\omega_k$, where $k=8$ in our application. A given combination of tones is used to represent a symbol of information, so the detection system's function is to determine what tones are present in the $x(n)$ input signal. A traditional method for multi-tone detection is illustrated in Figure 12-1.

As shown in this figure, the incoming $x(n)$ multi-tone signal is multiplied by the frequency references $\exp(j\omega_k t)$ for all possible multi-tone frequencies $\omega_k$, down-converting any incoming tones so they become centered at zero hertz. Next, the complex $u_k(n)$ products are lowpass filtered. Finally, the magnitudes of the complex lowpass-filtered sequences are logically compared to determine which of the $\omega_k$ tones are present. For the multi-tone detection system in Figure 12-1, the DSP tricks employed are as follows:
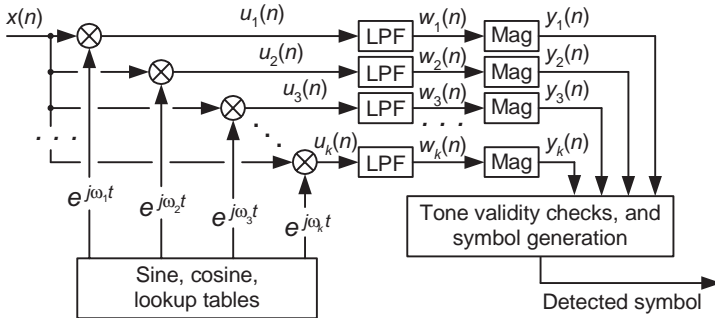
**Figure 12-1**    Traditional multi-tone detection process.

- Use a 1-bit signal representation so multiplications are replaced with exclusive-or operations.
- Group signal samples into 8-bit words, so eight signal samples are manipulated simultaneously in one CPU instruction cycle.
- Perform multiplier-free complex downconversion using a lookup table (LUT).
- Perform multiplier-free lowpass filtering.
- Perform computation-free elimination of quantization DC bias error.
- perform multiplier-free complex sample magnitude estimation.

## 12.2   COMPLEX DOWN CONVERSION

As a first trick to simplify our implementation, the $x(t)$ input signal is converted from an analog signal to a 1-bit binary signal, and the $\exp(j\omega_k t)$ reference frequencies are quantized to a 1-bit representation. Because of this step, the downconversion multiplication of 1-bit data by a 1-bit reference frequency sample can be performed using a simple exclusive-or (XOR) operation. To improve computational efficiency the incoming 1-bit data samples from the comparator, arriving at an 8 kHz sample rate, are collected into one 8-bit byte. The 8-bit XOR operations, one XOR for the sine part and one XOR for the cosine part of the complex reference frequencies, are performed to process eight $x(n)$ samples simultaneously. The logical zeros in the result of XOR operations correspond to the situation when the input is in-phase with an $\exp(j\omega_k t)$ reference, and the logical ones of the XOR result indicate that the input is in opposite phase with an $\exp(j\omega_k t)$ reference.

The number of zeros and ones in the XOR results, which comprise the real and imaginary parts of the $u_k(n)$ sequences, can be found by counting the non-zero bits directly. However, to enhance execution speed, we use a LUT indexed by the XOR result instead of the direct count of ones and zeros.

The XOR lookup table entries, optimized to provide the best numeric accuracy in conjunction with the follow-on lowpass filters, take advantage of the full numeric

**Table 12-1**   XOR Lookup Table

| N | LUT entry, $u_k(n)$ |
| --- | --- |
| 0 | −111 |
| 1 | −81 |
| 2 | −52 |
| 3 | −22 |
| 4 | +8 |
| 5 | +38 |
| 6 | +68 |
| 7 | +97 |
| 8 | +127 |

range (−128 . . . +127) of a signed 8-bit word. If $N$ is the number of ones in an 8-bit XOR result, the LUT entries are computed as follows:

$$\text{Table}[N] = \text{Round}[29.75(N-4)+8] \qquad (12\text{–}1)$$

where Round [·] denotes rounding to the nearest integer. Using (12–1) for our application, the XOR LUT contains the entries listed in Table 12-1. Because XOR results are 8-bit words, the XOR LUT has 256 entries where each entry is one of the nine values in Table 12-1 depending on the number of logical ones in an XOR result. The multiplication factor 29.75 in (12–1) was chosen to partition the LUT entry range, −128 . . . +127, into nine intervals while accommodating the +8 term. (The constant term in (12–1) equal to +8 will be explained shortly.)

The reference frequency samples (actually squarewaves) are also stored in LUTs. Ideally, the size of the sine and cosine LUTs should be equal to the least common period for all DTMF frequencies. However, the least common period of the DTMF frequencies is one second. For the sample rate of 8 kHz, the reference frequency LUT sizes are equal to 16k ($2^{14}$). Because tables of this size cannot be realized in low-end microcontrollers, the values of the frequencies were modified to fit the least common period into a smaller table. We found that the common period of 32 milliseconds is a good compromise between the frequency accuracy and the size of the LUT. In this case the size of the LUT is equal to 512 bytes, and the difference between the LUT frequencies and the DTMF standard frequencies is 10 Hz or less. This mismatch does not affect the operation of the multi-tone detector.

The LUT of 512 bytes for the reference frequencies may be too large for some applications. Numerically controlled oscillator (NCO) sine and cosine generation can be used as an alternative [1], [2]. In that scenario, typical 16-bit NCOs require an extra 16 bytes of random-access memory (RAM) and create an additional computing workload on the order of the three million operations per second (MIPS); however the NCO method frees the 512 bytes of ROM.
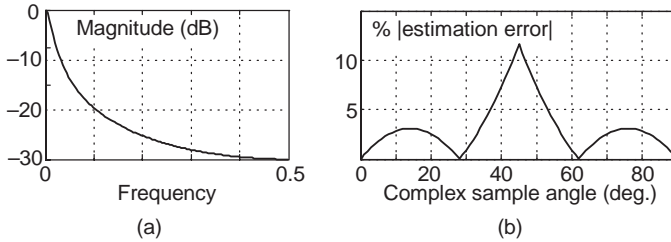
**Figure 12-2**  Detector performance: (a) lowpass filter response; (b) approximation error of the $w_k(n)$ magnitude.

## 12.3  LOWPASS FILTERING

The $w_k(n)$ product sequences must be lowpass filtered. The next trick is that we apply the $w_k(n)$ sequences to a bank of computationally efficient lowpass filters defined by

$$w_k(n) = w_k(n-1) + [u_k(n) - w_k(n-1)]/16. \qquad (12\text{--}2)$$

This economical filter requires only a single data memory storage location and no filter coefficient storage. To enhance execution speed, we implement the divide by 16 with an arithmetic right-shift by 4 bits. The filter outputs are computed with the precision of 8 bits.

The astute reader may recognize (12–2) as the computationally efficient *exponential averaging* lowpass filter. That filter, whose frequency magnitude response is shown in Figure 12-2(a), exhibits nonlinear phase but that is of no consequence in our application. The filter's weighting factor of 1/16 is determined by the filter's necessary 3 dB bandwidth of 10 Hz, as mandated by the accuracy of the entries stored in the reference frequencies lookup table.

When the arithmetic right-shift is executed in the lowpass filters, a round-to-the-nearest-integer operation is performed to improve precision by adding 8 to the number before making the right-shift by 4 bits. Instead of repeatedly adding 8 to each filter output sample, the trick we use next to eliminate all these addition operations is to merely add a constant value of 8 to the XOR LUT. This scheme accounts for the constant 8 term in (12–1).

## 12.4  MAGNITUDE APPROXIMATION

Every 16 ms, the magnitudes of all the complex $w_k(n)$ sequences are computed by obtaining the real and imaginary parts of $w_k(n)$ and using the approximation

$$|w_k(n)| \approx y_k(n) = \max\{|\text{real}[w_k(n)]|, |\text{imag}[w_k(n)]|\}$$
$$+ \min\{|\text{real}[w_k(n)]|, |\text{imag}[w_k(n)]|\}/4 \qquad (12\text{--}3)$$

to estimate the magnitude of the complex $w_k(n)$ sequences. This magnitude approximation trick requires no multiplications because the multiplication by 1/4 is imple-

mented using an arithmetic right-shift by 2 bits. The maximum magnitude estimation error of (12–3), shown in Figure 12-2(b), is equal to 11.6%, which is acceptable for the operation of the DTMF decoder.

## 12.5  VALIDITY CHECKING

The computed magnitudes are processed to decide whether valid DTMF tones are present and, if so, the DTMF symbol is decoded. First, the three maximum magnitudes are found and arranged in descending order: $M_1 \geq M_2 \geq M_3$. Then the validity conditions are checked using the following rules:

1. The two frequencies corresponding to $M_1$ and $M_2$ must fall into the two different touch-tone telephone frequency groups (the <1 kHz group and the >1 kHz group).
2. Magnitude $M_2$ must be greater than the absolute threshold value $T_1$.
3. Ratio $M_2/M_3$ must be greater than the relative threshold value $T_2$.

The threshold values $T_1$ and $T_2$ depend on the real-world system's characteristics such as the signal sample rate, the analog signal's spectrum, the analog filtering prior to digital processing, the probability density of the signal and the noise, the desired probability of false alarm, and the probability of detection. As such, the threshold values are found empirically while working with the hardware implementation of the multi-tone detector. Note that if we normalize the ideal single-tone maximum amplitude at the output to unity, the value of the threshold $T_1$ will be on the order of the absolute value range of the XOR LUT divided by four. As such, in this application $T_1$ is approximately 120/4 = 30. The relative threshold $T_2$ does not depend on tone amplitudes. A typical value for $T_2$ is approximately 2.

Also note that the third conditional check above does not require a division operation. That condition is verified if $M_2 \geq T_2 M_3$. To enhance speed, the multiplication by the constant threshold $T_2$ is implemented by arithmetic shift and addition operations. Finally, if all three validity conditions are satisfied, the DTMF tones are considered to be valid. This validity checking allows us to distinguish true DTMF tones from speech, the silence in speech pauses, noise, or other signals.

## 12.6  IMPLEMENTATION ISSUES

The multi-tone detection algorithm requires roughly 1000 bytes of read-only memory (ROM), and 64 bytes of read/write memory (RAM). The algorithm requires no analog-to-digital (A/D) converter, no multiply operations, and its processing workload is quite low, equal to only 0.5 MIPS. (For comparison, the DTMF detection system described in [3] requires 24 MIPS of processing power on a fixed-point DSP chip.) As such, fortunately, it is possible to implement the DTMF multi-tone detector/decoder using a low-cost 8-bit microcontroller such as Atmel's AVR, Microchip Technology's PIC, Freescale's HC08, or the ×51 family of microcontrollers.

## 12.7   OTHER CONSIDERATIONS

The performance of the above algorithm is satisfactory for most practical DTMF detection applications. However, in some cases more robust processing is needed. For example, if the DTMF signal has become distorted by the processing of a low-bitrate speech coder/decoder, the above algorithm may not operate properly. The weakness of the algorithm is its sensitivity to *frequency twist* and other noise interference. Frequency twist occurs when the power levels of the two detected tones differ by 4–8 dB, and the sensitivity in this case is due to the input signal being digitized to 1 bit by the comparator (a highly nonlinear procedure).

To illustrate this behavior, Figure 12-3(a) shows the output spectra of the 1-bit comparator for the input tones of 944 and 1477 Hz when no frequency twist occurs. There we can see the mix of the input tones and the miscellaneous nonlinear spectral products. If the amplitudes of both input tones are equal, the nonlinear spectral artifacts are at least 10 dB lower than the desired signals. Figure 12-3(b) also shows the output spectra of the comparator with frequency twist. If the amplitudes of the two input tones differ by 6 dB or more (twist), then the strongest nonlinear spectral products have approximately equal amplitude as that of the weakest of the two input tones. Some of the products are falling into the bandwidth of the lowpass filters. Note also that a difference in the levels of input tones equal to 6 dB causes that difference to be 12 dB after the comparator, which limits the performance of the multi-tone detector. As such, we caution the reader that 1-bit quantization is not always appropriate for signal detection systems. Sadly, there are no hard-and-fast rules to identify the cases when 1-bit quantization can be used.

For the detection of spectrally simple, high-signal-to-noise-ratio, stable-amplitude, oversampled signals, the 1-bit quantization may be applicable. However, because the statistical analysis of 1-bit quantization errors is complicated, careful
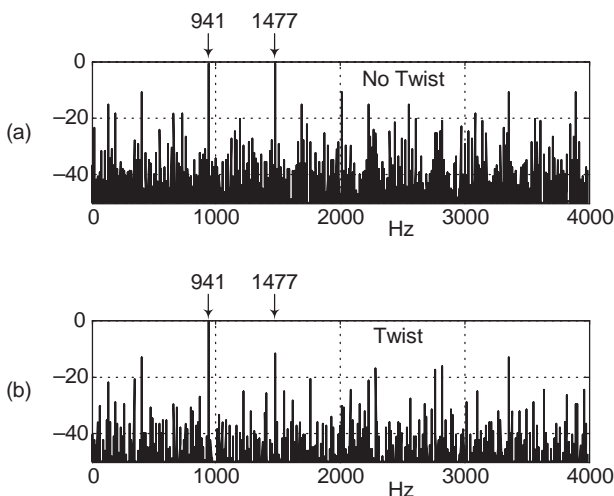


**Figure 12-3**   One-bit comparator output spectra: (a) no frequency twist; (b) with frequency twist.

modeling must be conducted to evaluate the nonlinear performance and spectral aliasing effects of 1-bit quantization.

If improved performance beyond that of 1-bit quantization is required, a similar algorithm can be applied for processing $x(n)$ input signals from an A/D converter. The multi-bit $x(n)$ signal's multiplication by the 1-bit reference frequencies can, again, be performed using simple 16-bit addition and subtraction operations. The A/D converter-based algorithm is able to decode the DTMF under any conditions; however, the trade-off is that the processing has to be done for every sample, not at eight samples simultaneously. This increases the computing burden to several MIPS.

Finally, note that the DTMF detection algorithm presented here is not strictly compliant with the ITU/Bellcore standards [4]. However, this is not detrimental in most practical cases.

## 12.8   REFERENCES

[1] ANALOG DEVICES INC., "A Technical Tutorial on Digital Signal Synthesis," [Online: http://www. analog.com/UploadedFiles/Tutorials/450968421DDS_Tutorial_rev12-2-99.pdf.]

[2] L. CORDESSES, "Direct Digital Synthesis: A Tool for Periodic Wave Generation," *IEEE Signal Processing Magazine: DSP Tips & Tricks Column*, vol. 21, no. 2, July 2005, pp. 50–54.

[3] M. FELDER, J. MASON, and B. EVANS, "Efficient dual-tone multifrequency detection using the non-uniform discrete Fourier transform," *IEEE Signal Processing Letters*, vol. 5, no. 7, July 1998, pp. 160–163.

[4] R. FREEMAN, *Reference Manual for Telecommunications Engineering*. Wiley-Interscience, New York, 2002.

## EDITOR COMMENTS

One-bit representation (quantization) of signals is an interesting and tricky process. For additional examples of systems using 1-bit signal quantization, see Chapters 5 and 22.

The weighting factor of the lowpass filters (exponential averagers) in this chapter was 1/16 in order to achieve a 3 dB bandwidth of 10 Hz when the filter input sample rate is 1 kHz. The following shows how to obtain that weighting factor.

We can compute the appropriate value of an exponential averager's weighting factor, $W$, to achieve any desired filter 3 dB bandwidth. If $f_c$ is the desired positive *cutoff* frequency in Hz where the frequency magnitude response is 3 dB below the averager's zero-Hz response, then the value of $W$ needed to achieve such an $f_c$ cutoff frequency is

$$W = \cos(\Omega) - 1 + \sqrt{\cos^2(\Omega) - 4\cos(\Omega) + 3} \qquad (12–4)$$

where $\omega = 2\omega f_c/f_s$ and $f_s$ is the filter's input sample rate in Hz. So when $f_c = 10$ Hz and $f_s = 1$ kHz, expression (12–4) yields a desired weighting factor of $W = 0.061$. Because $1/16 = 0.0625$ is very close to the desired value for $W$, a weighting factor of 1/16 was acceptable for use in (12–2).
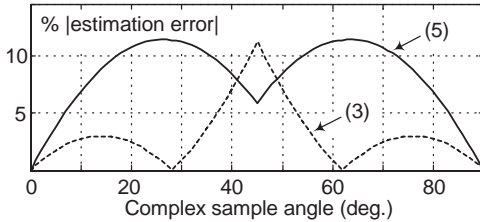
**Figure 12-4**    Magnitude approximation errors in percent.


To elaborate on the lowpass filters' complex output sample magnitude estimation algorithm, the two magnitude estimation schemes appropriate for use in this application are the above expression (12–3) and a similar algorithm defined by

$$|w_k(n)| \approx y_k(n) = \max\{|\text{real}[w_k(n)]|, |\text{imag}[w_k(n)]|\} \\ + \min\{|\text{real}[w_k(n)]|, |\text{imag}[w_k(n)]|\}/2.$$
(12–5)

The difference between (12–3) and (12–5) is their second terms' scaling factors. For both scaling factors, the multiplication can be implemented with binary right-shifts. Both expressions have almost identical maximum error as shown in Figure 12-4. However, (12–3) has the lowest average error, making it the optimum choice with respect to minimizing the average magnitude estimation error. Complex sample magnitude approximation algorithms similar to (12–3) and (12–5) having improved accuracy at the expense of additional shift and add/subtract operations are described in Chapter 16.

# Chapter 13

# Turning Overlap-Save into a Multiband, Mixing, Downsampling Filter Bank

## Mark Borgerding

*3dB Labs, Inc.*

**I**n this chapter, we show how to extend the popular overlap-save fast convolution filtering technique to create a flexible and computationally efficient bank of filters, with frequency translation and decimation implemented in the frequency domain. In addition, we supply some tips for choosing appropriate fast Fourier transform (FFT) size.

Fast convolution is a well-known and powerful filtering technique. All but the shortest finite impulse response (FIR) filters can be implemented more efficiently in the frequency domain than when performed directly in the time domain. The longer the filter impulse response is, the greater the speed advantage of fast convolution.

When more than one output is filtered from a single input, some parts of the fast convolution algorithm are redundant. Removing this redundancy increases fast convolution's speed even more. Sample rate change by decimation (downsampling) and frequency-translation (mixing) techniques can also be incorporated efficiently in the frequency domain. These concepts can be combined to create a flexible and efficient bank of filters. Such a filter bank can implement mixing, filtering, and decimation of multiple arbitrary channels much faster than direct time-domain implementation.

**135**

## 13.1   SOMETHING OLD AND SOMETHING NEW

The necessary conditions for vanilla-flavored fast convolution are covered pretty well in the literature. However, the choice of FFT size is not. Filtering multiple channels from the same forward FFT requires special conditions not detailed in textbooks. To downsample and shift those channels in the frequency domain requires still more conditions.

The first section is meant to be a quick reminder of the basics before we extend the *overlap-save* (OS) fast convolution technique. If you feel comfortable with these concepts, skip ahead. However, if this review does not jog your memory, check your favorite DSP book for "fast convolution," "overlap-add" (OA), "overlap-save," or "overlap-scrap" [1]–[5].

## 13.2   REVIEW OF FAST CONVOLUTION

The convolution theorem tells us that multiplication in the frequency domain is equivalent to convolution in the time domain [1]. Circular convolution is achieved by multiplying two discrete Fourier transforms (DFTs) to effect convolution of the time sequences the transforms represent. By using the FFT to implement the DFT, the computational complexity of circular convolution is approximately $O(N\log_2 N)$ instead of $O(N^2)$, as in direct linear convolution. Although very small FIR filters are most efficiently implemented with direct convolution, fast convolution is the clear winner as the FIR filters get longer. Conventional wisdom places the efficiency crossover point at 25–30 filter coefficients. The actual value depends on the relative strengths of the platform in question (CPU pipelining, zero-overhead looping, memory addressing modes, etc.). On a desktop processor with a highly optimized FFT library, the value may be as low as 16. On a fixed-point DSP with a single-cycle multiply-accumulate instruction, the efficiency crossover point can be greater than 50 coefficients.

Fast convolution refers to the blockwise use of circular convolution to accomplish linear convolution. Fast convolution can be accomplished by overlap-add or overlap-save methods. Overlap-save is also known as "overlap-scrap" [5]. In OA filtering, each signal data block contains only as many samples as allows circular convolution to be equivalent to linear convolution. The signal data block is zero-padded prior to the FFT to prevent the filter impulse response from "wrapping around" the end of the sequence. OA filtering adds the input-on transient from one block with the input-off transient from the previous block.

In OS filtering, shown in Figure 13-1, no zero-padding is performed on the input data; thus the circular convolution is not equivalent to linear convolution. The portions that wrap around are useless and discarded. To compensate for this, the last part of the previous input block is used as the beginning of the next block. OS requires no addition of transients, making it faster than OA. The OS filtering method is recommended as the basis for the techniques outlined in the remainder of this discussion. The nomenclature "$\text{FFT}_N$" in Figure 13-1 indicates that an FFT's input sequence is zero-padded to a length of $N$ samples, if necessary, before performing the $N$-point FFT.
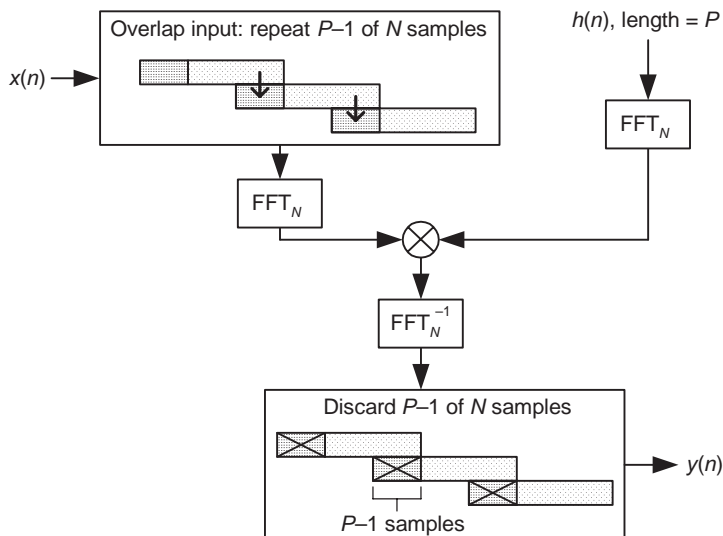
**Figure 13-1**   Overlap-save (OS) filtering, $y(n)=x(n)*h(n)$.

For clarity, the following table defines the symbols used in this material:

Symbol Conventions

| | |
|---|---|
| $x(n)$ | Input data sequence |
| $h(n)$ | FIR filter impulse response |
| $y(n)=x(n)*h(n)$ | Convolution of $x(n)$ and $h(n)$ |
| L | Number of new input samples consumed per data block |
| P | Length of $h(n)$ |
| $N=L+P-1$ | FFT size |
| $V=N/(P-1)$ | Overlap factor, (FFT length)/(filter transient length) |
| D | Decimation factor |

## 13.3   CHOOSING FFT SIZE: COMPLEXITY IS RELATED TO FILTER LENGTH AND OVERLAP FACTOR

Processing a single block of input data that produces $L$ outputs incurs a computational cost related to $N\log(N)=(L+P-1)\log(L+P-1)$. The computational cost per sample is related to $(L+P-1)\log(L+P-1)/L$. The filter length $P$ is generally a fixed parameter, so choosing $L$ such that this equation is minimized gives the theoretically optimal FFT length. It may be worth mentioning that the log base is the radix of the FFT. The only difference between differently based logarithms is a scaling factor. This is irrelevant to "big O" scalability, which generally excludes constant scaling factors.

Larger FFT sizes are more costly to perform, but they also produce more usable (non-wraparound) output samples. In theory, one is penalized more for choosing too small an overlap factor, $V$, than for one too large. In practice, the price of large FFTs paid in computation and/or numerical accuracy may suggest a different conclusion.

> "In theory there is no difference between theory and practice. In practice there is."
> —Yogi Berra (American philosopher and occasional baseball manager)

Here are some other factors to consider while deciding the overlap factor for fast convolution filtering:

- **FFT speed.** It is common for actual FFT computational cost to differ greatly from theory (e.g., due to memory caching).
- **FFT accuracy.** The numerical accuracy of fast convolution filtering is dependent on the error introduced by the FFT-to-inverse-FFT round trip. For floating-point implementations, this may be negligible, but fixed-point processing can lose significant dynamic range in these transforms.
- **Latency.** Fast convolution filtering process increases delay by at least $L$ samples. The longer the FFT, the longer the latency.

While there is no substitute for benchmarking on the target platform, in the absence of benchmarks, choosing a power-of-2 FFT length about four times the length of the FIR filter is a good rule of thumb.

## 13.4  FILTERING MULTIPLE CHANNELS: REUSE THE FORWARD FFT

It is often desirable to apply multiple filters against the same input sample sequence. In these cases, the advantages of fast convolution filtering become even greater. The computationally expensive operations are the forward FFT, the inverse FFT, and the multiplication of the frequency responses. The forward FFT needs to be computed just once. This is roughly a "Buy one filter—get one 40% off" sale.

In order to realize this computational cost savings for two or more filters, all filters must have the same impulse response length. This condition can always be achieved by zero padding the shorter filters. Alternatively, the engineer may redesign the shorter filter(s) to make use of the additional coefficients without increasing the computational workload.

## 13.5  FREQUENCY DOMAIN DOWNSAMPLING: ALIASING IS YOUR FRIEND

The most intuitive method for reducing sample rate in the frequency domain is to simply perform a smaller inverse FFT using only those frequency bins of interest.

This is not a 100% replacement for time domain downsampling. This simple method will cause ripples (Gibbs phenomenon) at FFT buffer boundaries caused by the multiplication in the frequency domain by a rectangular window.

The sampling theorem tells us that sampling a continuous signal aliases energy at frequencies higher than the Nyquist rate (half the signal sample rate) back into the baseband spectrum (below the Nyquist rate). This is as true for decimation of a digital sequence as it is for analog-to-digital conversion [6]. Aliasing is a natural part of downsampling. In order to accurately implement downsampling in the frequency domain, it is necessary to preserve this behavior. It should be noted that, with a suitable antialiasing filter, the energy outside the selected bins might be negligible. This discussion is concerned with the steps necessary for equivalence. The designer should decide how much aliasing, if any, is necessary.

Decimation (i.e., downsampling) can be performed exactly in the frequency domain by coherently adding the frequency components to be aliased [7]. The following octave/MATLAB code demonstrates how to swap the order of an inverse DFT and decimation:

```
% Make up a completely random frequency spectrum
Fx = randn(1,1024) + i*randn(1,1024);
% Time-domain decimation -- inverse transform then decimate
x_full_rate = ifft(Fx);
x_time_dom_dec = x_full_rate(1:4:1024); % Retain every fourth sample
% Frequency-domain decimation, alias first, then inverse transform
Fx_alias = Fx(1:256) + Fx(257:512) + Fx(513:768) + Fx(769:1024);
x_freq_dom_dec = ifft(Fx_alias)/4;
```

The sequences x_time_dom_dec and x_freq_dom_dec are equal to each other. The above sample code assumes a complex time-domain sequence for generality. The division by 4 in the last step accounts for the difference in scaling factors between the inverse FFT sizes. As various FFT libraries handle scaling differently, the designer should keep this in mind during implementation. It's worth noting that this discussion assumes the FFT length is a multiple of the decimation rate $D$. That is, $N/D$ must be an integer.

To implement time-domain decimation in the frequency domain as part of fast convolution filtering, the following conditions must be met.

1. The FIR filter order must be a multiple of the decimation rate $D$, $P-1 = K_1 D$.
2. The FFT length must be a multiple of the decimation rate $D$, $L+P-1 = K_2 D$, where
   - $D$ is the decimation rate or the least common multiple of the decimation rates for multiple channels.
   - $K_1$ and $K_2$ are integers.

Note that if the overlap factor $V$ is an integer, then the first condition implies the second. It is worth noting that others have also explored the concepts of rate

conversion in overlap-add/save [8]. Also note that decimation by large primes can lead to FFT inefficiency. It may be wise to decimate by such factors in the time domain.

## 13.6   MIXING AND OS FILTERING: ROTATE THE FREQUENCY DOMAIN FOR COARSE MIXING

Mixing, or frequency shifting, is the multiplication of an input signal by a complex sinusoid [1]. It is equivalent to convolving the frequency spectrum of an input signal with the spectrum of a sinusoid. In other words, the frequency spectrum is shifted by the mixing frequency.

It is possible to implement time-domain mixing in the frequency domain by simply rotating the DFT sequence, but there are limitations:

1. The precision with which one can mix a signal by rotating a DFT sequence is limited by the resolution of the DFT.

2. The mixing precision is limited further by the fact that we don't use a complete buffer of output in fast convolution. We use only $L$ samples. We must restrict the mixing to the subset of frequencies whose periods complete in those $L$ samples. Otherwise, phase discontinuities occur. That is, one can shift only in multiples of $V$ bins.

The number of "bins" to rotate is

$$N_{rot} = round\left(\frac{Nf_r}{Vf_s}\right) \cdot V$$

where $f_r$ is the desired mixing frequency and $f_s$ is the sampling frequency. The second limitation may be overcome by using a bank of filters corresponding to different phases. However, this increase in design/code complexity probably does not outweigh the meager cost of multiplying by a complex phasor.

If coarse-grained mixing is unacceptable, mixing in the time domain is a better solution. The general solution to allow multiple channels with multiple mixing frequencies is to postpone the mixing operation until the filtered, decimated data is back in the time domain.

If mixing is performed in the time domain:

• All filters must be specified in terms of the input frequency (i.e., nonshifted) spectrum.

• The complex sinusoid used for mixing the output signal must be created at the output rate.

## 13.7   PUTTING IT ALL TOGETHER

By making efficient implementations of conceptually simple tools we help ourselves to create simple designs that are as efficient as they are easy to describe. Humans
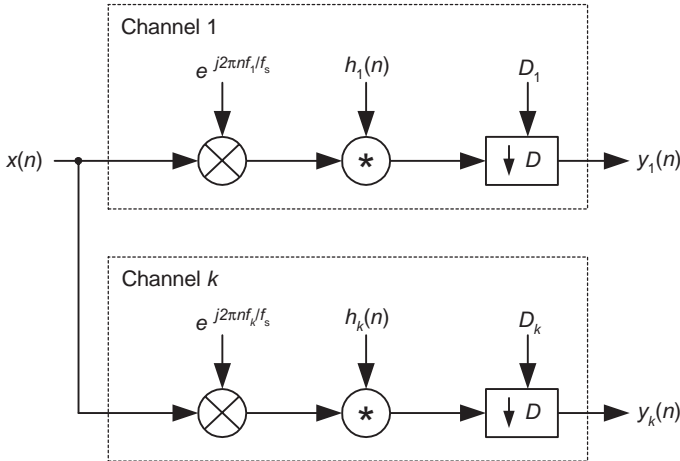
**Figure 13-2**  Conceptual model of a filter bank.

are greatly affected by the simplicity of the concepts and tools used in designing and describing a system. We owe it to ourselves as humans to make use of simple concepts whenever possible.

"Things should be described as simply as possible, but no simpler."
—A. Einstein.

We owe it to ourselves as engineers to realize those simple concepts as efficiently as possible.

The familiar and simple concepts shown in Figure 13-2 may be used for the design of mixed, filtered, and decimated channels. The design may be implemented more efficiently using the equivalent structure shown in Figure 13-3.

## 13.8  FOOD FOR THOUGHT

Answering a question or exploring an idea often leads to more questions. Along the path to understanding the concepts detailed in this chapter, various side paths have been glimpsed but not fully explored. Here are a few such side paths:

- Highly decimated channels are generally filtered by correspondingly narrow bandwidth filters. Those frequency bins whose combined power falls below a given threshold may be ignored without adversely affecting the output. Skipping the multiplications and additions associated with filtering and aliasing those bins can speed processing at the expense of introducing arbitrarily low error energy. Some researchers have suggested ways to compensate for the error created by zeroing all frequency bins outside the retained spectrum. By allowing some of these bins to be aliased the error is made arbitrarily low and compensation is then made unnecessary [7].
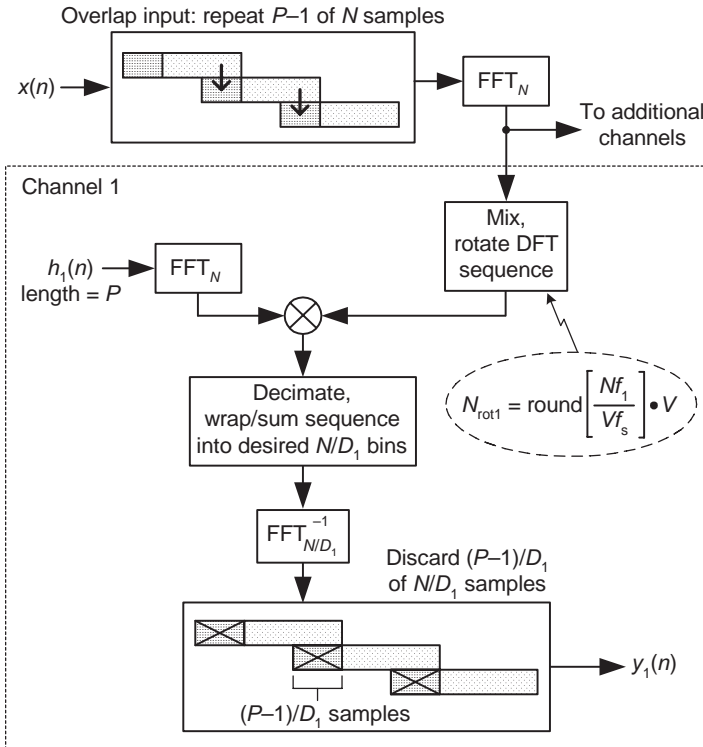
**Figure 13-3**    Overlap-save filter bank.

- If channels of highly differing rates and filter orders are required, it may be more efficient to break up the structure using partitioned convolution or multiple stages of fast convolution. What is the crossover point? Could such designs effectively use intermediate FFT stages?

- What application, if any, does multiple fast convolution filtering have with regard to number theoretic transforms (NTT)?

## 13.9   CONCLUSIONS

We outlined considerations for implementing multiple overlap-save channels with decimation and mixing in the frequency domain, as well as supplying recommendations for choosing FFT size. We also provided implementation guidance to streamline this powerful multichannel filtering, downconversion, and decimation process.

## 13.10   REFERENCES

[1] A. Oppenheimer and R. Schafer, *Discrete-Time Signal Processing*. Prentice Hall, Upper Saddle River, NJ, 1989.

[2] L. RABINER and B. GOLD, *Theory and Application of Digital Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1975.

[3] R. LYONS, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 2004.

[4] S. ORFANIDIS, *Introduction to Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[5] M. FRERKING, *Digital Signal Processing in Communication Systems*. Chapman & Hall, New York, 1994.

[6] R. CROCHIERE and L. RABINER, *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1983.

[7] M. BOUCHERET, I. MORTENSEN, and H. FAVARO, "Fast Convolution Filter Banks for Satellite Payloads with On-Board Processing," *IEEE Journal on Selected Areas in Communications*, February 1999.

[8] S. MURAMATSU and H. KIYA, "Extended Overlap-Add and -Save Methods for Multirate Signal Processing," *IEEE Trans. on Signal Processing*, September 1997.

## EDITOR COMMENTS

One important aspect of this overlap-save fast convolution scheme is that the FFT indexing bit-reversal problem inherent in some hardware-FFT implementations is not an issue here. If the identical FFT structures used in Figure 13-1 produce $X(m)$ and $H(m)$ having bit-reversed indexes, the multiplication can still be performed directly on the scrambled $H(m)$ and $X(m)$ sequences. Next, an appropriate inverse FFT structure can be used that expects bit-reversed input data. That inverse FFT then provides an output sequence whose time-domain indexing is in the correct order.

An implementation issue to keep in mind is that the complex amplitudes of the standard radix-2 FFT's output samples are proportional to the FFT size, $N$. As such, we can think of the FFT as having a gain of $N$. (That's why the standard inverse FFT has a *scaling*, or *normalizing*, factor of $1/N$.) So the product of two FFT outputs, as in our fast convolution process, will have a gain proportional to $N^2$ and the inverse FFT will have a normalizing gain reduction of only $1/N$. Thus, depending on the forward and inverse FFT software being used, the fast convolution filtering method may have an overall gain that is not unity. The importance of this possible nonunity gain depends, of course, on the numerical format of the data as well as the user's filtering application. We won't dwell on this subject here because it's so dependent on the forward and inverse FFT software routines being used. We'll merely suggest that this normalization topic be considered during the design of any fast convolution system.

# Chapter 14

# Sliding Spectrum Analysis

**Eric Jacobsen**

*Abineau Communications*

**Richard Lyons**

*Besser Associates*

**T**he standard method for spectrum analysis in DSP is the discrete Fourier transform (DFT), typically implemented using a fast Fourier transform (FFT) algorithm. However, there are applications that require spectrum analysis only over a subset of the $N$ center frequencies of an $N$-point DFT. A popular, as well as efficient, technique for computing sparse DFT results is the Goertzel algorithm, which computes a single complex DFT spectral bin value for every $N$ input time samples. This chapter describes *sliding spectrum analysis* techniques whose spectral bin output rates are equal to the input data rate, on a sample-by-sample basis, with the advantage that they require fewer computations than the Goertzel algorithm for real-time spectral analysis. In applications where a new DFT output spectrum is desired every sample, or every few samples, the *sliding DFT* is computationally simpler than the traditional radix-2 FFT. We'll start our discussion by providing a brief review of the Goertzel algorithm, and use its behavior as a yardstick to evaluate the performance of the sliding DFT technique. Following that, we will examine stability issues regarding the sliding DFT implementation and review the process of frequency-domain convolution to accomplish time-domain windowing. Finally, a modified sliding DFT structure is proposed that provides improved computational efficiency.

## 14.1   GOERTZEL ALGORITHM

The Goertzel algorithm, used in dual-tone multifrequency decoding and PSK/FSK modem implementations, is commonly used to compute DFT spectra [1]–[4]. The algorithm is implemented in the form of a second-order IIR filter as shown in Figure 14-1. This filter computes a single DFT output (the $k$th bin of an $N$-point DFT) defined by

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi nk/N}. \tag{14-1}$$

The filter's $y(n)$ output is equal to the DFT output frequency coefficient, $X(k)$, at the time index $n=N$. For emphasis, we remind the reader that the filter's $y(n)$ output is not equal to $X(k)$ at any time index when $n \neq N$. The frequency-domain index $k$ is an integer in the range $0 \leq k \leq N-1$. The derivation of this filter's structure is readily available in the literature [5]–[7].

The $z$-domain transfer function of the Goertzel filter is

$$H_G(z) = \frac{1 - e^{-j2\pi k/N}z^{-1}}{1 - 2\cos(2\pi k/N)z^{-1} + z^{-2}} \tag{14-2}$$

with a single $z$-domain zero located at $z = e^{-j2\pi k/N}$ and conjugate poles at $z = e^{\pm j2\pi k/N}$ as shown in Figure 14-2(a). The pole/zero pair at $z = e^{-j2\pi k/N}$ cancel each other. The frequency magnitude response, provided in Figure 14-2(b), shows resonance centered at a normalized frequency of $2\pi k/N$, corresponding to a cyclical frequency $k \cdot f_s/N$ hertz (where $f_s$ is the signal sample rate).

We remind the reader that the typical Goertzel algorithm description in the literature specifies the frequency resonance variable $k$ in (14–2) and Figure 14-1 to be an integer (making the Goertzel filter's output equivalent to an $N$-point DFT bin output). Variable $k$ can in fact be *any* value between 0 and $N-1$, giving us full flexibility in specifying a Goertzel filter's resonance frequency.

While the Goertzel algorithm is derived from the standard DFT equation, it's important to realize that the filter's frequency magnitude response is not the $\sin(x)/(x)$-like response of a single-bin DFT. The Goertzel filter is a complex resonator having an infinite-length unit impulse response, $h(n) = e^{j2\pi nk/N}$, and that's why its magnitude response is so narrow. The time-domain difference equations for the Goertzel filter are
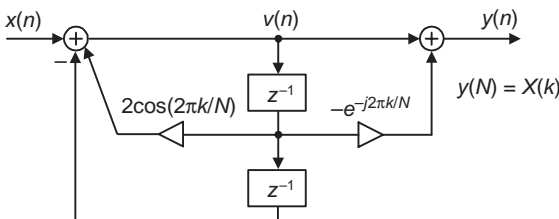


**Figure 14-1**   IIR filter implementation of the Goertzel algorithm.

(a)

(b)
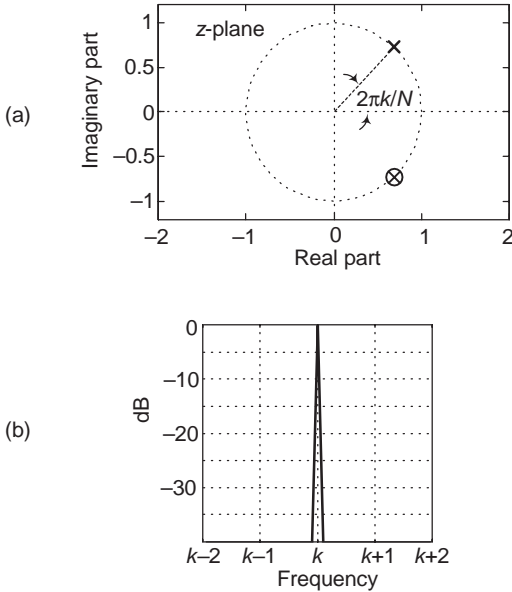
**Figure 14-2**   Goertzel filter: (a) *z*-domain pole/zero locations; (b) frequency magnitude response.

$$v(n) = 2\cos(2\pi k/N)v(n-1) - v(n-2) + x(n). \qquad (14\text{–}3a)$$

$$y(n) = v(n) - e^{-j2\pi k/N}v(n-1). \qquad (14\text{–}3b)$$

An advantage of the Goertzel filter in calculating an *N*-point $X(k)$ DFT bin is that (14–3a) is implemented *N* times, while (14–3b), the feedforward path in Figure 14-1, need only be computed once after the arrival of the *N*th input sample. Thus for real $x(n)$ the filter requires $N+2$ real multiplies and $2N+1$ real adds to compute an *N*-point $X(k)$. However, when modeling the Goertzel filter, if the time index begins at $n=0$, the filter must process $N+1$ time samples with $x(N)=0$ to compute $X(k)$. Now let's look at the sliding DFT process.

## 14.2   SLIDING DISCRETE FOURIER TRANSFORM (SDFT)

The sliding DFT (SDFT) algorithm performs an *N*-point DFT on time samples within a sliding window as shown in Figure 14-3. In this example the SDFT initially computes the DFT of the $N=16$ time samples in Figure 14-3(a). The time window is then advanced one sample, as in Figure 14-3(b), and a new *N*-point DFT is calculated. The value of this process is that each new DFT is efficiently computed directly from the results of the previous DFT. The incremental advance of the time window for each output computation is what leads to the name *sliding DFT* or *sliding-window DFT*.

The principle used for the SDFT is known as the *DFT shifting theorem*, or the *circular shift property* [8]. It states that if the DFT of a windowed (finite-length)
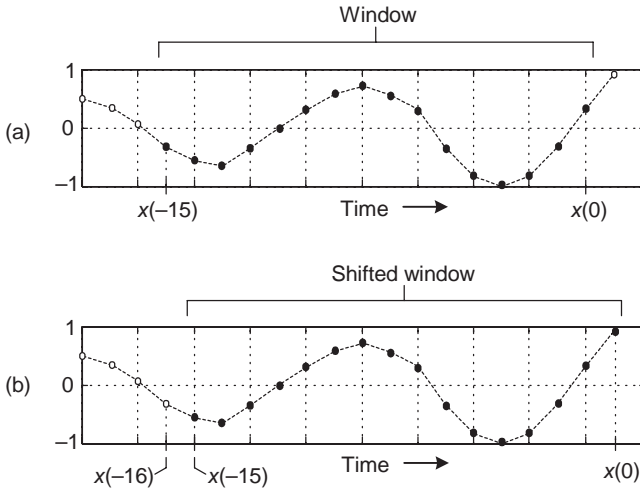
**Figure 14-3**   Signal windowing for two 16-point DFTs: (a) data samples in the first computation; (b) second computation samples.

time-domain sequence is $X(k)$, then the DFT of that sequence, circularly shifted by one sample, is $X(k)e^{j2\pi k/N}$. Thus the spectral components of a shifted time sequence are the original (unshifted) spectral components multiplied by $e^{j2\pi k/N}$, where $k$ is the DFT bin of interest. We use this shift principle to express our sliding DFT process as

$$S_k(n) = e^{j2\pi k/N}[S_k(n-1) + x(n) - x(n-N)] \qquad (14\text{--}4)$$

where $S_k(n)$ is the new spectral component and $S_k(n-1)$ is the previous spectral component. The subscript $k$ reminds us that the spectra are those associated with the $k$th DFT bin.

Equation (14–4), whose derivation is provided in the appendix to this chapter, reveals the value of this process in computing real-time spectra. We calculate $S_k(n)$ by phase shifting the sum of the previous $S_k(n-1)$ with the difference between the current $x(n)$ sample and the $x(n-N)$ sample. The difference between the $x(n)$ and $x(n-N)$ samples can be computed once for each $n$ and used for each $S_k(n)$ computation. So the SDFT requires only one complex multiply and two real adds per output sample.

The computational complexity of each successive $N$-point output is then O($N$) for the sliding DFT compared to O($N^2$) for the DFT and O[$N\log_2(N)$] for the FFT. Unlike the DFT or FFT, however, due to its recursive nature the sliding DFT output *must* be computed for each new input sample. If a new $N$-point DFT output is required only every $N$ inputs, the sliding DFT requires O($N^2$) computations and is equivalent to the DFT. When output computations are required every $M$ input samples, and $M$ is less than $\log_2(N)$, the sliding DFT can be computationally superior to traditional FFT implementations even when all $N$ DFT outputs are required.
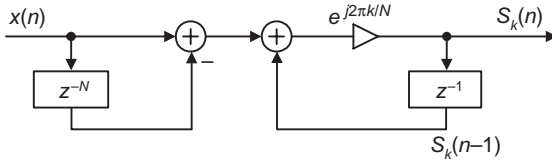
**Figure 14-4**   Single-bin sliding DFT filter structure.

Equation (14–4) leads to the single-bin SDFT filter structure shown in Figure 14-4.

The single-bin SDFT algorithm is implemented as an IIR filter with a comb filter followed by a complex resonator [9]. (If you want to compute all $N$ DFT spectral components, $N$ resonators with $k=0$ to $N-1$ will be needed, all driven by a single comb filter.) The comb filter delay of $N$ samples forces the filter's transient response to be $N-1$ samples in length, so the output will not reach steady state until the $S_k(n)$ sample. In practical applications the algorithm can be initialized with zero-input and zero-output. The output will not be valid, or equivalent to (14–1)'s $X(k)$, until $N$ input samples have been processed. The $z$-domain transfer function for the $k$th bin of the sliding DFT filter is

$$H_{\text{SDFT}}(z) = \frac{e^{j2\pi k/N}(1-z^{-N})}{1-e^{j2\pi k/N}z^{-1}}. \qquad (14\text{--}5)$$

This complex filter has $N$ zeros equally spaced around the $z$-domain's unit circle, due to the $N$-delay comb filter, as well as a single pole canceling the zero at $z=e^{j2\pi k/N}$ as shown in Figure 14-5(a). The SDFT filter's complex $h(n)$ unit impulse response is shown in Figure 14-5(b) for the example where $k=2$ and $N=20$.

Because of the comb subfilter, the SDFT filter's complex sinusoidal unit impulse response is finite in length—truncated in time to $N$ samples—and that property makes the frequency magnitude response of the SDFT filter identical to the $\sin(Nx)/\sin(x)$ response of a single DFT bin centered at a normalized frequency of $2\pi^{k/N}$.

We've encountered a useful property of the SDFT that's not widely known, but is important. If we change the SDFT's comb filter feedforward coefficient (in Figure 14-4) from $-1$ to $+1$, the comb's zeros will be rotated counterclockwise around the unit circle by an angle of $\pi/N$ radians. This situation, for $N=8$, is shown on the right side of Figure 14-6(a). The zeros are located at angles of $2\pi(k+1/2)/N$ radians. The $k=0$ zeros are shown as solid dots. Figure 14-6(b) shows the zeros locations for an $N=9$ SDFT under the two conditions of the comb filter's feedforward coefficient being $-1$ and $+1$.

This alternative situation is useful; we can now expand our set of spectrum analysis center frequencies to more than just $N$ angular frequency points around the unit circle. The analysis frequencies can be either $2\pi k/N$ or $2\pi(k+1/2)/N$, where integer $k$ is in the range $0 \le k \le N-1$. Thus we can build an SDFT analyzer that resonates at any one of $2N$ frequencies between 0 and $f_s$ Hz. Of course, if the comb filter's feedforward coefficient is set to $+1$, the resonator's feedforward coefficient must be $e^{j2\pi(k+1/2)/N}$ to achieve pole/zero cancellation.
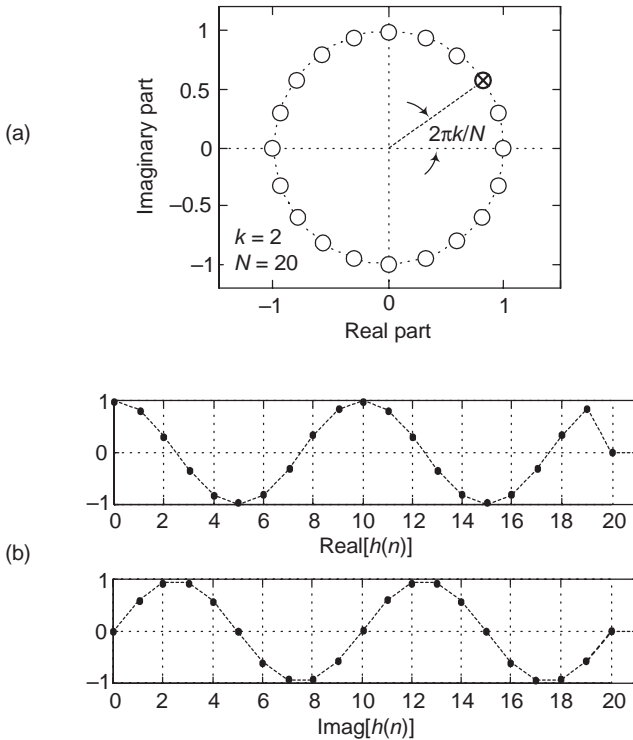
**Figure 14-5**   Sliding DFT characteristics for $k=2$ and $N=20$: (a) impulse response; (b) pole/zero locations.

One of the attributes of the SDFT is that once an $S_k(n-1)$ is obtained, the number of computations to calculate $S_k(n)$ is fixed and independent of $N$. A computational workload comparison between the Goertzel and SDFT filters is provided later in this discussion. Unlike the radix-2 FFT, the SDFT's $N$ can be any positive integer, giving us greater flexibility to *tune* the SDFT's center frequency by defining integer $k$ such that $k=N \cdot f_i/f_s$, when $f_i$ is a frequency of interest in Hz. In addition, the SDFT requires no bit-reversal processing as does the FFT. Like Goertzel, the SDFT is especially efficient for narrowband spectrum analysis.

For completeness, we mention that a radix-2 *sliding FFT* technique exists for computing all $N$ bins of $X(k)$ in (14–1) [10], [11]. This method is computationally attractive because it requires only $N$ complex multiplies to update the $N$-point FFT for all $N$ bins; however it requires $3N$ memory locations ($2N$ for data and $N$ for twiddle coefficients). Unlike the SDFT, the radix-2 sliding FFT scheme requires address bit-reversal processing and restricts $N$ to be an integer power of two.

## 14.3   SDFT STABILITY

The SDFT filter is only marginally stable because its pole resides on the $z$-domain's unit circle. If filter coefficient numerical rounding error is not severe, the SDFT is
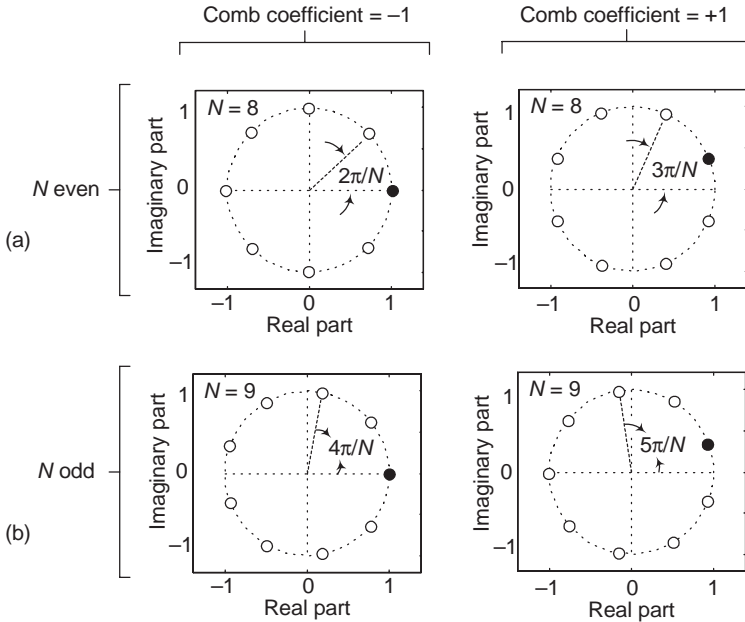
**Figure 14-6**    Four possible orientations of comb filter zeros on the unit circle.

bounded-input-bounded-output stable. Filter instability can be a problem, however, if numerical coefficient rounding causes the filter's pole to move outside the unit circle. We can use a damping factor $r$ to force the pole to be at a radius of $r$ inside the unit circle and guarantee stability using a transfer function of

$$H_{\text{SDFT,gs}}(z) = \frac{re^{j2\pi k/N}(1 - r^N z^{-N})}{1 - re^{j2\pi k/N} z^{-1}} \qquad (14\text{--}6)$$

with the subscript "gs" meaning guaranteed-stable. The stabilized feedforward and feedback coefficients become $-r^N$ and $re^{j2\pi k/N}$ respectively. The difference equation for the stable SDFT filter becomes

$$S_k(n) = re^{j2\pi k/N}[S_k(n-1) + x(n) - r^N x(n-N)] \qquad (14\text{-}7)$$

with the stabilized-filter structure shown in Figure 14-7.

Using a damping factor as in Figure 14-7 guarantees stability, but the $S_k(n)$ output, defined by

$$X_{r<1}(k) = \sum_{n=0}^{N-1} x(n)re^{-j2\pi nk/N} \qquad (14\text{--}8)$$

is no longer exactly equal to the $k$th bin of an $N$-point DFT in (14–1). While the error is reduced by making $r$ very close to (but less than) unity, a scheme does exist for eliminating that error completely once every $N$ output samples at the expense of additional conditional logic operations [12]. Determining whether the damping

**Figure 14-7**    Guaranteed-stable sliding DFT filter structure.

factor $r$ is necessary for a particular SDFT application requires careful empirical investigation.

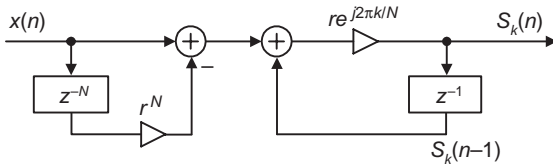Another stabilization method worth consideration for fixed-point applications is decrementing the largest component (either real or imaginary) of the filter's $e^{j2\pi k/N}$ feedback coefficient by one least significant bit. This technique can be applied selectively to problematic output bins and is effective in combating instability due to rounding errors that result in finite-precision $e^{j2\pi k/N}$ coefficients having magnitudes greater than unity.

Like the DFT, the SDFT's output is proportional to $N$, so in fixed-point binary implementations the designer must allocate sufficiently wide registers to hold the computed results.

## 14.4    TIME-DOMAIN WINDOWING IN THE FREQUENCY DOMAIN

The spectral leakage of the SDFT can be reduced by the standard concept of windowing the $x(n)$ input time samples. However, windowing by time-domain multiplication would compromise the computational simplicity of the SDFT. Alternatively, we can implement a time-domain window by means of frequency-domain convolution.

Spectral leakage reduction performed in the frequency domain is accomplished by convolving adjacent $S_k(n)$ values with the DFT of a window function. For example, the DFT of a Hanning window comprises only three non-zero values, $-0.25$, $0.5$, and $-0.25$. As such we can compute a Hanning-windowed $S_k(n)$, the $k$th DFT bin, with a three-point convolution using

$$\text{Hanning-windowed } S_k(n) = -0.25 \cdot S_{k-1}(n) + 0.5 \cdot S_k(n) - 0.25 \cdot S_{k+1}(n). \quad (14\text{--}9)$$

Figure 14-8 shows this process where the comb filter stage need only be implemented once. Thus a Hanning window can be implemented by binary right-shifts and two complex adds for each SDFT bin, making the Hanning window attractive in ASIC and FPGA implementations where single-cycle hardware multiplies are costly. If a gain of four is acceptable, then only two left-shifts (one for the real part and one for the imaginary parts of $S_k(n)$) and two complex adds are required using

$$\text{Hanning-windowed } S_k(n) = -S_{k-1}(n) + 2 \cdot S_k(n) - S_{k+1}(n). \quad (14\text{--}10)$$
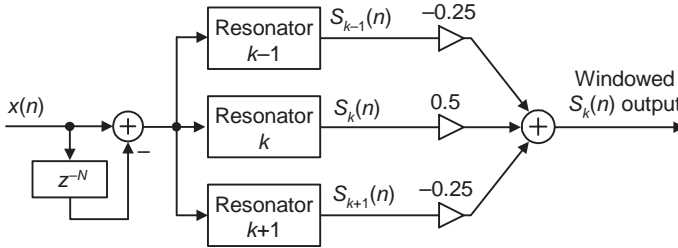
**Figure 14-8**   Three-resonator structure to compute three SDFT bin results, and a three-point convolution.

**Table 14-1**   $\cos^\alpha(x)$ Windows, Frequency Domain Coefficients

| Window function: | $a_0$ | $a_1$ | $a_2$ |
|---|---|---|---|
| Rectangular | 1.0 | — | — |
| Hanning, ($\alpha=2$) | 0.5 | 0.5 | — |
| Blackman, ($\alpha=3$) | 0.42 | 0.5 | 0.08 |
| Exact Blackman, ($\alpha=3$) | 0.4265907 | 0.4965606 | 0.0768487 |
| Hamming | 0.54 | 0.46 | — |

The Hanning window is a member of a category called $\cos^\alpha(x)$ *window functions* [13], [14]. These functions are also known as *generalized cosine windows* because their $N$-point time-domain samples are defined as

$$w(n) = \sum_{m=0}^{\alpha-1} (-1)^m a_m \cos(2\pi mn/N) \qquad (14\text{--}11)$$

where $n=0, 1, 2, \ldots, N-1$, and the integer $\alpha$ specifies the number of terms in the window's time function. These window functions are attractive for frequency domain convolution because their DFTs contain only a few non-zero samples. The frequency domain vectors of various $\cos^\alpha(x)$ window functions follow the form $(1/2)\cdot(a_2, -a_1, 2a_0, -a_1, a_2)$, with a few examples presented in Table 14-1. Additional $\cos^\alpha(x)$ window functions are described in the literature [14].

## 14.5   SLIDING GOERTZEL DFT

We can reduce the number of multiplications required in the SDFT by creating a new pole/zero pair in its $H_{\text{SDFT}}(z)$ system function [15]. This is done by multiplying the numerator and denominator of $H_{\text{SDFT}}(z)$ in (14–5) by the factor $(1-e^{-j2\pi k/N}z^{-1})$ yielding
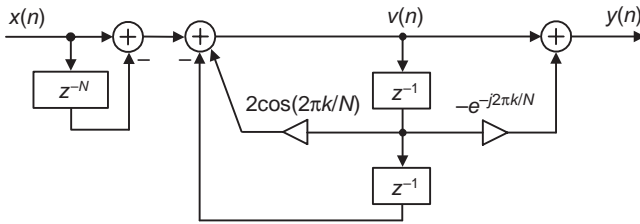
**Figure 14-9**    Structure of the sliding Goertzel DFT filter.

$$H_{SG}(z) = \frac{(1 - e^{-j2\pi k/N} z^{-1})(1 - z^{-N})}{(1 - e^{-j2\pi k/N} z^{-1})(1 - e^{j2\pi k/N} z^{-1})}$$

$$= \frac{(1 - e^{-j2\pi k/N} z^{-1})(1 - z^{-N})}{1 - 2\cos(2\pi k/N)z^{-1} + z^{-2}}$$

(14–12)

where the subscript "SG" means sliding Goertzel. The filter block diagram for $H_{SG}(z)$ is shown in Figure 14-9 where this new filter is recognized as the standard Goertzel filter preceded by a comb filter. The sliding Goertzel DFT filter, unlike the standard Goertzel filter, has a finite-duration impulse response identical to that shown in Figure 14-5(b), for $k=2$ and $N=20$.

Of course, unlike the traditional Goertzel filter in Figure 14-1, the sliding Goertzel DFT filter's complex feedforward computations must be performed for each input time sample. The sliding Goertzel filter's $\sin(Nx)/\sin(x)$ frequency magnitude response, for $k=2$ and $N=20$, is provided in Figure 14-10(a). The asymmetrical frequency response is defined by the filter's $N$ zeros equally spaced around the $z$-domain's unit circle in Figure 14-10(b) due to the $N$-delay comb filter, as well as an additional (uncanceled) zero located at $z=e^{-j2\pi k/N}$ on account of the $(1 - e^{-j2\pi k/N}z^{-1})$ factor in the $H_{SG}(z)$ transfer function's numerator. In addition, the filter has conjugate poles canceling zeros at $z=e^{\pm j2\pi k/N}$.

The sliding Goertzel DFT filter is of interest because its computational workload is less than that of the SDFT. This is because the $v(n)$ samples in Figure 14-9 are real-only due to the real-only feedback coefficients. A single-bin DFT computational comparison, for real-only inputs, is provided in Table 14-2. For real-time processing requiring spectral updates on a sample-by-sample basis, the sliding Goertzel method requires fewer multiplies than either the SDFT or the traditional Goertzel algorithm.

## 14.6    CONCLUSIONS

The sliding DFT process for spectrum analysis was presented, and shown to be more efficient than the popular Goertzel algorithm for sample-by-sample DFT bin computations. The sliding DFT provides computational advantages over the traditional DFT or FFT for many applications requiring successive output calculations, especially when only a subset of the DFT output bins are required. Methods for output
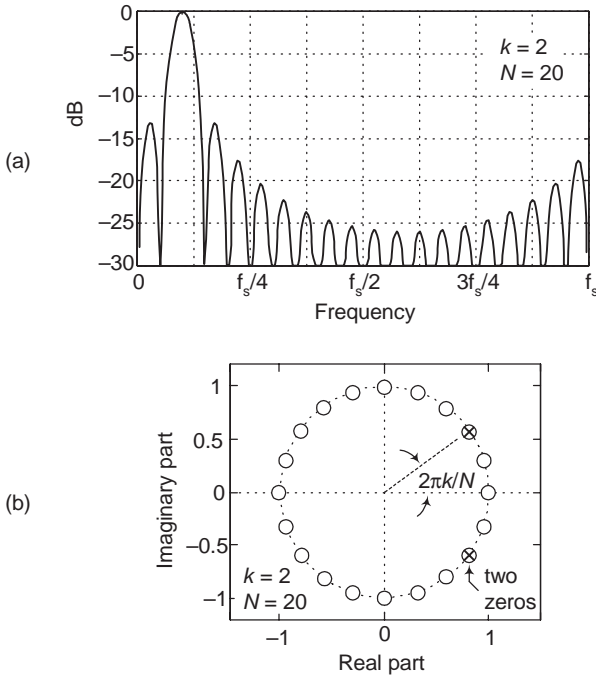
**Figure 14-10**   Sliding Goertzel filter for $N=20$ and $k=2$: (a) frequency magnitude response; (b) $z$-domain pole/zero locations.

**Table 14-2**   Single-Bin DFT Comparison

| Method | Single $S_k(n)$ computation: | | Next $S_k(n+1)$ computation: | |
|---|---|---|---|---|
| | Real multiplies: | Real adds: | Real multiplies: | Real adds: |
| DFT | $2N$ | $2N$ | $2N$ | $2N$ |
| Goertzel | $N+2$ | $2N+1$ | $N+2$ | $2N+1$ |
| Sliding DFT | $4N$ | $4N$ | 4 | 4 |
| Sliding Goertzel | $N+2$ | $3N+1$ | 3 | 4 |

stabilization as well as time-domain data windowing by means of frequency-domain convolution were also discussed. A modified sliding DFT algorithm, called the sliding Goertzel DFT, was proposed to further reduce computational workload.

## 14.7   REFERENCES

[1]  M. FELDER, J. MASON, and B. EVANS, "Efficient Dual-Tone Multi-Frequency Detection Using the Non-uniform Discrete Fourier Transform," *IEEE Signal Processing Lett.*, vol. 5, July 1998, pp. 160–163.

[2] ANALOG DEVICES INC., *ADSP-2100 Family User's Manual*, 3rd ed., Chapter 14, vol. 1, 1995, Norwood, MA. [Online: http://www.analog.com/Analog_Root/static/library/dspManuals/Using_ADSP-2100_Vol1_books.html.]

[3] S. GAY, J. HARTUNG, and G. SMITH, "Algorithms for Multi-Channel DTMF Detection for the WERDSP32 Family," Proceedings on the International Conference on ASSP, 1989, pp. 1134–1137.

[4] K. BANKS, "The Goertzel Algorithm," *Embedded Systems Programming*, September 2002, pp. 34–42.

[5] G. GOERTZEL, "An Algorithm for the Evaluation of Finite Trigonometric Series," *American Math. Monthly*, vol. 65, 1958, pp. 34–35.

[6] J. PROAKIS and D. MANOLAKIS, *Digital Signal Processing: Principles, Algorithms, and Applications*, 3rd ed., Prentice Hall, Upper Saddle River, 1996, pp. 480–481.

[7] A. OPPENHEIM, R. SCHAFER, and J. BUCK, *Discrete-Time Signal Processing*, 2nd ed. Prentice Hall, Upper Saddle River, 1996, pp. 633–634.

[8] T. SPRINGER, "Sliding FFT Computes Frequency Spectra in Real Time," *EDN*, September 29, 1988, pp. 161–170.

[9] L. RABINER and B. GOLD, *Theory and Application of Digital Signal Processing*, Prentice Hall, Upper Saddle River, 1975, pp. 382–383.

[10] B. FARHANG-BOROUJENY and Y. LIM, "A Comment on the Computational Complexity of Sliding FFT," *IEEE Trans. Circuits and Syst. II*, vol. 39, no. 12, December 1992, pp. 875–876.

[11] B. FARHANG-BOROUJENY and S. GAZOR, "Generalized Sliding FFT and its Application to Implementation of Block LMS Adaptive Filters," *IEEE Trans. Sig. Proc.*, vol. 42, no. 3, March 1994, pp. 532–538.

[12] S. DOUGLAS and J. SOH, "A Numerically-Stable Sliding-Window Estimator and Its Application to Adaptive Filters," Proc. 31st Annual Asilomar Conf. on Signals, Systems, and Computers, Pacific Grove, CA, vol. 1, November 1997, pp. 111–115.

[13] F. HARRIS, "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proc. IEEE*, vol. 66, January 1978, pp. 51–84.

[14] A. NUTTALL, "Some Windows with Very Good Sidelobe Behavior," *IEEE Trans.*, vol. 29, no. 1, February 1981, pp. 84–91.

[15] K. LARSON, Texas Instruments Inc., Private communication, November 2002.

## 14.8  APPENDIX

The derivation of the general SDFT can be understood starting with the definition of the DFT of a contiguous subset of a longer sequence. For this derivation we consider a DFT of length $N$ computed on a subset of an input sequence with length of at least $N+q+1$ where $q$ is the index of the start of the DFT window in the input sequence. The additional time unit increment is merely to accommodate the slide to the next window.

We start the derivation with the definition of the $k$th DFT bin for a window starting at the $q$th element of the input sequence:

$$X(k, q) = \sum_{n=0}^{N-1} x(n+q)e^{-j2\pi nk/N}. \qquad (14\text{–}A1)$$

The transform of the $(q+1)$th window, the next DFT in the sliding sequence, is then:

$$X(k, q+1) = \sum_{n=0}^{N-1} x(n+q+1)e^{-j2\pi nk/N}. \qquad (14\text{–}A2)$$

Substituting $p = n + 1$, so the range of $p$ is 1 to $N$, and we have:

$$X(k, q+1) = \sum_{p=1}^{N} x(p+q)e^{-j2\pi(p-1)k/N}. \tag{14–A3}$$

Next we change the summation by formally expressing the $N$th component separately and adding the $p = 0$ case to the summation, and then subtracting it formally:

$$X(k, q+1) = \sum_{p=0}^{N-1} x(p+q)e^{-j2\pi(p-1)k/N} + x(q+N)e^{-j2\pi(N-1)k/N} - x(q)e^{j2\pi k/N}. \tag{14–A4}$$

The exponential terms can be factored as follows:

$$X(k, q+1) = e^{j2\pi k/N} \left[ \sum_{p=0}^{N-1} x(p+q)e^{-j2\pi pk/N} + x(q+N)e^{-j2\pi Nk/N} - x(q) \right]. \tag{14–A5}$$

Because $e^{-j2\pi Nk/N} = 1$, (14–A5) can be rewritten as:

$$X(k, q+1) = e^{j2\pi k/N} \left[ \sum_{p=0}^{N-1} x(p+q)e^{-j2\pi pk/N} + x(q+N) - x(q). \right] \tag{14–A6}$$

Notice that the summation is the DFT of the $q$th time window, but (14–A6) is the DFT of the $(q+1)$th window. The sliding DFT can therefore be expressed as:

$$X(k, q+1) = e^{j2\pi k/N} [X(k, q) + x(q+N) - x(q)]. \tag{14–A7}$$

The individual, $k$th, frequency bins for the $(q+1)$th SDFT window can therefore be computed from the $q$th window bins and the time-domain inputs by:

$$S_k(n) = e^{j2\pi k/N} [S_k(n-1) + x(n) - x(n-N)]. \tag{14–A8}$$

The above derivation provides exact equivalence to the traditional DFT computation with a recursive algorithm that reduces the computational workload for successive sliding windows.

# Fast Function Approximation Algorithms

# Chapter 15

# Another Contender in the Arctangent Race

**Richard Lyons**

*Besser Associates*

**T**his chapter presents a computationally efficient algorithm for approximating the angle of a complex sample value. Estimating the angle θ of an $x = I + jQ$ complex sample has many applications in the field of signal processing. (The angle of $x$ is defined as $\theta = \tan^{-1}[Q/I]$).

## 15.1 COMPUTING ARCTANGENTS

Signal processing engineers interested in high-speed (minimum computations) arctangent computations typically use lookup tables where the values $I$ and $Q$ specify an address in read-only memory (ROM) containing an approximation of angle θ. The lookup table method is fast but requires much memory to provide acceptable arctangent accuracy. Those interested in high accuracy implement compute-intensive high-order algebraic polynomials to approximate angle θ, such as the

$$\tan^{-1}(Q/I) \approx \theta' = \frac{(Q/I) + 0.372003(Q/I)^3}{1 + 0.703384(Q/I)^2 + 0.043562(Q/I)^4} \, \text{radians} \quad (15\text{–}1)$$

algorithm, whose maximum error is on the order of 0.003 degrees when $|\theta| \le \pi/4$ radians [1].

Unfortunately, because it is such a nonlinear function, the arctangent is resistant to accurate small-length polynomial approximations. So we end up choosing the *least undesirable* method for computing arctangents: fast but inaccurate table lookup methods, or accurate but computationally intensive polynomial approximations.
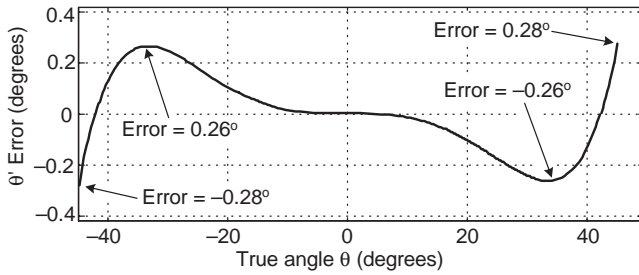
**Figure 15-1**   Error of estimated angle $\theta'$.

Here we present another contender in the arctangent approximation race that uses neither lookup tables nor high-order polynomials. We can estimate the angle $\theta$, in radians, of $x = I + jQ$ using the following approximation

$$\tan^{-1}(Q/I) \approx \theta' = \frac{(Q/I)}{1 + 0.28125(Q/I)^2} \text{ radians} \tag{15-2}$$

where $-1 \le Q/I \le 1$. That is, $\theta$ is in the range $-45°$ to $+45°$ ($-\pi/4 \le \theta \le +\pi/4$ radians). Equation (15–2) has surprisingly good performance, particularly for a $90°$ ($\pi/2$ radians) angle range. Figure 15-1 shows that the maximum error in $\theta'$ is $0.28°$ using (15–2) when the true angle $\theta$ is within the angular range of $-45°$ to $+45°$.

A nice feature of this $\theta'$ computation is that it can be written as:

$$\theta' = \frac{IQ}{I^2 + 0.28125Q^2} \text{ radians} \tag{15-3}$$

eliminating (15–2)'s $Q/I$ division operation, at the expense of two additional multiplies. Another attribute of (15–3) is that a single multiply can be eliminated with binary right-shifts. The product $0.28125Q^2$ is equal to $(2/4 + 1/32)Q^2$, so we can implement the product by adding $Q^2$ shifted right by two bits to $Q^2$ shifted right by five bits.

We can extend the angle range over which our approximation operates. If we partition a circle into eight $45°$ octants, with the first octant being $0°$–$45°$, we can compute the arctangent of a complex number residing in any octant. We do this by using the rotational symmetry properties of the arctangent:

$$\tan^{-1}(-Q/I) = -\tan^{-1}(Q/I) \tag{15-4}$$
$$\tan^{-1}(Q/I) = \pi/2 - \tan^{-1}(I/Q). \tag{15-4'}$$

Those properties allow us to create Table 15-1, listing the appropriate arctan approximation based on the octant location of complex $x$. The maximum angle approximation error is $0.28°$ for all octants. So we have to check the signs of $Q$ and $I$, and compare magnitudes $|Q|$ and $|I|$, to determine the original octant location of $\theta$, and then use the appropriate approximation in Table 15-1.

With that thought in mind, Table 15-2 shows how to identify the original octant of $\theta$ based on the values of components $I$ and $Q$.

**Table 15-1**    Arctangent Expressions versus Octant
Location

| Octant | Arctan approximation (radians) |
|---|---|
| 1st, or 8th | $\theta' = \dfrac{IQ}{I^2 + 0.28125Q^2}$ |
| 2nd, or 3rd | $\theta' = \dfrac{\pi}{2} - \dfrac{IQ}{Q^2 + 0.28125I^2}$ |
| 4th, or 5th | $\theta' = \text{sign}(Q)\pi + \dfrac{IQ}{I^2 + 0.28125Q^2}$ |
| 6th, or 7th | $\theta' = -\dfrac{\pi}{2} - \dfrac{IQ}{Q^2 + 0.28125I^2}$ |

**Table 15-2**    Identification of Octant

| Sign($I$) | Sign($Q$) | $\lvert Q\rvert - \lvert I\rvert$ | Octant |
|---|---|---|---|
| + | + | $\lvert Q\rvert - \lvert I\rvert < 0$ | 1st |
| + | + | $\lvert Q\rvert - \lvert I\rvert \geq 0$ | 2nd |
| – | + | $\lvert Q\rvert - \lvert I\rvert \geq 0$ | 3rd |
| – | + | $\lvert Q\rvert - \lvert I\rvert < 0$ | 4th |
| – | – | $\lvert Q\rvert - \lvert I\rvert < 0$ | 5th |
| – | – | $\lvert Q\rvert - \lvert I\rvert \geq 0$ | 6th |
| + | – | $\lvert Q\rvert - \lvert I\rvert \geq 0$ | 7th |
| + | – | $\lvert Q\rvert - \lvert I\rvert < 0$ | 8th |

## 15.2  CONCLUSIONS

This arctangent algorithm may be useful in a digital receiver application where $I^2$ and $Q^2$ have been previously computed in conjunction with an AM (amplitude modulation) demodulation process or envelope detection associated with automatic gain control (AGC). Although this chapter focused on estimating the angle of an $x = I + jQ$ complex sample value, we remind the reader that (15–2) can be viewed as a generic arctangent approximation represented by

$$\tan^{-1}(x) \approx \frac{x}{1 + 0.28125x^2} \text{ radians} \tag{15–5}$$

where $-1 \leq x \leq 1$. Such a generic viewpoint is discussed in Chapter 18.

## 15.3  REFERENCE

[1] C. ZAROWSKI, "Differential Evolution for a Better Approximation to the Arctangent Function," Nanodottek Report NDT3-04-2006, 26 April 2006.

# Chapter 16

# High-Speed Square Root Algorithms

**Mark Allie**
*University of Wisconsin—Madison*

**Richard Lyons**
*Besser Associates*

In this chapter we discuss several useful tricks for estimating the square root of a number. Our focus will be on high-speed (minimum computations) techniques for approximating the square root of a single value as well as the square root of a sum of squares for quadrature (I/Q) vector magnitude estimation.

In the world of DSP, computing a square root is found in many applications, such as calculating root mean squares, computing the magnitudes of fast Fourier transform (FFT) results, implementing automatic gain control (AGC) techniques, estimating the instantaneous envelope of a signal (AM demodulation) in digital receivers, and implementing three-dimensional graphics algorithms. The fundamental trade-off to be made in choosing a particular square root algorithm is execution speed versus algorithm accuracy. Here we disregard the advice of a legendary lawman ("Fast is important, but accuracy is everything"—Wyatt Earp [1848–1929]), and concentrate on various high-speed square root approximations. In particular, we focus on algorithms that can be implemented efficiently in fixed-point arithmetic. Other constraints we've set are: No divisions are allowed; only a small number of computational iterations are permitted; and only a small lookup table, if needed, is allowed.

The first two methods below describe ways to estimate the square root of a single value using iterative methods. The last two techniques describe methods for estimating the magnitude of a complex number.

## 16.1   NEWTON-RAPHSON INVERSE (NRI) METHOD

A venerable technique for computing the square root of $x$ is to use the *Newton-Raphson square root* iterative technique to find $y(n)$, the approximation of

$$\text{sqrt}(x) \approx y(n+1) = [y(n) + x/y(n)]/2. \qquad (16\text{--}1)$$

Variable $n$ is the iteration index, and $y(0)$ is an initial guess of sqrt($x$) used to compute $y(1)$. Successive iterations of (16–1) yield more accurate approximations to sqrt($x$) because the number "bits of accuracy" doubles for each iteration.

However, to avoid that computationally expensive division by $y(n)$ in (16–1) we can use the iterative *Newton-Raphson inverse* (NRI) method, by first finding the approximation to the inverse square root of $x$ using (16–2), where $p = 1/\text{sqrt}(x)$. Next we compute the square root of $x$ by multiplying the final $p$ by $x$,

$$p(n+1) = 0.5p(n)[3 - xp(n)^2]. \qquad (16\text{--}2)$$

Two iterations of (16–2) provide surprisingly good accuracy. As suggested by Cordesses and Araujo [1], the initial $p(0)$ value used in the first iteration is

$$p(0) = 1.63841001x^2 - 3.28504011x + 2.67057805. \qquad (16\text{--}3)$$

Equation (16–3) can be evaluated using Horner's Rule, requiring only two multiplies and two adds as

$$p(0) = (1.63841001x - 3.28504011)x + 2.67057805. \qquad (16\text{--}3')$$

The square root function looks more linear when we restrict the range of our $x$ input. As it turns out, it's convenient to limit the range of $x$ to $0.25 \leq x < 1$. So if $x < 0.25$, then it must be *normalized*, and fortunately we have a slick way to do so. When $x$ needs normalization, then $x$ is multiplied by $4^n$ until $0.25 \leq x < 1$. A factor of 4 is 2 bits of arithmetic left-shift. The final square root result is *denormalized* by a factor of $2^n$ (the square root of $4^n$). A denormalizing factor of 2 is a single arithmetic right-shift. After implementing this normalization, to ensure $0.25 \leq x < 1$, the error curve for this two-iteration NRI square root method is shown in Figure 16-2, where we see the maximum error is roughly 0.00011%. The curve is a plot of the error in our estimated square root divided by the true square root of $x$. That maximum error value is impressively small—almost worth writing home about.

This NRI method is not recommended for fixed-point format with its sign and fractional bits. The coefficients, $p(n)$, and intermediate values in (16–2) are greater than one. In fixed-point math, using bits to represent internal results greater than one increases the error by a factor of two per bit. (Certainly using more bits for the integer part of intermediate values without taking them away from the fractional part can be done, but only at the cost of more CPU cycles and memory.)

## 16.2   NONLINEAR IIR FILTER (NIIRF) METHOD

The next iterative technique, by Mikami et al. [2], specifically aimed at fixed-point implementation, is configured as a nonlinear IIR filter (NIIRF) as depicted in
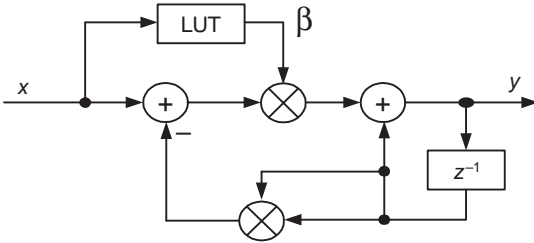
**Figure 16-1**    NIIRF implementation.

**Table 16-1**    β versus Normalized-*x* LUT

| 4 MSBs of *x* | β fixed-point | β flt-point |
|---|---|---|
| 0100 | 0x7b20 | 0.961914 |
| 0101 | 0x6b90 | 0.840332 |
| 0110 | 0x6430 | 0.782715 |
| 0111 | 0x5e10 | 0.734869 |
| 1000 | 0x5880 | 0.691406 |
| 1001 | 0x53c0 | 0.654297 |
| 1010 | 0x4fa0 | 0.622070 |
| 1011 | 0x4c30 | 0.595215 |
| 1100 | 0x4970 | 0.573731 |
| 1101 | 0x4730 | 0.556152 |
| 1110 | 0x4210 | 0.516113 |
| 1111 | 0x4060 | 0.502930 |

Figure 16-1, where again input *x* has been normalized to the range $0.25 \leq x < 1$. The output is given by (16–4) where the constant β, called the iteration *acceleration* factor, depends on the value of input *x*,

$$y(n+1) = \beta[x - y(n)^2] + y(n). \tag{16–4}$$

The magnitude of the error of (16–4) is known in advance when $\beta = 1$. The β acceleration factor scales the square root update term, $x - y(n)^2$, so the new estimate of y has less error. The acceleration factor results in reduced error for a given number of iterations.

In its standard form, this technique uses a lookup table (LUT) to provide β and performs two iterations of (16–4). For the first iteration $y(0)$ is initialized using

$$y(0) = 2x/3 + 0.354167. \tag{16–5}$$

The constants in (16–5) were determined empirically. The acceleration constant β, based on *x*, is stored in a LUT made up of 12 subintervals of *x* with an equal width of 1/16. This square root method is implemented with the range of *x* set between 4/16 and 16/16 (12 regions) as was done for the NRI method. This convenient interval directly gives the offset into the lookup table when using the four most significant mantissa bits of the normalized *x* value as a pointer, as shown in Table 16-1.
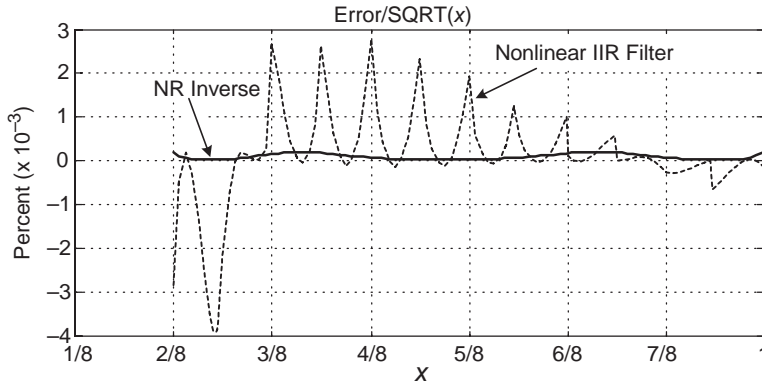
**Figure 16-2**    Normalized error for the NRI and NIIRF methods.

Figure 16-2 shows the *normalized* NIIRF error curve as a function of *x*. (Normalized, in this instance, means the error of the estimated square root divided by the true square root of *x*.) The NRI square root method is the most accurate, in floating-point math, followed by the NIIRF method. This is not surprising considering the greater number of operations needed to implement the NRI method.

The effect of the lookup table for β is clearly seen in Figure 16-2 where the dotted curve shows discontinuities (spikes) at the table lookup boundaries. These are, of course, missing from the NRI error curve.

One sensible thing to do, when presented with algorithms such as the NIIRF square root method, is to experiment with variations of the algorithm to investigate accuracy versus computational cost. (We did that for the above NRI method by determining the maximum error when only one iteration of (16–2) was performed). With this *explore and investigate* thought in mind, we examined seven variations of this NIIRF algorithm. The first variation, a simplification, is the original NIIRF algorithm using only 1 iteration. We then studied the following quadratic function of *x* to find β

$$\beta = 0.763x^2 - 1.5688x + 1.314. \tag{16–6}$$

Next we used a linear function of *x* to find β, defined by

$$\beta = -0.61951x + 1.0688. \tag{16–7}$$

For the quadratic and linear variations to compute β, we investigated their performances when using both one and two iterations of (16–4).

For the final NIIRF method variation, we set β equal to the constants 0.633 and 0.64 for two- and one-iteration variations respectively. Table 16-2 provides a comparison of the error magnitude behavior of the original two-iteration (LUT) NIIRF algorithm and all the above variations. For all of the variations listed in Table 16-2, the range of the input, *x*, was limited to $0.25 \leq x < 1$. (The term *normalized* as used

**Table 16-2**    Iterative Algorithm Normalized Absolute Error

| Algorithm | Max. normalized error (%) | Mean normalized error (%) |
|---|---|---|
| **NR inverse (NRI)** | | |
| NRI, 2 iters | 1.1e−4 | 4.2e−5 |
| **NIIRF floating-pt** | | |
| LUT β, 2 iters | 0.004 | 5.4e−4 |
| LUT β, 1 iter | 0.099 | 0.026 |
| Quad. β, 2 iters | 0.0013 | 2.8e−4 |
| Quad. β, 1 iter | 0.056 | 0.019 |
| Linear β, 2 iters | 0.024 | 0.0061 |
| Linear β, 1 iter | 0.28 | 0.088 |
| β=0.633, 2 iters | 0.53 | 0.05 |
| β=0.64, 1 iter | 1.44 | 0.23 |
| **NIIRF fixed-pt** | | |
| NIIRF, 2 iters | 0.0035 | 5.1e−4 |
| Quad. β, 2 iters | 0.0019 | 4.1e−4 |
| Linear β, 2 iters | 0.011 | 0.0029 |

in Table 16-2 means the error of the estimated square root divided by the true square root of $x$.)

The error data in Table 16-2 was generated using double-precision floating-point math. In comparing the accuracy and complexity of the NRI method to the NIIRF, we might ask why one ever uses the NIIRF method. Often, double-precision floating-point math is not available to us. Fixed-point data is one of the assumptions of this chapter. This fixed-point constraint requires that the algorithms must be reevaluated for the error when fixed-point math is used. It turns out that this is why Mikami et al., developed the NIIRF square root method in the first place [2]. In fixed-point math most of the internal values are close to each other and less than one, so these values result in the lower error in the NIIRF method compared to the NRI method when both are implemented in fixed-point format. In fact, there is less error in the fixed-point compared to the floating-point implementations of NIIRF method. All of the NIIRF variations are well suited to fixed-point math. Next, we look at high-speed square root approximations used to estimate the magnitude of a complex number.

## 16.3  BINARY-SHIFT MAGNITUDE ESTIMATION

When we want to compute the magnitude $M$ of the complex number $I+jQ$ the exact solution is, of course,

$$M = \text{sqrt}(I^2 + Q^2). \tag{16–8}$$
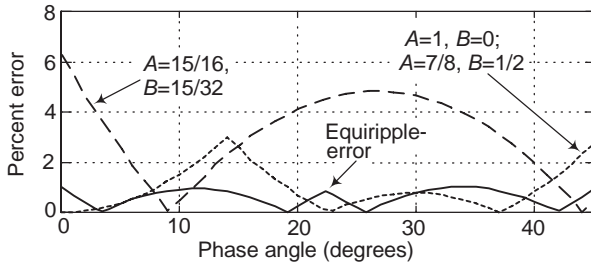
**Figure 16-3**    Error of *binary-shift* and *equiripple-error* methods.

To approximate the square root operation in (16–8), the following *binary-shift magnitude estimation* algorithm can be used to estimate $M$ using

$$M \approx A\mathrm{M_{ax}} + B\mathrm{M_{in}} \qquad (16–9)$$

where $A$ and $B$ are constants, and $\mathrm{M_{ax}}$ is the maximum of either $|I|$ or $|Q|$ while $\mathrm{M_{in}}$ is the minimum of $|I|$ or $|Q|$.

Many combinations of $A$ and $B$ are provided in [3] yielding various accuracies for estimating $M$. (Chapter 12 discusses the use of $A=1$ and $B=4$ yielding estimates of magnitude $M$ with a maximum error of 11.6% and a mean error of 3.2%.) However, of special interest is the combination $A=15/16$ and $B=15/32$ using

$$M \approx 15\mathrm{M_{ax}}/16 + 15\mathrm{M_{in}}/32. \qquad (16–10)$$

This algorithm's appeal is that it requires no explicit multiplies because the $A$ and $B$ values are binary fractions and (16–10) is implemented with simple binary right-shifts, additions, and subtractions. (For example, a 15/32 times $z$ multiply can be performed by first shifting $z$ right by 4 bits and subtracting that number from $z$ to obtain 15/16 times $z$. That result is then shifted right 1 bit.) This algorithm estimates the magnitude $M$ with a maximum error of 6.2% and a mean error of 3.1%.

The percent error of this binary-shift magnitude estimation scheme is shown as the dashed curve in Figure 16-3 as a function of the angle between $I$ and $Q$. (The curves in Figure 16-3 repeat every 45°.)

At the expense of a compare operation, we can improve on the accuracy of (16–10) [4]. If $\mathrm{M_{in}} \leq \mathrm{M_{ax}}/4$, we use the coefficients $A=1$ and $B=0$ to compute (16–9); otherwise, if $\mathrm{M_{in}} > \mathrm{M_{ax}}/4$, we use $A=7/8$ and $B=1/2$. This dual-coefficients (and still multiplier-free) version of the binary-shift square root algorithm has a maximum error of 3.0% and a mean error of 0.95%, as shown by the dotted curve in Figure 16-3.

## 16.4    EQUIRIPPLE-ERROR MAGNITUDE ESTIMATION

Another noteworthy scheme for computing the magnitude of the complex number $I+jQ$ is the *equiripple-error magnitude estimation* method by Filip [5]. This technique, whose maximum error is roughly 1%, is sweet in its simplicity: If $\mathrm{M_{in}} \leq 0.4142135\mathrm{M_{ax}}$, the complex number's magnitude is estimated using

$$M \approx 0.99 M_{ax} + 0.197 M_{in}. \qquad (16\text{–}11)$$

However, if $M_{in} > 0.4142135 M_{ax}$, the following is used to estimate $M$:

$$M \approx 0.84 M_{ax} + 0.561 M_{in}. \qquad (16\text{–}12)$$

This algorithm is so named because its maximum error is 1.0% for both (16–11) and (16–12). Its mean error is 0.6%. Because its coefficients are not simple binary fractions, this method is best suited for implementations on programmable hardware. The percent error of this equiripple-error magnitude estimation method is also shown in Figure 16-3, where we see the more computationally intensive method, equiripple-error, is more accurate. As usual, accuracy comes at the cost of computations. Although these methods are less accurate than the iterative square root techniques, they can be useful in those applications where high accuracy is not needed, such as when $M$ is used to scale (control the amplitude of) a system's input signal.

One implementation issue to keep in mind when using integer arithmetic is that even though values $|I|$ and $|Q|$ may be within your binary word-width range, the estimated magnitude value may be larger than can be contained within the numeric range. The practitioner must limit $|I|$ and $|Q|$ in some way to ensure that the estimated $M$ value does not cause overflows.

## 16.5   CONCLUSIONS

We discussed several square root and complex vector magnitude approximation algorithms with a focus on high-throughput (high-speed) algorithms as opposed to high-accuracy methods. We investigated the performance of several variations of the iterative NRI and NIIRF square root methods and found, not surprisingly, that the number of iterations has the most profound effect on accuracy. The NRI method is not appropriate for implementation using fixed-point fractional binary arithmetic, while the NIIRF technique and the two magnitude estimation schemes lend themselves nicely to fixed-point implementation. The three magnitude estimation methods are less accurate but more computationally efficient than the NRI and NIIRF schemes. All algorithms described here are open to modification and experimentation, which will make them either more accurate and computationally expensive or less accurate and computationally cheap. Your accuracy and data throughput needs will determine the path you take.

## 16.6   REFERENCES

[1]  L. Cordesses and M. Araujo, Private communication, November 28, 2006.
[2]  N. Mikami et al., "A New DSP-Oriented Algorithm for Calculation of Square Root Using a Nonlinear Digital Filter," *IEEE Trans. on Signal Processing*, July 1992, pp. 1663–1669.
[3]  R. Lyons, *Understanding Digital Signal Processing*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2004, pp. 481–482.
[4]  W. Adams and J. Brady, "Magnitude Approximations for Microprocessor Implementation," *IEEE Micro*, October 1983, pp. 27–31.

[5] A. Filip, "Linear Approximations to sqrt($x^2+y^2$) Having Equiripple Error Characteristics," *IEEE Trans. on Audio and Electroacoustics*, December 1973, pp. 554–556.

## EDITOR COMMENTS

The above material discusses two high-speed algorithms for approximating the square root function. While the first algorithm (the NRI method) is of interest when used in floating-point systems, the second algorithm (the NIIRF method) is most useful in fixed-point fractional binary arithmetic. After first learning of this second algorithm, the authors did what all inquisitive DSP practitioners should do—they experimented with variations of the algorithm to compare computational workload versus algorithm accuracy.

To expand a bit on the NRI square root method, recall that the argument $x$ was limited to the range $0.25 \leq x < 1$. That limitation can be relaxed, such that $x$ need only be limited to be a positive number, if we store the appropriate $p(0)$ values in a lookup table (LUT). The $p(0)$ value retrieved from the LUT is based on the original value of $x$. An example of this implementation is described in Analog Devices Inc.'s "ADSP-21000 Family Application Handbook, vol. 1." In that handbook an NRI square root algorithm is described where seven bits from the floating-point binary representation of $x$ are extracted and used as address bits to access a read-only memory (ROM) containing initial inverse square root values, $p(0)$, accurate to 4 bits. In their NRI square root implementation, three iterations of (16–2) yields results accurate to 32 bits (32-bit mantissa).

Texas Instruments' document, "TMS320C4x General-Purpose Applications, User's Manual," describes their table lookup NRI square root method where the initial $p(0)$ value is accurate to 8 bits. So they achieve 32 bits of accuracy after only two iterations of (16–2).

# Chapter 17

# Function Approximation Using Polynomials

**Jyri Ylöstalo**

*Nokia Siemens Networks*

**D**SP designers sometimes encounter a situation where they have to use some of the elementary functions (logarithm, square root, exponential, trigonometric functions, etc.) on a fixed-point processor. Although optimized math libraries are widely available for floating-point processors, this is not necessarily true for the fixed-point case, which means that designers must implement the elementary functions using *home-grown* algorithms. This chapter describes a powerful technique for designing algorithms for such implementations, describes range-reduction methods to make the algorithms more accurate, and provides useful tips on using the proposed techniques in floating-point and fixed-point number systems.

In what follows we show how ordinary polynomials can be used for approximating the values of functions (logarithms, square root, trigonometric functions, etc.) of a real variable. The polynomial approximations discussed here are an attractive alternative for this purpose for several reasons:

- They require only multiplication and addition, which means that the calculation is fast if a hardware multiplier is available, as is the case with commercial DSP chips.

- The computation is noniterative, which makes efficient parallelization possible.

- They can be very accurate, and the speed-versus-accuracy-versus-memory-use trade-off can be tailored for the requirements of the application. For

example, you can create a function approximation that is fast and accurate and uses a lot of memory, or one that is slow and accurate and does not need much memory, or yet another one that is fast and inaccurate and uses very little memory.

- The same unified approach can be used for all continuous functions.

Approximation theory in itself is a rather broad subject and several extensive treatments have been written on it [1]–[3]. Here we provide a concise, practical presentation (no proofs for the theorems!) enabling the reader to create approximation polynomials for their own purposes and to utilize them sensibly. The approach presented here is a simple, basic one—those who want to delve really deeply into elementary function implementation can study, for example, [4], [5].

## 17.1   USING LAGRANGE INTERPOLATION

Polynomial approximation is based on the classical Weierstrass theorem, according to which for any continuous function $f(x)$ defined within a closed range interval $[a,b]$ there exists a polynomial $p(x)$ for approximating the function in that range so that the maximum approximation error is kept below a given limit. The theorem doesn't say anything about how such a polynomial can be found. Usually the most straight-forward method is to calculate the value of $f(x)$ for $n+1$ distinct *fitting points* within the interval range $a \le x \le b$ and to satisfy the simultaneous equations $p(x_i)=f(x_i)$, $i=0$, $1, \ldots, n$. There is always a unique polynomial $p(x)$ of degree $n$ that satisfies these conditions and it is given by the *Lagrange interpolation formula*:

$$p(x) = \sum_{i=0}^{n} p_i(x) \tag{17-1}$$

where each $p_i(x)$ term is defined by

$$p_i(x) = f(x_i) \prod_{j=0, j \neq i}^{n} \frac{x - x_j}{x_i - x_j}. \tag{17-1'}$$

For example, let's try to fit a second-degree polynomial to the fitting points $[x_i, f(x_i)] = (1,1), (2,3), (4,5)$. According to the above formulas, $n=2$ and we get

$$p(x) = 1\left[\frac{x-2}{1-2} \cdot \frac{x-4}{1-4}\right] + 3\left[\frac{x-1}{2-1} \cdot \frac{x-4}{2-4}\right] + 5\left[\frac{x-1}{4-1} \cdot \frac{x-2}{4-2}\right] \tag{17-2}$$

$$= -x^2/3 + 3x - 5/3.$$

## 17.2   FINDING THE OPTIMAL APPROXIMATION POLYNOMIAL

In principle, we could now improve the accuracy of the approximation as much as we want simply by adding new polynomial fitting points to the interval. For every
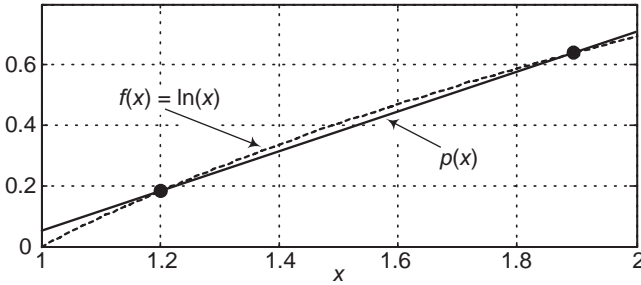
**Figure 17-1**  Nonoptimal, first-degree (linear) approximation of $\ln(x)$ with $x_0 = 1.2$ and $x_1 = 1.9$.
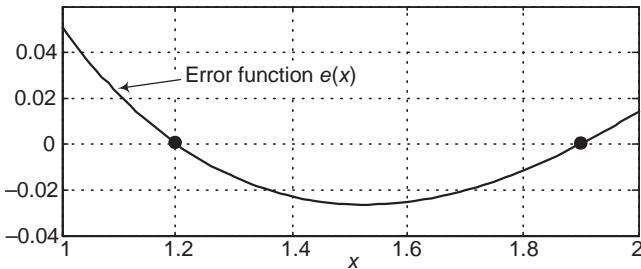


**Figure 17-2**  Error function of the nonoptimal, first-degree (linear) approximation of $\ln(x)$.

new fitting point, the degree of the approximation polynomial would increase and calculation of the value of $p(x)$ would slow down correspondingly. Also, the amount of memory required for storing the parameters would increase. Instead of this kind of brute force approach, however, we could ask, "What is the optimal approximation, that is, the polynomial of a given degree that minimizes the maximum approximation error within the interval?" (Other definitions of *optimal approximation* are possible, but we won't deal with them here). It turns out that there is no straightforward mapping from the function being approximated to the optimal polynomial. We must instead use some kind of iterative algorithm to find our optimal approximating polynomial.

Consider the case of the first-degree polynomial $p(x)$ (the solid straight line in Figure 17-1), which attempts to approximate the function $f(x) = \ln(x)$ in the $x$ interval $[a,b] = [1,2]$ using fitting points $[x_0 = 1.2, \ln(1.2)]$ and $[x_1 = 1.8, \ln(1.8)]$, which yield $p(x) = 0.6565x - 0.6054$. The approximation error $e(x) = f(x) - p(x)$ is depicted in Figure 17-2.

This approximation is in fact not optimal because, as we shall see, better approximations are available. We can perform an exhaustive search (e.g., using a 0.01 step size between new fitting points) using all possible pairs of fitting points on $f(x)$ in the interval. In doing so we find out that in the optimal case, the extrema of the error function have the same absolute value (0.03) and alternating signs as indicated in Figure 17-3. (It's because the maximum error in Figure 17-2 is greater than 0.03 that we said the first-degree polynomial approximation of $\ln(x)$ is not
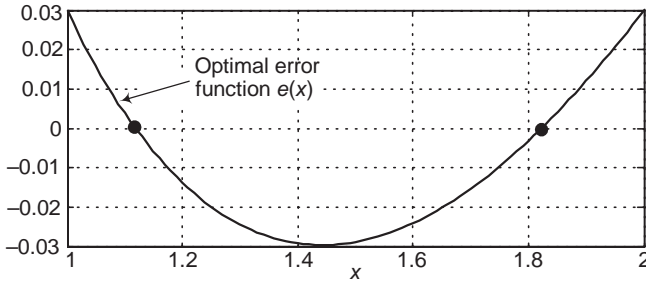
**Figure 17-3** The optimal fit is reached with fitting points $x_0 = 1.12$ and $x_1 = 1.82$.

optimal.) *Equal absolute error extrema values* is an important and necessary property of *all* optimal (in the minimax sense) polynomial approximations of any degree for any function in any closed interval. This is called the *Chebyshev characterization theorem* and its first version was presented by Pafnuty Chebyshev in 1859.

Exhaustive search is hardly a viable method for obtaining optimal polynomials of any degree higher than one. We could therefore try to use the above "equal absolute values of error extrema" property directly and move the fitting points gradually so that the absolute values of the extrema of the resulting error function would become more equal. Because the error is always zero at the fitting point, it seems clear that if we move a fitting point in the direction of an error peak, the height of the peak will decrease. Thus it might be a good idea to move each of the fitting points in the direction of the higher of its neighboring error peaks. Doing so leads us toward the optimal situation, where all of the error peaks have equal absolute values.

We now propose the following iterative algorithm for finding the optimal polynomial $p(x)$ for approximating the continuous function $f(x)$ in the interval $[a,b]$. This proposed process is not the fastest one available, but it is very simple and illustrative and in any case the iterations compute in mere seconds. Some hardcore mathematician might say to you that we should use the Remez exchange algorithm. It is admittedly a lot more efficient (2 to 6 iterations instead of 20 to 30), and there is very solid theory behind it as to why, when, and how it works. With our proposed technique you launch the routine using, say, MathCAD or MATLAB, go get yourself a cup of coffee, and upon returning you have parameters you can use for the rest of your life.

1. Choose approximation interval $[a,b]$, and the Degree of the approximating polynomial, $D$. The number of fitting points needed will be $D+1$. The points are denoted as $x_0, f(x_0), \ldots, x_D, f(x_D)$.

2. Define initial iteration Step Size $s$, Decrease Factor $d$, and number of iterations $n$. Suitable values can be found by trial and error; however, $s = D/10$, $d = 0.9$, $n = 10D$ seem to work well.

3. Place a fitting point at both ends of the approximation interval (i.e., $x_0 = a$, $x_D = b$). These are the *terminal points* and they don't move during the itera-

tion. Place the remaining $D-1$ points (the *intermediate fitting points*) at suitable places in the interval (the initial positions are not very important, as long as two points don't have the same coordinates). Equidistantly scattered points work okay.

4. Fit a polynomial (Lagrange interpolation) to the $D+1$ fitting points using (17–1) and (17–2), yielding the current $p(x)$.

5. For each of the intermediate fitting points $[x_i, f(x_i)]$, $i=1 \ldots D-1$, find the local error peak $|e(x_{ip})|$ in the interval defined by $[x_{i-1}, x_{i+1}]$, and compute the peak's $x$-position, $x_{ip}$.

6. Shift each $x_i$ intermediate point by moving it in the direction of its interval's local error peak located at $x_{ip}$, using $x_{inew}=x_i+s(x_{ip}-x_i)$.

7. Define the next iteration Step Size: $s_{new}=s_d$.

8. Return to step 4 until steps 4 through 8 have been performed $n$ times.

(A feature of this algorithm is that you can easily fix points you don't want to move.)

The reason why we need two fixed terminal points at both ends of the interval is to assure continuity of the resulting approximated function. This becomes important when *piecewise polynomials* are used for approximation, as we will see later. Another good reason to fix the terminal points is to minimize error for critical values. For example, it might be embarrassing, or even devastating to your application, to have your logarithm approximation claim that log(1)=0.0004 instead of 0. But if neither of these concerns, continuity or critical values, applies to you, then by all means move the terminal points in the same way as all the other points. In this way you will get tiny discontinuities, but slightly smaller error.

## 17.3   RANGE REDUCTION

Although in theory it is possible to approximate any continuous function over any closed interval with a polynomial, it hardly ever makes sense to try to fit one single polynomial to the entire possible input range. Such a polynomial might need to be of very high order to meet the desired accuracy specifications. In general, the shorter the approximation interval the closer to linear the function is in that interval, and the easier our job will be—the lower the degree of the polynomial required. We should therefore devise a suitable range-reduction scheme, in order to be able to compute all possible output values from an approximation performed within a short, closed, interval [4].

Consider the case of applying this range-reduction notion to the logarithm (of any desired base $k$). From the *laws of exponents*, using $\log_k(y^n x)=n\cdot\log_k(y)+\log_k(x)$, we can write:

$$\log_k (x) = \log_k (2^n \cdot x) - n \cdot \log_k (2). \tag{17–3}$$

Equation (17–3) tells us that to compute $\log_k(x)$, where $x$ may cover a wide range of values, we can *normalize* $x$ (reduce its range) to a smaller range of values for

which a more accurate approximation polynomial can be found. Once we compute $\log_k(2n\cdot x)$, we then *denormalize* that computation result by subtracting $n\cdot\log_k(2)$ to arrive at the desired $\log_k(x)$ value. In practice this means that we must decide on some integer $M$ and develop the approximation polynomial for the interval $[2^{M-1},2^M]$. It is always possible to find a suitable $n$ to normalize $2^n x$ between $[2^{M-1},2^M]$ and then use (17–3) to calculate $\log_k(x)$.

In the floating-point case, the normalization can be done simply by adjusting the exponent part of the number $x$. In the fixed-point case, let us define the operation HighestBit($x$), for any integer $x>0$, as the index of the most significant 1 bit of the number $x$. (The index of the MSB of an unsigned 16-bit number is 15; the index of the LSB is 0.) Luckily, some programmable DSP chips have a machine instruction for performing this kind of an operation in a single cycle, which of course speeds up calculation enormously. (For example, Texas Instruments' DSP chips have the instruction LMBD—*leftmost bit detection*.) Now, the required power of two for the normalization can be found with

$$n = M - 1 - \text{HighestBit}(x). \qquad (17\text{–}4)$$

We can define $M=0$, in which case the normalized logarithm, $\log_k(2^n\cdot x)$, is calculated for $2^n\cdot x$, which will be in the interval $[0.5,1]$ and can be represented as a Q-15 number. For example, if we want to compute $\log(x)$ where $x=1234$, we get HighestBit($x$)$=10$ and $n=-11$. The $x=1234$ input to the approximation will thus be converted to $1234\cdot 2^{-11}=0.6025390625$, which is in the range $[0.5,1]$.

A rather similar kind of range reduction scheme can be used for square roots as

$$\sqrt{x} = \sqrt{2^n \cdot x}(\sqrt{2})^{-n} \qquad (17\text{–}5)$$

where $x$ is normalized to $x_{norm}=2^n\cdot x$, with $n$ being defined by (17–4). Note that the *denormalization* by $[\text{sqrt}(2)]^{-n}$ can be realized in the fixed-point case with shifting by $n/2$ in the case of $n$ even. A possible approach is to perform the calculation in the interval $[0.25,1]$. We can first calculate $n$ just as in (17–4), and then, if $n$ is odd, decrement it by one. For example, say we want to compute the square root of $x=1234$. First we determine $n=-11$, which is an odd number so we decrement it to $n=-12$. Next we normalize $x=1234$ to $x_{norm}=1234\cdot 2^{-12}=0.30126953125$, which is in the range $[0.25,1]$, upon which we compute the square root. The final step is to *denormalize* the square root result by multiplying it by $[\text{sqrt}(2)]^{-n}$.

In the case of polynomial approximations of trigonometric functions, we need an entirely different type of range reduction: additive instead of multiplicative. A suitable reduced range is $[0,\pi/4]$. We first normalize the input $x$ to be between $[-\pi/4,\pi/4]$: If $x$ is outside this range, we add or subtract $\pi/2$ until the normalized input, $x_{norm}$, falls into this range. We get $x_{norm}=x-m\pi/2$. If $x_{norm}$ is in the range $[-\pi/4,0]$, we use the familiar relations $\cos(x)=\cos(-x)$, $\sin(x)=-\sin(-x)$. Now

$$
\begin{aligned}
\cos(x) &= \cos(x_{norm}) && \text{if } m \bmod 4 = 0, \\
\cos(x) &= -\sin(x_{norm}) && \text{if } m \bmod 4 = 1, \\
\cos(x) &= -\cos(x_{norm}) && \text{if } m \bmod 4 = 2, \\
\cos(x) &= \sin(x_{norm}) && \text{if } m \bmod 4 = 3.
\end{aligned}
$$

For example, instead of calculating cos(9) (input in radians), we calculate $-\cos(6\pi/2-9)$.

## 17.4 SUBINTERVAL DIVISION

We don't need to stop our range-reduction efforts here. One possibility is to further divide the approximation interval into $S$ subintervals, which—let's assume for simplicity—are of equal width. For instance, we could divide the interval [0.5,1] into two subintervals, [0.5,0.75] and [0.75,1], or four subintervals, [0.5,0.625], [0.625, 0.75], [0.75,0.875], and [0.875,1]. Each of the subintervals has a separate approximation polynomial, which approximates a function $f(x)$ in that subinterval.

Subintervals are identified by their index $i$, which is defined in the following way:

$$i = \text{trunc}\left(\frac{S(x_{norm}-a)}{b-a}\right) \qquad (17\text{–}6)$$

where "trunc" stands for truncation of the decimal part and the input $x$ has been previously normalized to the interval $a \leq x_{norm} < b$. The index $i$ starts from zero like in the C language. The speed of calculation of the approximation result is independent of the value of $S$, and if the parameters ($S$ relative to $b-a$) are chosen smartly, division won't be needed and the correct subinterval for the normalized input $x$ can be found with a simple shift operation. We can thus, at least in theory, increase accuracy as much as we want without any speed penalty simply by dividing the approximation interval into smaller subintervals—only the required amount of memory for storing the polynomial coefficients will increase. If the degree of approximation is $D$ and the number of subintervals is $S$, we need $S(D+1)$ polynomial coefficients.

When trying to improve the accuracy of a polynomial approximation, we have two alternatives: Increase either $D$ or $S$ (i.e., use a higher degree of approximation or divide the approximation interval to smaller subintervals). Increasing $D$ means increased computation time and a slight increase in memory use, whereas increasing $S$ means a slightly faster increase in memory use *without any increase in computation time*. In modern systems, storage space is usually cheap while time isn't. Subinterval division is therefore an attractive option for improving accuracy. Another factor favoring increasing $S$ instead of $D$ is related to arithmetic problems using fixed-point calculations, as we shall soon see.

## 17.5 PRACTICAL CONSIDERATIONS

On a floating-point system, calculating polynomial approximation results is trivial once a suitable range-reduction scheme has been developed, the degree of approximation and subinterval division have been selected, and the polynomial coefficients have been calculated. Usually, in order to reach high accuracy it is preferable to use relatively high-degree polynomials, possibly taking advantage of available pipelining features and finding a suitable compromise between accuracy and speed.

With fixed-point systems, things aren't necessarily as simple as this. The use of high-degree polynomials may easily lead to disappointing results for the following reason: Suppose that we have a 32-bit DSP with 16-bit hardware multiplier. In such a case it is usually advisable to represent the coefficients using 16 bits and accumulate the sum of the polynomial terms to a 32-bit variable. When we multiply the input $x$ with itself, we have to discard half of the bits of the result in order to fit it into 16 bits. This introduces a quantization error, which becomes progressively larger in proportion to the result with higher powers of $x$. Therefore, on fixed-point systems the most viable alternative is usually to utilize low-degree (e.g. quadratic or cubic) polynomials and increase the number of subintervals, if the accuracy must be improved.

## 17.6  ERROR STUDIES

Controlling approximation error is easy with a proper choice of $S$ and $D$. On a fixed-point system the real problem is the quantization effects, which may spoil the outcome no matter how excellent the results look with a floating-point system like MATLAB. The most obvious solution to reduce quantization errors is to increase the multiplication wordlength. However, this will almost inevitably lead to increased execution time. It pays therefore to study the error generation a bit more precisely, in order to squeeze the best possible accuracy out of the fewest possible CPU cycles. Using the tricks described in this section, it should be possible to reduce the maximum approximation error to below 5 ppm.

For simplicity, let us concentrate on the case where we use $16 \times 16$-bit multiplication and 32-bit accumulation to calculate a second-degree polynomial:

$$y = ax^2 + bx + c. \tag{17–7}$$

Because of quantization effects, this becomes

$$y = (a + \varepsilon_a)[(x + \varepsilon_x)(x + \varepsilon_x) + \varepsilon_{x^2}] + (b + \varepsilon_b)(x + \varepsilon_x) + c + \varepsilon_c, \tag{17–8}$$

where

$\varepsilon_a$    is the error made in the quantization of $a$,

$\varepsilon_b$    is the error made in the quantization of $b$,

$\varepsilon_c$    is the error made in the quantization of $c$,

$\varepsilon_x$    is the error made in the quantization of $x$, and

$\varepsilon_{x^2}$    is the error made in the quantization of $x^2$.

We can first make an observation that has important practical consequences: The polynomial coefficient $c$ is not multiplied by anything and therefore can (and should) be represented with 32 bits. In this way $\varepsilon_c$ will be <1 ppm and can be considered insignificant. On the other hand, $x$, $a$, and $b$ must be represented with 16 bits and therefore $\varepsilon_x$, $\varepsilon_{x^2}$, $\varepsilon_a$, and $\varepsilon_b$ are significant. However, if these 16-bit quantization

error terms are multiplied with each other, the result is <1 ppm and can be considered insignificant. Therefore (17–8) reduces to

$$y \approx ax^2 + bx + c + 2ax\varepsilon_x + a\varepsilon_{x^2} + x^2\varepsilon_a + b\varepsilon_x + x\varepsilon_b. \tag{17–9}$$

We can see from the total error $2ax\varepsilon_x + a\varepsilon_{x^2} + x^2\varepsilon_a + b\varepsilon_x + x\varepsilon_b$ that large values of $a$, $b$, and $x$ have an amplifying effect on the quantization errors. Luckily, there is a simple cure for large values of $x$: Always have $x$ start from zero. Instead of approximating $f(x)$ with $p(x)$, approximate $f(x)$ with $p(x-A)$, where $A$ is the start of the approximation interval. Note that this must be taken into account already when finding the coefficients for $p$.

It may also be possible to reduce the values of $a$ and $b$ by tuning range reduction suitably. In the case of log, sqrt, and similar type of functions, we notice that $a$ and $b$ tend to get smaller with large values of parameter $M$ (see formula (17–4) above). However, a very large value of $M$ may lead to mathematical problems in polynomial fitting, impractically small a, and so on. A value of $M=4$ is usually a good compromise. The approximation interval will thus be [8,16] in the case of log and [4,16] in the case of sqrt.

## 17.7   FUNCTION APPROXIMATION EXAMPLE

To summarize all of the above, let's study a complete example of function approximation using polynomials. We will approximate the function $\ln(x)$ and use [8,16] as the approximation interval. We select $S=4$ and $D=3$ and thus have to find the coefficients for the polynomial $p(x-A)=a(x-A)^3+b(x-A)^2+c(x-A)+d$ in the subintervals [8,10], [10,12], [12,14], and [14,16]. Our optimal approximation polynomial search routine returns to us the polynomial coefficients shown in Table 17-1.

Let's now see what happens with the input $x=987$. First x is normalized to the interval [8,16] by moving the binary point six places to the left. (In the fixed-point case, we don't really move anything; we just imagine there is a point on the right side of the fourth bit from the left. Then we take this into account when doing the final denormalization.) The normalized input becomes $987.2^{-6}=15.421875$. This

**Table 17-1**   Polynomial Coefficients versus Subinterval Range

| Subinterval | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| $8 \le x < 10$ | 0.0004627 | −0.007606 | 0.124933 | 2.079442 |
| $10 \le x < 12$ | 0.0002524 | −0.004910 | 0.099970 | 2.302585 |
| $12 \le x < 14$ | 0.0001526 | −0.003427 | 0.083318 | 2.484907 |
| $14 \le x < 16$ | 0.0000992 | −0.002526 | 0.071420 | 2.63906 |

value falls into the last subinterval [14,16]. With the values given in the table we get $p(x-14)=2.73579$. Denormalization yields $2.73579+6\cdot\ln(2)=6.89467$. The precise value for $\ln(987)$ is $6.89467004$.

## 17.8   CONCLUSIONS

We have explained the use of polynomial approximations for calculating function values and demonstrated their viability in a fixed-point numerical environment. Because of their fast computation and capability of efficient parallelization, polynomial approximations are the algorithm of choice for the realization of elementary functions in modern, pipelined DSP architectures.

## 17.9   REFERENCES

[1]  M. POWELL, *Approximation Theory and Methods*. Cambridge Univ. Press, Cambridge, UK, 1981.
[2]  J. RICE, *The Approximation of Functions*. Addison Wesley, Reading, MA, 1964.
[3]  A. TIMAN, *Theory of Approximation of Functions of a Real Variable*. Dover Publications, New York, 1994.
[4]  J. MULLER, *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, 1997.
[5]  P. TAN, "Table Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, June 1991, pp. 232–236.

## EDITOR COMMENTS

Here we provide details of the derivation of (17–2) in order to clarify the indexing used in the Lagrange interpolation formula given in (17–1). Repeating (17–1) with $n=2$, we have

$$p(x) = \sum_{i=0}^{2} p_i(x) = p_0(x) + p_1(x) + p_2(x).$$

Using Eq. (17–1′) we write polynomial $p(x)$ as

$$p(x) = f(x_0)\left[\frac{x-x_1}{x_0-x_1}\cdot\frac{x-x_2}{x_0-x_2}\right] + f(x_1)\left[\frac{x-x_0}{x_1-x_0}\cdot\frac{x-x_2}{x_1-x_2}\right]$$
$$+ f(x_2)\left[\frac{x-x_0}{x_2-x_0}\cdot\frac{x-x_1}{x_2-x_1}\right].$$

With

$$x_0 = 1,\ f(x_0) = 1,$$
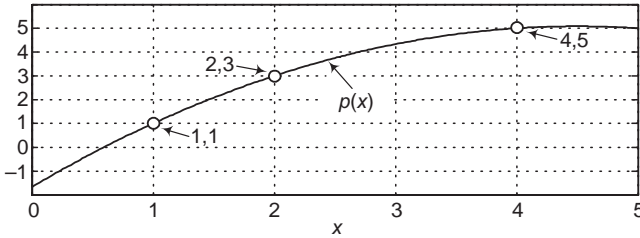$$x_1 = 2,\ f(x_1) = 3,$$
$$x_2 = 4,\ f(x_2) = 5,$$

polynomial $p(x)$ becomes

**Figure 17-4**   Second-degree $p(x)$ polynomial.

$$p(x) = 1\left[\frac{x-2}{1-2}\cdot\frac{x-4}{1-4}\right] + 3\left[\frac{x-1}{2-1}\cdot\frac{x-4}{2-4}\right] + 5\left[\frac{x-1}{4-1}\cdot\frac{x-2}{4-2}\right]$$

$$= \frac{(x-2)(x-4)}{3} + \frac{3(x-1)(x-4)}{-2} + \frac{5(x-1)(x-2)}{6}$$

$$= \frac{(-12)(x-2)(x-4) + (18)(3)(x-1)(x-4) + (-6)(5)(x-1)(x-2)}{-36}$$

$$= \frac{2x^2 - 18x + 10}{-6} = \frac{-x^2}{3} + 3x - \frac{5}{3}.$$

So the coefficients of the above polynomial $p(x)$ are $-1/3$, 3, and $-5/3$, agreeing with (17–2). Figure 17-4 depicts the continuous second-degree $p(x)$ polynomial defined by (17–2).

To expand on the explanation for the proposed function approximation algorithm, here is a detailed example of the method. Assume we are trying to approximate $\ln(x)$ between [1,2] using a third-degree polynomial, $D=3$, and the $(x,f(x))$ coordinates of our initial four fitting points are $(1,0)$, $(1.4,\ln(1.4))$, $(1.8,\ln(1.8))$, $(2,\ln(2))$ as shown by the dots in Figure 17-5(a). Performing step 4 of the proposed algorithm, we use Lagrange interpolation to obtain our initial third-degree $p(x)$ polynomial of

$$p(x) = 0.09697899x^3 - 0.67342994x^2 + 2.03458402x - 1.45813308. \quad (17\text{–}10)$$

Evaluating $p(x)$ provides the curve in Figure 17-5(a). This $p(x)$ has the absolute approximation error $|e(x)| = |\ln(x) - p(x)|$ shown in Figure 17-5(b). The extrema (peaks) in the error function are not equal so our initial fitting points were not optimal. Step 5 of the proposed algorithm tells us how to move the intermediate fitting points 1.4 and 1.8. (The phrase *intermediate fitting points* means all the fitting points except the first and last points, so $x_1$ and $x_2$ are the intermediate fitting points in this example.)

Because we have two intermediate fitting points, $x_1$ and $x_2$, we must examine Interval 1 and Interval 2 to find the $x$-coordinates ($x_{1p}$ and $x_{2p}$) of the error peaks in those two intervals. Knowing $x_{1p}$ and $x_{2p}$, we perform step 6 by moving fitting point $x_1$ toward $x_{1p}$ by the amount $s(x_{1p}-x_1)$. Next we move fitting point $x_2$ toward $x_{2p}$ by the amount $s(x_{2p}-x_2)$. That completes the first iteration of the algorithm.
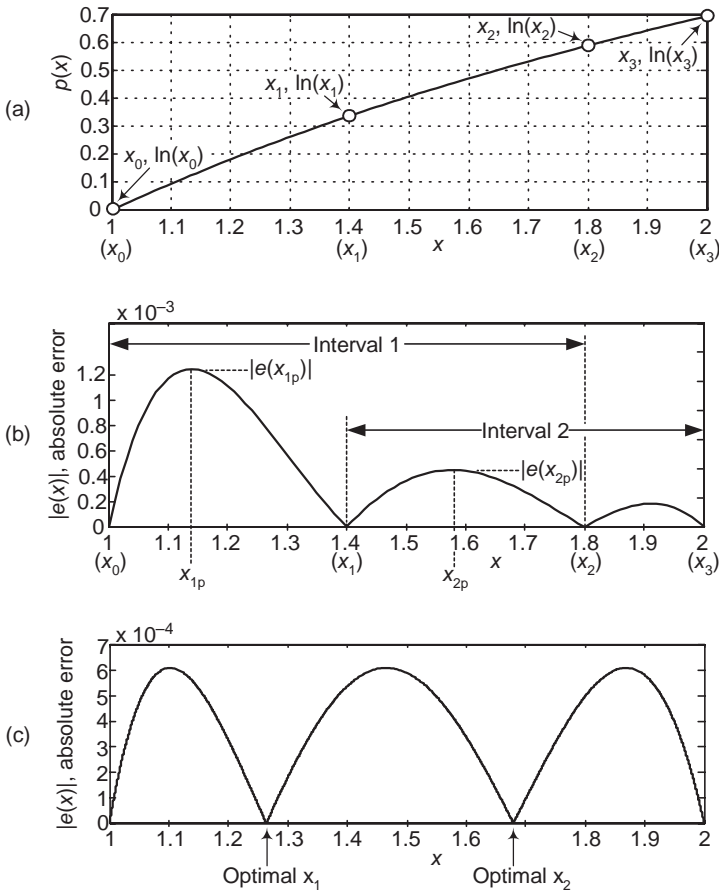
**Figure 17-5**    Approximating ln(x): (a) initial fitting points; (b) first-iteration error function; (c) final error function.

Continuing, we use the new fitting points (and Lagrange interpolation) to compute a new $p(x)$ polynomial, and begin analyzing its performance to find two new intermediate fitting points. We repeat this process and after $10D$ (30) iterations of steps 4 through 8 we arrive at the optimum fitting-points locations that yield the optimum $p(x)$ polynomial, whose absolute approximation error is shown in Figure 17-5(c). Notice how the final extrema in that error function are now equal—that's what the author calls "optimal."

One last issue to consider is that to reduce the computational workload in evaluating polynomials *Horner's rule* should be used. That rule reduces the necessary number of multiplies during polynomial computations. For example, computing

$$p(x) = ax^4 + bx^3 + cx^2 + dx + e$$

requires seven multiplications. Using Horner's rule to rewrite $p(x)$, we have the equivalent

$$p(x) = x(x(x(ax + b) + c) + d) + e$$

requiring only four multiplications. Horner's rule lends itself well to implementations using programmable chips having single-cycle *multiply and accumulate* (MAC) capabilities.

# Chapter 18

# Efficient Approximations for the Arctangent Function

**Sreeraman Rajan**
**Sichun Wang**
**Robert Inkol**
**Alain Joyal**
*Defense Research and Development Canada*

**T**his chapter provides several efficient approximations for the arctangent function using Lagrange interpolation and minimax optimization techniques. These approximations are particularly useful for the implementation of the arctangent function when processing power, memory, and power consumption are important issues. In addition to comparing the errors and the computational workload of these approximations, we also give methods for extending arctangent approximations to all four quadrants.

## 18.1 ARCTANGENT APPROXIMATIONS

The evaluation of the arctangent function is commonly encountered in real-time applications of signal processing, such as biomedical engineering, instrumentation, communications, and electronic warfare systems. Numerous algorithms are available to implement the arctangent function when computational cost is unimportant. The most direct solution is based on the Taylor series expansion for $|x| < 1$, which is:

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots . \qquad (18\text{--}1)$$

Since $\arctan(x) = \pi/2 - \arctan(x^{-1})$, we can use (18–1) to calculate $\arctan(x)$ for all $x$. However, the series (18–1) converges slowly for values of $x$ close to 1 and hence is inefficient. In order to expedite the convergence of the expansion, transformations such as the one discussed in [1] may be used. However, these transformations involve the computation of square roots. Another interesting transformation to handle slow convergence is the idea of series folding, where the argument $x$ is repeatedly transformed to a number close to zero by using the following transformation for various values of $k$ [2]:

$$y = \frac{1 - kx}{k + x}, \quad k > 0. \qquad (18\text{--}2)$$

The expressions, $\arctan(x)$ and $\arctan(y)$, are related by the following:

$$\arctan(y) = \arctan(k^{-1}) - \arctan(x). \qquad (18\text{--}3)$$

Although this technique circumvents the convergence issues, it requires additional arithmetic operations. It is also clear from the discussion that this process requires a lookup table for $\arctan(k^{-1})$.

Iterative algorithms, such as the CORDIC (*CO*rdinate *R*otation *DI*gital *C*omputer) algorithm, have been successfully used to compute trigonometric functions [3]. These algorithms require only shift and add operations and are thus multiplierless [4]. However, the sequential nature of these algorithms makes them less attractive when speed is a major concern while attempts to increase speed have been at the expense of additional hardware [5]. Lookup table–based approaches to the computation of inverse trigonometric functions are very fast, but require considerable memory [6], [7].

Polynomial and rational function approximations have been proposed in the literature [5] that are more suitable for numerical coprocessors. Most of these approximations are recursive in nature. Some of these approximations are linear combinations of orthogonal polynomials in a given closed interval. Given the precision requirements, a fixed formula for such approximations may be easily derived. Approximations using polynomials of large degrees are computationally expensive. Rational approximations are in principle more accurate than polynomial approximations for the same number of coefficients. However, the required division operations are relatively complex to implement; iterative techniques, such as those based on Newton's method, are often used.

The aforementioned approaches are best suited for applications where processing power and/or memory are readily available. However, for many applications, simpler and more efficient ways of evaluating $\arctan(x)$ are desirable. Here we propose simple approximations for evaluating the arctangent function that may be easily implemented in hardware with limited memory and processing power.

## 18.2 PROPOSED METHODOLOGY AND APPROXIMATIONS

Approximations to the arctangent function can be obtained using second- and third-order polynomials and simple rational functions. For these classes of approximation functions, Lagrange interpolation–based and minimax criterion–based approaches are used to obtain the polynomial coefficients. The following is our initial derivation of an arctangent approximation using Lagrange interpolation.

Consider the three points $x_0 = -1$, $x_1 = 0$, $x_2 = 1$. Let

$$\Phi(x) = \arctan\left(\frac{1+x}{1-x}\right), \quad -1 \le x \le 1.$$

According to the Lagrange interpolation formula, we have

$$\Phi(x) \approx \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}\Phi(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}\Phi(x_1)$$
$$+ \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}\Phi(x_2) = \frac{\pi}{4}(x+1), \quad -1 \le x \le 1. \tag{18–4}$$

It can be verified that $\Phi(x) = \pi/4 + \arctan(x)$; hence, $\arctan(x)$ can be approximated by the first-order formula

$$\arctan(x) \approx \frac{\pi}{4}x, \quad -1 \le x \le 1. \tag{18–5}$$

This linear approximation has been used in [8] for FM demodulation due to its minimal complexity. It requires only a scaling operation by a fixed constant and can be computed in one cycle in most processors. Figure 18-1 shows the deviation of this linear (first-order) approximation from the arctangent function on the interval $[-1,1]$.

This deviation is antisymmetric about $x = 0$ and attains a maximum at $x_{max} = \pm(4/\pi - 1)^{1/2}$, and is approximately quadratic in nature in $[0,1]$. The maximum error due to approximation (18–5) is about 0.07 radians (4°).

Here's a useful trick to improve the accuracy of the first-order approximation (18–5). We obtain an expression for the error in (18–5); we then subtract that error
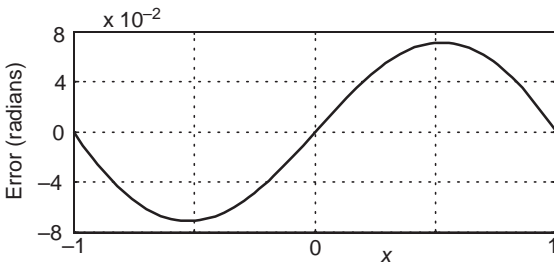


**Figure 18-1**   Approximation errors using linear approximation (18–5).

from (18–5) to yield a more accurate arctan approximation. Following this strategy, consider the error function given by

$$\Psi(x) = \arctan(x) - \frac{\pi}{4}x, \quad -1 \le x \le 1. \tag{18–6}$$

Applying Lagrange's interpolation formula to $\Psi(x)$ using $x_0 = 0$, $x_1 = x_{max}$, and $x_2 = 1$, $\Psi(x)$ can be approximated by

$$\Psi(x) \approx 0.285x(1 - |x|), \quad -1 \le x \le 1. \tag{18–7}$$

where the odd symmetry of $\Psi(x)$ has been applied. A second-order, and more accurate, approximation for $\arctan(x)$ thus becomes:

$$\arctan(x) \approx \frac{\pi}{4}x + 0.285x(1 - |x|), \quad -1 \le x \le 1 \tag{18–8}$$

with a maximum absolute error of 0.0053 radians (0.3 degrees).

Exploring other second-order arctan algorithms, it is also of interest to approximate the error function, $\Psi(x)$, given by (18–6) by a second-order polynomial of the form $(\alpha x^2 - \alpha x)$ that passes through the endpoints of the interval [0,1]. The optimum parameter, $\alpha > 0$, may be obtained using the following minimax criterion:

$$J = \min_{\alpha}\{\max_{0 \le x \le 1}\{|\Psi(x) - \alpha x(x-1)|\}\}. \tag{18–9}$$

The above criterion (18–9) does the following: For every $\alpha$, the maximum absolute error between $\Psi(x)$ and the second-order polynomial $(\alpha x^2 - \alpha x)$ is determined. This error, as a function of $\alpha$, is then minimized. This will yield a unique $\alpha$ that gives the least error $J$—hence the name *minimax criterion*, and the approximation is called *minimax approximation*.

Using an extensive computer search, the optimum $\alpha \approx 0.273$ and the following approximation for the $\arctan(x)$ is obtained:

$$\arctan(x) \approx \frac{\pi}{4}x + 0.273x(1 - |x|), \quad -1 \le x \le 1 \tag{18–10}$$

with a maximum absolute error of 0.0038 radians (0.22°).

A third-order polynomial is also a good candidate to fit the error curve shown in Figure 18-1. When the third-order polynomial is constrained to be of the form $\alpha x^3 + \beta x$, the best minimax approximation for the arctangent function is given by

$$\arctan(x) \approx \frac{\pi}{4}x + x(0.186982 - 0.191942x^2), \quad -1 \le x \le 1. \tag{18–11}$$

The maximum absolute error for this approximation is 0.005 radians (0.29°) and is worse than that given by (18–10).

A better third-order polynomial for approximating the error is of the following form: $x(x-1)(\alpha x - \beta)$ for $x$ in the interval [0,1]. Using the minimax criterion, the following polynomial is identified as the optimal approximation to $\arctan(x)$:

$$\arctan(x) \approx \frac{\pi}{4}x - x(|x|-1)(0.2447 + 0.0663|x|), \quad -1 \le x \le 1 \qquad (18\text{--}12)$$

with a maximum absolute error of 0.0015 radians (0.086°).

Another candidate for approximating the arctangent function is from the class of rational functions of the form $\tau(x) = x/(1+\beta x^2)$ in the interval $[-1,1]$. For $0 \le \beta \le 1$, the first derivative of $\tau(x)$ is positive and the second derivative of $\tau(x)$ is negative. This implies that $\tau(x)$ has a shape very similar to that of the arctangent function in the same interval. Using the minimax criterion, the following approximation is obtained:

$$\arctan(x) \approx \frac{x}{1 + 0.28086x^2}, \quad -1 \le x \le 1 \qquad (18\text{--}13)$$

with the maximum absolute error to be about 0.0047 radians (0.27°).

Recently a similar idea was presented in [7] with a maximum absolute error of 0.0049 radians (0.28°). The approximation in [7] is as follows:

$$\arctan(x) \approx \frac{x}{1 + 0.28125x^2}, \quad -1 \le x \le 1. \qquad (18\text{--}14)$$

The scaling constant, 0.28125, was chosen to permit multiplication to be performed (as discussed later) with two arithmetic binary right-shifts and a single addition. This yields a negligible increase in maximum error.

## 18.3  DISCUSSION

Now we compare the various arctangent approximations presented here. Table 18-1 contains the maximum error, number of adds, multiplies, and divides for various arctan approximations. The third-order approximation given by (18–12) has the least error among the proposed approximations, but has the highest computational cost. The second-order approximation given by (18–10) has the next lowest error and has

**Table 18-1**  Maximum Approximation Error and Computational Workload

| Equation | Error (rad.) | Adds | Multiplies | Divides |
|---|---|---|---|---|
| (18–5) | 0.07 | 0 | 1 | 0 |
| (18–8) | 0.0053 | 1 | 2 | 0 |
| (18–10) | 0.0038 | 1 | 2 | 0 |
| (18–11) | 0.005 | 1 | 3 | 0 |
| (18–12) | 0.0015 | 2 | 3 | 0 |
| (18–13) | 0.0047 | 1 | 2 | 1 |
| (18–14) | 0.0049 | 2 | 1 | 1 |

fewer operations than (18–12). Hence (18–10) provides a favorable compromise between accuracy and computational cost. The linear approximation given by (18–5) has only one multiplication operation, but has the least accuracy. With a single-cycle multiply-and-accumulate (MAC) processor, the evaluation of the arctangent using (18–10) would take only two cycles, if the sign information of the argument is already available. However, one needs to check the sign of the argument, which may take an extra cycle.

It is interesting to note that all the second-order approximations given in this chapter have better accuracy than the third-order polynomial approximation given in reference [9]. The approximation in [9] was used in the computation of bearing and had an accuracy of only about 0.0175 radians (1°).

Comparison of the approximations to arctan($x$) using the proposed two second-order approximations given by (18–8) and (18–10) are shown in Figure 18-2. These approximations have maximum errors that are an order of magnitude better than that of the linear approximation (18–5). Furthermore, the second-order approximation given by (18–8) provides better accuracy for the subintervals $1>x>0.5$ and $-1<x<-0.5$, where the maximum error is only about 0.001 radians (0.057°). No such observation can be made for (18–10).

Figure 18-3 shows that (18–12) provides the best accuracy among the third-order approximations considered in this chapter.
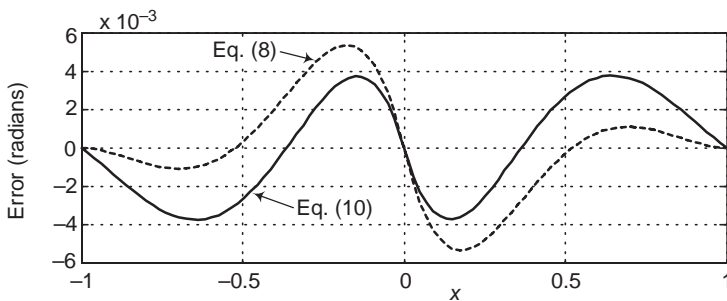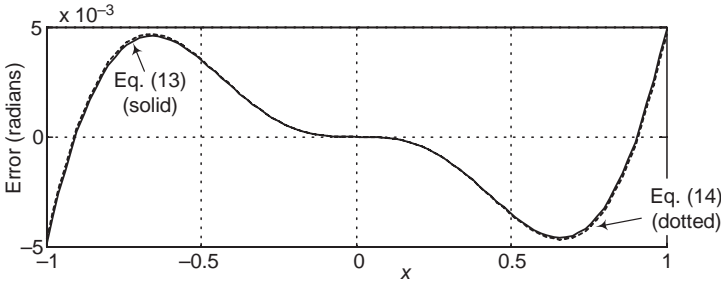


**Figure 18-2**    Approximation errors using second-order polynomials (18–8), (18–10).
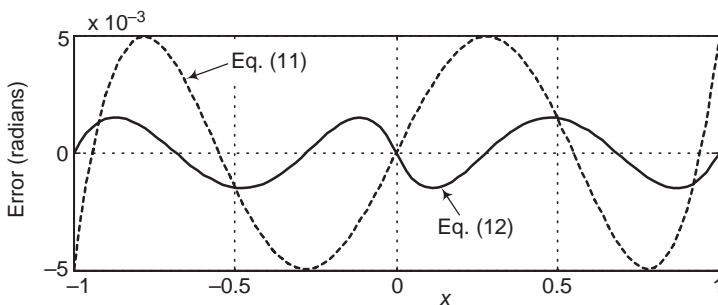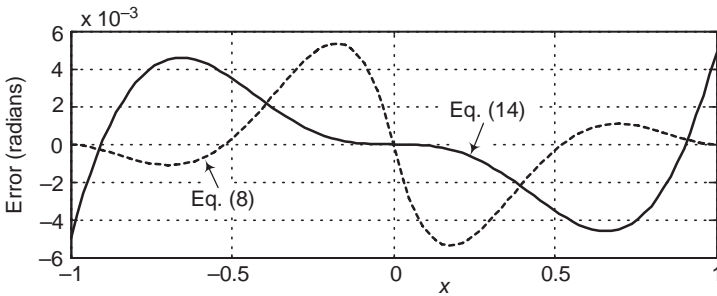


**Figure 18-3**    Approximation errors using cubic polynomials (18–11), (18–12).

**Figure 18-4**    Approximation errors using rational functions (18–13), (18–14).



**Figure 18-5**    Approximation errors using second-order polynomial (18–8) and rational approximation (18–14).

The error due to the rational approximations, (18–13) and (18–14), is presented in Figure 18-4. The behavior of these approximations is almost identical, except near the maximum error. The rational approximations given by (18–13) and (18–14) are computationally more expensive than the second-order ones given by (18–8) and (18–10), though this cost increase is partly offset by the elimination of the need for sign comparisons. Approximations using (18–13) or (18–14) have a division operation that may slow down the processing. The approximation using (18–14) and the approximation given by (18–8) intersect at approximately $x_c = 0.3933$ in the interval [0,1]. For values of $|x| > x_c$, the proposed approximation given by (18–8) yields better accuracy, as shown in Figure 18-5, and for values of $|x| < x_c$, the approximation in (18–14) gives better accuracy.

An alternative approach to improving the accuracy, though at the expense of computational cost, would be to combine (18–8) and (18–14). This combined equation is given by

$$\arctan(x) \approx \gamma A + (1 - \gamma)B \tag{18–15}$$

where $A$ is given by (18–14) and $B$ is given by (18–8). The weighting variable $\gamma$ is given by the following:

$$\gamma = \begin{cases} 1, & 0 \leq |x| \leq x_c \\ 0, & x_c < |x| \leq 1. \end{cases} \tag{18–16}$$

Using this approach, the maximum error is less than 0.0025 radians (0.14°) as can be seen from Figure 18-5. Also the approximation given by (18–15) is better than (18–8) and (18–14) in the minimax sense.

## 18.4  FOUR-QUADRANT APPROXIMATIONS

The arctan algorithms presented thus far are applicable for angles in the range of $-\pi/4$ to $\pi/4$ radians. Here we provide the scheme for extending that angular range to $-\pi$ to $\pi$ radians. Let $z = I + jQ$ be any complex number and let $x = Q/I$ and $y = I/Q$. In signal processing terms, $I$ may be considered as the in-phase component of a complex signal, while $Q$ may be considered as the quadrature component. The four-quadrant arctangent function atan2($I,Q$) can be evaluated over the four quadrants by substituting either $Q/I$ or $I/Q$ appropriately for $x$ in the above arctangent approximations. All the approximations given above can be extended in a straightforward manner to all four quadrants. Due to space limitations, we consider only the extensions for approximations given by (18–10) and compare them with the four-quadrant arctangent expressions given for (18–13).

In order to obtain the four-quadrant arctangent approximations, the range over which the approximation operates is extended. The complex plane is divided into eight octants where octant I, for example, covers the angle range of 0 to $\pi/4$ radians. The four-quadrant calculations are thus reduced to the first-octant calculations, and using the rotational symmetries of arctangent function, the approximations for the other octants can be easily obtained. Tables 18-2 and 18-3 provide the four-quadrant approximations based on (18–10) and (18–13). It should be noted that these tables use the following definition for determining the sign value of an argument:

$$\text{sign}(z) = \begin{cases} 1, & z \geq 0 \\ -1, & z < 0. \end{cases} \tag{18–17}$$

Many desirable features of the four-quadrant rational approximation based on (18–14) are given in [7]. The multiplication by 0.28125 in the denominator of

**Table 18-2**  Second-Order Approximation (18–10) versus Octant Locations

| Octant | Approximation |
|---|---|
| I, VIII | $\dfrac{Q}{I}[1.0584 - \text{sign}(Q) \cdot 0.273(Q/I)]$ |
| II, III | $\dfrac{\pi}{2} - \dfrac{I}{Q}[1.0584 - \text{sign}(I) \cdot 0.273(I/Q)]$ |
| IV, V | $\text{sign}(Q) \cdot \pi + \dfrac{Q}{I}[1.0584 + \text{sign}(Q) \cdot 0.273(Q/I)]$ |
| VI, VII | $-\dfrac{\pi}{2} - \dfrac{I}{Q}[1.0584 + \text{sign}(I) \cdot 0.273(I/Q)]$ |

**Table 18-3**    Rational Approximation (18–13) versus
Octant Locations

| Octant | Approximation |
|---|---|
| I, VIII | $\dfrac{Q/I}{1+0.28086(Q^2/I^2)}$ |
| II, III | $\dfrac{\pi}{2}-\dfrac{I/Q}{1+0.28086(I^2/Q^2)}$ |
| IV, V | $\text{sign}(Q)\cdot\pi+\dfrac{Q/I}{1+0.28086(Q^2/I^2)}$ |
| VI, VII | $-\dfrac{\pi}{2}-\dfrac{I/Q}{1+0.28086(I^2/Q^2)}$ |

(18–14) can be implemented as a sum of two arithmetic right-shifts, $x^2/4$ and $x^2/32$. (This fact accounts for the two adds and single-multiply requirements for (18–14) in Table 18-1.) The constants appearing in the four-quadrant approximations using either (18–8) or (18–13) cannot be implemented with shifts. However, the complexity of the four-quadrant arctangent approximations using any of (18–8), (18–13), and (18–14) are the same. The four-quadrant arctangent approximation using (18–10) has better accuracy and deserves consideration.

## 18.5  CONCLUSIONS

Simple approximations to the arctangent function and four-quadrant arctangent functions have been introduced. The second-order polynomial in (18–10) provides a favorable compromise between accuracy and computational cost. Furthermore, it is well suited for implementation in hardware. The demodulators in digital receivers and software-defined radios are an important application where these approximations may be useful.

## 18.6  REFERENCES

[1] R. INKOL, A. JOYAL, and S. WANG, "Simple Approximations to the Four-Quadrant Arctangent Function," *Technical Note*, DRDC Ottawa TN 2003-180, November 2003.

[2] D. DAS, K. MUKHOPADHYAYA, and B. SINHA, "Implementation of Four Common Functions on an LNS Co-processor," *IEEE Transactions on Computers*, vol. 44, no. 1, January 1995, pp. 155–161.

[3] W. WONG and E. GOTO, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, March 1994, pp. 278–294.

[4] D. HWANG, D. FU, and A. WILLSON, "A 400-MHz Processor for the Conversion of Rectangular to Polar Coordinates in 0.25-μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 10, October 2003.

[5] I. KOREN and O. ZINATY, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Transactions on Computers*, vol. 39, no. 8, August 1990.

[6] M. Rodriguez, J. Zurawski, and J. Gosling, "Hardware Evaluation of Mathematical Functions," *IEE Proc.*, vol. 128, pt. E, no. 4, July 1981.

[7] R. Lyons, "Another Contender in the Arctangent Race," *IEEE Signal Processing*, vol. 20, no.1, January 2004, pp. 109–111, [See Chapter 15 herein.]

[8] J. Shima, "FM Demodulation Using a Digital Radio and Digital Signal Processing," M.Sc. thesis, University of Florida, 1995.

[9] T. Stover, "An Improved Arctangent Calculation Algorithm for DIFAR Bearing Computation in the ASP," Report No. NADC-82013-30, Naval Air Systems Command, U.S. Department of the Navy, January 1982.

## EDITOR COMMENTS

Here we supply the details of how Lagrange interpolation was used to obtain the $\Psi(x)$ error expression in (18–7). We start by recalling that the value of $x$ where (18–5) experiences its maximum error was given as

$$x_{max} = \pm\sqrt{4/\pi - 1} = \pm 0.523.$$

That $x_{max}$ value was obtained by setting the derivative of (18–6) to zero and solving for $x$.

In preparation for using the Lagrange interpolation method to find a second-order polynomial approximation to the positive-$x$ portion of (18–5)'s error curve in Figure 18-1, we start by identifying three *fitting points* as those shown in Figure 18-6, with (18–5)'s error shown as the solid curve.

The three fitting points $[x_0, \Psi(x_0)], [x_1, \Psi(x_1)], [x_2, \Psi(x_2)]$ are $(0,0), (x_{max}, E_{max}), (1,0)$ as shown in Figure 18-6. Using the principles of Lagrange interpolation we can approximate Figure 18-1's $\Psi(x)$ error curve as

$$\Psi(x) = (0)\frac{x - x_{max}}{0 - x_{max}}\frac{x - 1}{0 - 1} + (E_{max})\frac{x - 0}{x_{max} - 0}\frac{x - 1}{x_{max} - 1} + (0)\frac{x - 0}{1 - 0}\frac{x - x_{max}}{1 - x_{max}}$$

$$= (E_{max})\frac{x}{x_{max}}\frac{x - 1}{x_{max} - 1} = \frac{x(x - 1)E_{max}}{x_{max}(x_{max} - 1)}.$$

(For those readers unfamiliar with Lagrange interpolation, Chapter 17 provides tutorial material on that topic.) Knowing that $x_{max} = 0.523$, and from (18–6) that
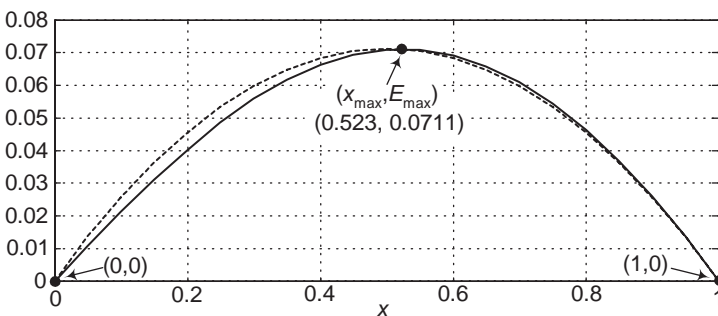


**Figure 18-6**    Error of the linear arctan approximation (18–5).

$$E_{max} = \arctan(x_{max}) - \frac{\pi x_{max}}{4} = 0.0711,$$

we can write our desired $\Psi(x)$ error function's Second-order polynomial approximation as

$$\Psi(x) \approx \frac{0.0711 \cdot x(x-1)}{0.523(0.523-1)} = -0.285 \cdot x(x-1)$$

for $0 \le x \le 1$. This $\Psi(x)$ error polynomial agrees with (18–7), which is what we set out to show. To indicate the accuracy of the Lagrange interpolation, $\Psi(x)$ is shown as the dashed curve in Figure 18-6, where it indeed passes through the three fitting points.

As a minor clarification, on first glance it may not be obvious how the authors arrived at the number of multiplies (Mults) in Table 18-1. For example (18–12), over the range $0 \le x \le 1$, is repeated here as

$$\arctan(x) \approx \frac{\pi}{4}x - x(x-1)(0.2447 + 0.0663x).$$

Multiplying through by the factors in $\arctan(x)$ yields

$$\arctan(x) \approx -0.0663x^3 - 0.1784x^2 + 1.03x,$$

which appears to require five multiplications. However, using Horner's rule reduces the number of multiplications from five to three. The above arctangent can be computed as

$$\arctan(x) \approx [(-0.0663x - 0.1784)x + 1.03]x.$$

One important aspect of the arctangent approximations described in this chapter is the source of the independent variable $x$. The computational requirements stated in Table 18-1 assume that $x$ has been previously computed and is available for use in the various arctan approximations. If, however, we are working with complex-valued samples, $x = Q/I$, and only $I$ and $Q$ are available, then a divide operation is required to compute $x$. In this situation the computational workloads of the various arctan approximations are quite similar as shown in Table 18-4. (We say "quite

**Table 18-4**   Max Error and Computational Workload
When Only $I$ and $Q$ Are Available

| Equation | Error (rad.) | Adds | Multiplies | Divides |
|---|---|---|---|---|
| (18–5) | 0.07 | 0 | 1 | 1 |
| (18–8) | 0.0053 | 1 | 2 | 1 |
| (18–10) | 0.0038 | 1 | 2 | 1 |
| (18–11) | 0.005 | 1 | 3 | 1 |
| (18–12) | 0.0015 | 2 | 3 | 1 |
| (18–13) | 0.0047 | 1 | 4 | 1 |
| (18–14) | 0.0049 | 2 | 3 | 1 |

similar" because a divide operation typically requires more computational horse-power, processor cycles, than a multiply operation.) In this case (18–14) can be written as

$$\arctan(Q/I) \approx \frac{IQ}{I^2 + 0.28125Q^2}, \quad -1 \le Q/I \le 1 \qquad (18\text{–}18)$$

requiring three multiplies.

So the bottom line here is that the arctangent computational workloads are, of course, dependent on whether the values $x = Q/I$, $I$ and $Q$ only, or $I^2$ and $Q^2$ are available at the start of an arctan approximation computation. Again, the $(0.28125)(Q^2)$ product in the denominator of (18–18) can be implemented as sum of two arithmetic right-shifts. That is,

$$0.28125Q^2 = Q^2/4 + Q^2/32.$$

# Chapter 19

# A Differentiator with a Difference

**Richard Lyons**

*Besser Associates*

$\mathbf{I}$n this chapter we discuss a computationally efficient network that approximates the derivative of a low-frequency discrete-time sequence.

## 19.1  DISCRETE-TIME DIFFERENTIATION

Even though the concept of differentiation is not well defined for discrete signals, we can approximate the calculus of derivatives in our domain of discrete signals. To explain our approach, consider a continuous sinewave, whose frequency is $\omega_o$ radians/second, represented by

$$x(t) = \sin(2\pi f_o t) = \sin(\omega_o t). \qquad (19\text{--}1)$$

The derivative of that sinewave is

$$\frac{dx(t)}{dt} = \frac{\sin(\omega_o t)}{dt} = \omega_o \cos(\omega_o t). \qquad (19\text{--}2)$$

Equation (19–2) tells us that the derivative of a sinewave is a cosine wave whose amplitude is proportional to the original $x(t)$ sinewave's $\omega_o$ frequency, and that an ideal differentiator's frequency magnitude response is a straight line increasing with frequency $\omega$. A pair of discrete-time differentiators, a *first-difference* differentiator and a *central-difference* differentiator, are often used to approximate the time-domain derivative in (19–2).
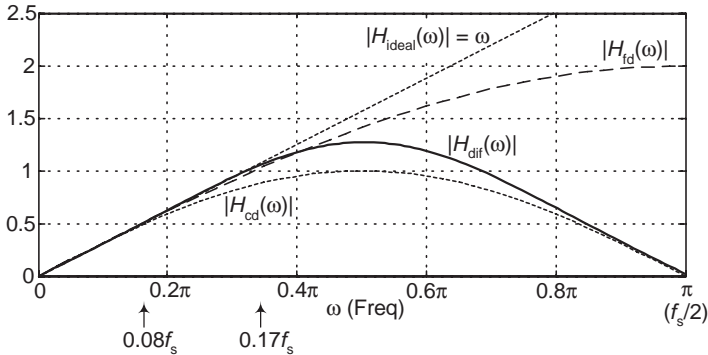
**Figure 19-1**    Differentiator performance responses.

The first-difference differentiator (what DSP purists call a *digital differencer*), the simple process of computing the difference between successive $x(n)$ signal samples, is defined in the time domain by

$$y_{fd}(n) = x(n) - x(n-1). \tag{19–3}$$

The frequency magnitude response of that differentiator is the dashed $|H_{fd}(\omega)|$ curve in Figure 19-1. (For comparison, we also show an ideal differentiator's straight-line $|H_{ideal}(\omega)| = \omega$ magnitude response in Figure 19-1. The frequency axis in that figure covers the positive normalized frequency range $0 \le \omega \le \pi$ samples/radian, corresponding to a cyclical frequency range of 0 to $f_s/2$, where $f_s$ is the $x(n)$ sample rate in Hz.)

Equation (19–3) is sweet in its simplicity but unfortunately its frequency response tends to amplify high-frequency noise that often contaminates real-world signals. For that reason the central-difference differentiator is often used in practice. The time-domain expression of the central-difference differentiator is

$$y_{cd}(n) = [x(n) - x(n-2)]/2. \tag{19–4}$$

The central-difference differentiator's frequency magnitude response is the dotted $|H_{cd}(\omega)|$ curve in Figure 19-1. The price we pay for $|H_{cd}(\omega)|$'s desirable high-frequency (noise) attenuation is that its frequency range of linear operation is only from zero to roughly $\omega = 0.16\pi$ samples/radian ($0.08f_s$ Hz), which is, unfortunately, less than the frequency range of linear operation of the first-difference differentiator.

## 19.2    AN IMPROVED DIFFERENTIATOR

Here we propose a computationally efficient differentiator that maintains the central-difference differentiator's beneficial high-frequency attenuation behavior, but extends its frequency range of linear operation. The improved differentiator is defined by
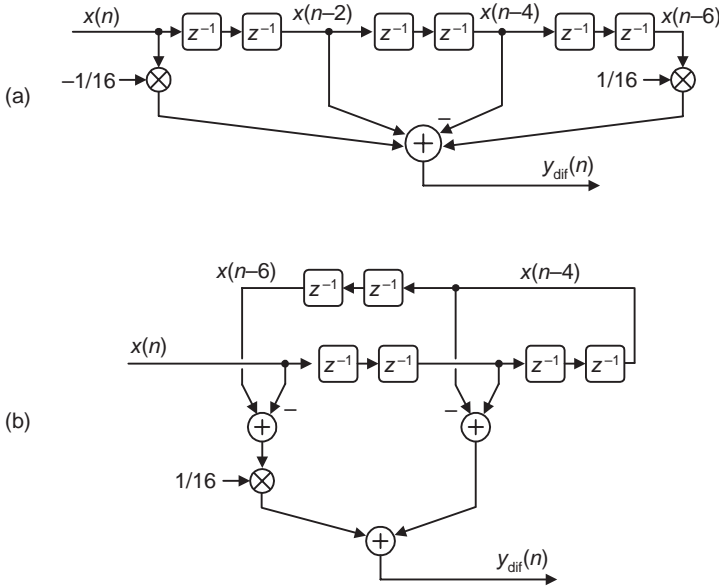
**Figure 19-2**    Efficient differentiator: (a) standard structure; (b) folded structure.

$$y_{\text{dif}}(x) = \frac{-x(n)}{16} + x(n-2) - x(n-4) + \frac{x(n-6)}{16}. \qquad (19\text{–}5)$$

This novel differentiator's frequency magnitude response is the solid $|H_{\text{dif}}(\omega)|$ curve in Figure 19-1, where we see its frequency range of linear operation extends from zero to approximately $\omega = 0.34\pi$ samples/radian $(0.17f_s\,\text{Hz})$. This is roughly double the frequency range of operation of the central-difference differentiator. The differentiator in (19–5) has a gain greater than that of the central-difference differentiator, so the solid curve in Figure 19-1 was normalized for easy comparison of $|H_{\text{cd}}(\omega)|$ and $|H_{\text{dif}}(\omega)|$.

The implementation of the differentiator is shown in Figure 19-2(a). The *folded-FIR* structure for this differentiator is presented in Figure 19-2(b), where only a single multiply is needed to compute a $y_{\text{dif}}(n)$ output sample. The fortunate aspect of the $y_{\text{dif}}(n)$ differentiator is that its nonunity coefficients $(\pm 1/16)$ are integer powers of two. This means that the multiplications in Figure 19-2 can be implemented with arithmetic right shifts by 4 bits. Happily, using bit-shifting means the network is a multiplierless differentiator!

This differentiator has another useful property—due to its linear phase, and seven-sample impulse reponse length, its time delay (group delay) is exactly three sample periods $(3/f_s)$. Such an integer delay makes the differentiator convenient for use when the $y_{\text{dif}}(n)$ output must be synchronized with other sequences in a system, such as with popular FM demodulation methods [1]–[3]. The only shortcoming of this very efficient differentiator is that its $x(n)$ input signals must be low frequency, less than $f_s/5$.

## 19.3 REFERENCES

[1] R. Lyons, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 2004, pp. 549–552.

[2] A. Bateman, "Quadrature Frequency Discriminator," *GlobalDSP*, October 2002.

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

# Part Four

# Signal Generation Techniques

# Chapter 20

# Recursive Discrete-Time Sinusoidal Oscillators

**Clay S. Turner**
*Wireless Systems Engineering Inc.*

**E**very few years a DSP article emerges that presents a method for generating sinusoidal functions with a DSP. While each oscillator structure has been developed pretty much on its own, there has not been presented a simple overlying theory that unifies all of the various oscillator structures and can easily allow one to look for other potential oscillator structures. We can find some guidance for our quest by first examining classical oscillators.

During the early 1900s, German physicist Heinrich Barkhausen formulated the necessary requirements for oscillation. He modeled an oscillator as an amplifier with its output fed back to its input via a phase-shifting network. From this model, he deduced and stated two necessary conditions for oscillation. The Barkhausen criteria, as they are now known, require the total loop gain to be equal to one, and the total loop phase shift needs to be a multiple of $2\pi$ radians. So, if we are to unify discrete-time oscillator designs into a single theory, we need to find the discrete-time equivalent of the Barkhausen criteria and use them to develop our theory.

But first we will look at how some very old trigonometric formulas, viewed in an unusual way, can be utilized for sinusoidal generation. Several oscillators have been designed via this approach. A sum-and-difference-of-angles formula written explicitly in recursive form is:

$$\cos(\varphi + \theta) = 2\cos(\theta)\cos(\varphi) - \cos(\varphi - \theta) \qquad (20\text{--}1)$$

We will refer to this as the *biquad* form. This is also called the *direct form* implementation. If the value $\theta$ is viewed as a step angle, then we can immediately see how this formula can be used to calculate the next sample of a sinusoid given two known samples spaced $\theta$ apart and a step factor that is just $2\cos(\theta)$. To see how this can be used for iterative generation of a sinusoid, just view the formula as follows:

$$NextCos = 2\cos(\theta) \times CurrentCos - LastCos \qquad (20\text{--}2)$$

Instead of using a single equation for a recurrence relation, a pair of trigonometric formulas may be used. An example that can be used to recursively generate sinusoids is:

$$\cos(\varphi + \theta) = \cos(\varphi)\cos(\theta) - \sin(\varphi)\sin(\theta)$$
$$\sin(\varphi + \theta) = \cos(\varphi)\sin(\theta) + \sin(\varphi)\cos(\theta) \qquad (20\text{--}3)$$

We will refer to this as the *coupled* form. The coupling is evident in that each equation uses not only its past value, but also the past value produced by the other equation. Again $\theta$ is used as the step angle per iteration, and this leads to an oscillator output frequency of $\theta f_s / 2\pi$ where $f_s$ is the sample rate. Just like the biquad form, the coupled form requires the use of two past values. This turns out to be the case for all sinusoidal oscillators that are limited to the use of real numbers [1]–[9].

## 20.1  MATRIX DERIVATION OF OSCILLATOR PROPERTIES

Now we desire to find a general enough process that can be used to express both the biquad and the coupled form equations. First we will denote the two past values (these are actually called state variables) as $x_1$ and $x_2$, and their "hatted" versions as the new values. Plus we notice that the update equations are linear for both forms, so we can write them in terms of matrix multiplication.

The matrix form of the update iteration for the biquad is:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} 2\cos(\theta) & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad (20\text{--}4)$$

Likewise the coupled form's update iteration is written as:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad (20\text{--}5)$$

The interpretation of the matrix iteration is that the column vector on the right-hand side contains the old state values, and when they are multiplied by the rotation matrix, you get a new set of state values. Then, for the next iteration, the *new values* from the last iteration are used as the *old values* for the next iteration. Thus each

iteration is just performed by a matrix multiplication times the state variables. While the idea of matrix math may seem to unnecessarily complicate things, it actually allows us to go and find new types of oscillators.

The term *rotation* is used since the matrix multiplication can be viewed from a special vantage point as doing a rotation. This special vantage point will be explained in more detail later. A general form that fits both of these aforementioned oscillators is the use of a 2-by-2 *rotation matrix* and two state variables. So a general oscillator iteration is written as:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{20-6}$$

We will now look at a numerical example of an oscillator iteration. (It is neither the biquad nor the coupled form mentioned earlier.) We will use

$$\begin{bmatrix} 0.95 & -1 \\ 0.0975 & 0.95 \end{bmatrix} \tag{20-7}$$

for the rotation matrix, and use

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{20-8}$$

for the initial state values. A graph of both state variables' values for the first 80 iterations is shown in Figure 20-1.

Now that we've seen a numerical example and two analytical examples, naturally the question becomes, "What are the constraints on the values of the four elements in the rotation matrix that will still allow the matrix to function for an oscillator iteration?" It turns out that there are two constraints and these are the discrete-time equivalent of the Barkhausen criteria. They are:

$$ad - bc = 1$$
$$|a + d| < 2 \tag{20-9}$$
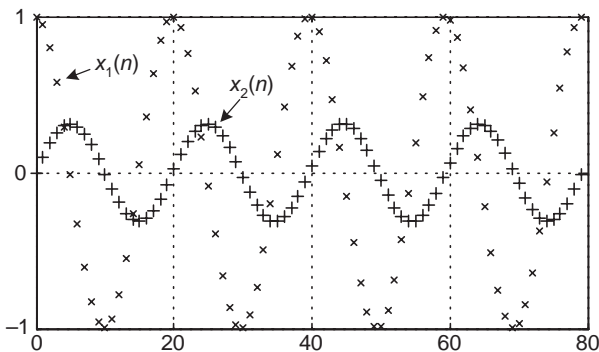


**Figure 20-1**    Output from example numeric oscillator.

The first constraint says the determinant of the rotation matrix must be one. This is analogous to saying the loop gain is unity. The second constraint (assuming the first constraint is met) says the matrix has complex eigenvalues. This means the oscillator output will eventually repeat. This is a discrete-time equivalent to Barkhausen's criterion for periodicity. While not obvious, it can be shown from these two constraints that both matrix elements, $b$ and $c$, must be nonzero—thus the rotation matrix can't be triangular. What we have done here is basically come up with a set of rules that can be easily applied to any 2-by-2 matrix to see whether it can be used to make an oscillator. And soon we will come up with more rules that will allow you to identify the type of oscillator just by looking at its matrix.

Now we will make a brief sojourn into eigenvalues and eigenvectors. *Eigen* comes from German, meaning *characteristic*. The reason we need to make this side trip is that this theory will allow us to perform an adroit factoring of the rotation matrix. And this will allow us to ascertain the oscillator's properties and determine the initial values for the state variables.

We are used to the idea of identity operations such as adding zero and multiplying by one. An analogous question in matrix theory is, "Is there a vector $x$ that when multiplied by a matrix $A$ results in a scalar multiple, $\lambda$, of the original vector?" Mathematically this is written as:

$$Ax = \lambda x \tag{20–10}$$

When this is satisfied, $\lambda$ is an eigenvalue and $x$ is its corresponding eigenvector of the matrix $A$. For our 2-by-2 matrices that obey the Barkhausen criteria, there will be two eigenvalues—in fact they are complex conjugates of each other and each has a magnitude of one. For the rest of this chapter, the rotation matrix $A$ is assumed to be a 2-by-2 matrix consisting of four real-valued elements and it obeys the aforementioned Barkhausen criteria. Explicitly, matrix $A$ is:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{20–11}$$

If we let the vector $x$ contain the initial state values, the $n$th output, $y(n)$, of the oscillator can be written as:

$$y(n) = A^n x \tag{20–12}$$

Now we will use a wonderful result of eigenvalue theory and factor the general matrix into a product of three matrices. This actually makes raising a matrix to a power much easier to perform.

$$A = SDS^{-1} \tag{20–13}$$

Thus $A^n$ can be written as $(SDS^{-1})(SDS^{-1})(SDS^{-1}) \ldots (SDS^{-1})$, and after canceling out paired $S^{-1}S$ terms, we get the following simplification for our oscillator output:

$$y(n) = SD^n S^{-1} x \tag{20–14}$$

While at first blush this doesn't seem to help, let's talk about the contents of the $S$ and $D$ matrices. First the $D$ matrix is a diagonal matrix whose only non-zero elements are on the main diagonal (upper left and lower right for our case). These elements are the eigenvalues. Also raising a diagonal matrix to a power is simply the raising of the diagonal elements to the same power. In terms of the original rotation matrix elements, the diagonal matrix is found thus:

$$D = \begin{bmatrix} e^{j\theta} & 0 \\ 0 & e^{-j\theta} \end{bmatrix} \tag{20--15}$$

where

$$\theta = \cos^{-1}\left(\frac{\Delta}{2}\right) \tag{20--16}$$

and

$$\Delta = a + d \tag{20--17}$$

It is at this point that we can see how raising the matrix $D$ to a power effects a rotation. In fact, $\theta$ is the step angle of the oscillator per iteration. Now the *change of basis* matrix, $S$, contains the eigenvectors that correspond to the eigenvalues used in matrix $D$. Again in terms of the original rotation matrix elements, the matrix $S$ is found thus:

$$S = \begin{bmatrix} 1 & 1 \\ \psi e^{j\phi} & \psi e^{-j\phi} \end{bmatrix} \tag{20--18}$$

where

$$\psi = \sqrt{\frac{-c}{b}} \tag{20--19}$$

and

$$\phi = \arg(\eta) \tag{20--20}$$

where

$$\eta = \frac{(d-a) + j\sqrt{4 - \Delta^2}}{2b} \tag{20--21}$$

It is interesting to comment on the fact that a real-valued rotation matrix is factored into a product of complex-valued matrices. However, the implemented oscillators will use only real-valued numbers. This brings to mind a saying attributed to the French mathematician Jacques Hadamard: "The shortest path between truths in the real domain passes through the complex domain."

Now the term *change of basis* was mentioned in connection with matrix $S$. This is used since the matrices $S$ and $S^{-1}$ map between external and internal space. This should be viewed as the state variables undergoing three processes. The first is a

transformation to internal space representation. The second process is a pair of rotations performed on them, and the third is a transformation back to external space. In internal space, the two rotations are in opposite directions—this allows us to combine complex numbers so as to result in only real numbers. Now let the variable $z$ be an internal representation as follows:

$$z = S^{-1}x \tag{20–22}$$

Next let $x$ have an initial value so that

$$z = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{20–23}$$

Then our oscillator output is simply written as

$$y(n) = SD^n z \tag{20–24}$$

This after simplification yields

$$y(n) = \begin{bmatrix} \cos(n\theta) \\ \psi\cos(n\theta + \phi) \end{bmatrix} \tag{20–25}$$

Likewise, given our initial choice for $z$, the initial state values are

$$x = Sz \tag{20–26}$$

This after simplification yields

$$x = \begin{bmatrix} 1 \\ \psi\cos(\phi) \end{bmatrix} \tag{20–27}$$

which is of course just $y(0)$.


## 20.2   INTERPRETING THE ROTATION MATRIX

The attractive aspect of this analysis method is that we can now evaluate an oscillator's behavior by merely looking at its rotation matrix! We see that our analysis includes two angles. The angle $\theta$ is the step angle per iteration and the angle $\phi$ is the phase shift between the two state variables. So, we can see that if we desire an oscillator to have quadrature outputs (the two state variables), then $\phi$ must be ±90 degrees, which in terms of the matrix elements means that the two values on the main diagonal must be the same! So if $a=d$, we have a quadrature oscillator. Likewise the scaling factor $\psi$ tells us the amplitude of the second state variable relative to the first one. If we desire the two outputs to have equal amplitudes, then the off-diagonal elements must be negatives of each other! That is, $b=-c$. So looking back at the matrix used in our numerical example, we can quickly ascertain the outputs are in quadrature but will have unequal amplitudes, as its graph confirms.

Now when it comes to programming the oscillator in a DSP chip, we would not normally elect to implement the iteration as a matrix multiply—something simpler
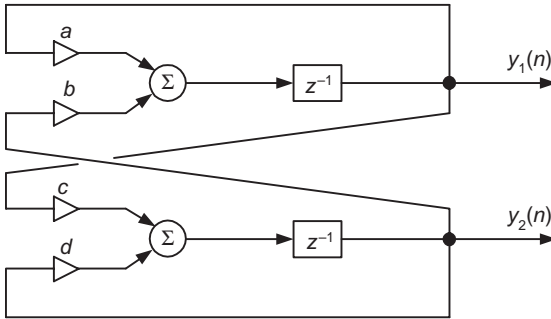
**Figure 20-2**    Generic oscillator iteration network diagram.

may be possible! So we will derive the network diagram that represents the general rotation matrix multiplication. In the world of signal processing, it is customary to try to use only addition, multiplication, and delays, and that is all we will use. The centerpiece of the network diagram in Figure 20-2 is the pair of delay elements that are interpreted to hold the state variables. For each iteration, the outputs of the delay elements are used as the past values and the inputs to the delay elements are the new values. For example, the input to the upper delay element in Figure 20-2 is calculated thus: $\hat{y}_1 = ay_1 + by_2$.

This generic form requires four multiplies and two additions for each iteration. If the rotation matrix has some values in common or some are simply zero or one, then the iteration computations may become simpler.

## 20.3   CATALOG OF OSCILLATORS

Now that we had gone through the theory of discrete-time recursive oscillators, we will now list some common oscillators along with their attributes. I have chosen five oscillators that span the gamut based on the number of multiplies per iteration and their type of outputs. All oscillators represented by 2-by-2 matrices will produce two sine waves simultaneously. These two sine waves will always have the same frequency and be out of phase with each other, and may have differing amplitudes. If the outputs are 90° out of phase, then we have a quadrature oscillator. Likewise, if the two sine wave outputs have the same amplitude, then we have an equi-amplitude oscillator. Four of the five oscillators in this catalog are in use in industry as they are mentioned in various application notes and trade journals. The quadrature oscillator with staggered update is one I put together using this matrix theory. Often you can make a new oscillator by permuting the matrix elements of a known oscil-lator. Just apply the Barkhausen criteria to the resulting matrix to see if they still hold. Also you can multiply an oscillator matrix by another matrix and often create another oscillator. So as you can gather, this catalog hardly exhausts the possible list of oscillators. In order to show the structure of an oscillator's matrix in its simplest form, the concept of tuning parameter is introduced. This parameter, $k$, is related to
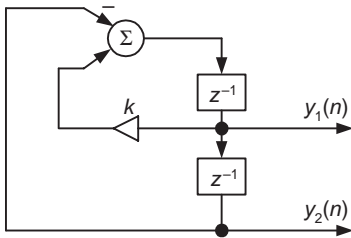
**Figure 20-3**    Biquad oscillator.

the step angle, $\theta$. The exact relation, which depends on the particular oscillator used, will be provided.

## 20.4  BIQUAD

The biquad oscillator was one of the first discrete oscillators to see use in signal processing applications. I recall an application patent issued in the 1980s that used this oscillator for generating call progress tones used in telephony. I found this interesting since François Viète discovered the trigonometric recurrence relation (20–1) long before. His result was published in the year 1571! The biquad oscillator has equi-amplitude outputs, which turn out to have a relative phase shift of $\theta$.

$$k = 2\cos(\theta) \tag{20–28}$$

$$\begin{bmatrix} k & -1 \\ 1 & 0 \end{bmatrix} \tag{20–29}$$

When the rotation matrix (20–29) elements are substituted in Figure 20-2, the generic oscillator network becomes the biquad oscillator shown in Figure 20-3.

## 20.5  DIGITAL WAVEGUIDE

The *digital waveguide oscillator* is the simplest (in terms of the number of multiplies) oscillator with quadrature outputs. For $k$ near zero, the outputs become nearly equal in amplitude. This means this oscillator can be effectively used to phase lock a signal near 1/4 the sampling rate. More on dynamic tuning (i.e., changing frequency while in operation) of oscillators later. Figure 20-4 shows the network form for the digital waveguide oscillator.

$$k = \cos(\theta) \tag{20–30}$$

$$\begin{bmatrix} k & k-1 \\ k+1 & k \end{bmatrix} \tag{20–31}$$

*Note*: The digital waveguide oscillator is a claimed item under U.S. patent #5701393, so consult the patent's owner before using this oscillator in a product.
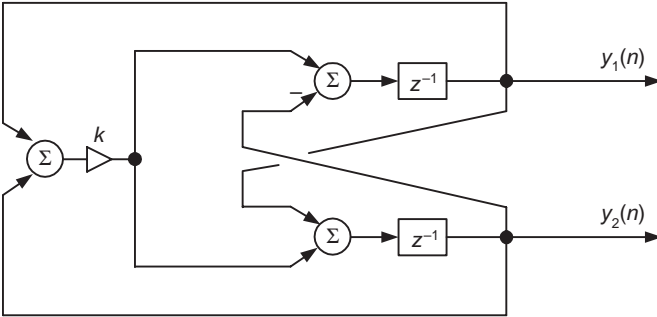
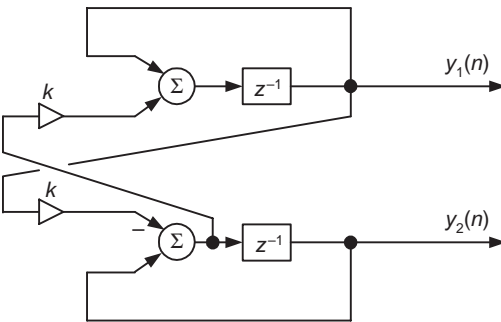**Figure 20-4**    Digital waveguide oscillator.



**Figure 20-5**    Equi-amplitude staggered update oscillator.

## 20.6  EQUI-AMPLITUDE STAGGERED UPDATE

The staggered update oscillators take their name from the fact that one state variable is first updated and then that new value is used in the update of the remaining variable. This oscillator's outputs are equi-amplitude and quasi-quadrature, with the quadrature relation being reached in the limit of small $k$. To explicitly show the oscillator iteration as being staggered, its matrix along with a factoring of the matrix is shown. Notice how the matrix factors into a product of two triangular matrices, neither of which can function as an oscillator alone. This oscillator is shown in Figure 20-5.

$$k = 2\sin\left(\frac{\theta}{2}\right) \qquad (20\text{--}32)$$

$$\begin{bmatrix} 1-k^2 & k \\ -k & 1 \end{bmatrix} = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -k & 1 \end{bmatrix} \qquad (20\text{--}33)$$

## 20.7  QUADRATURE STAGGERED UPDATE

This quadrature oscillator has nearly equi-amplitude outputs when $k$ is small. Again as before, we show the factoring of its matrix, but here we notice that the
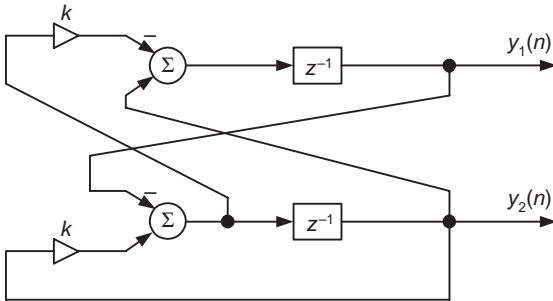
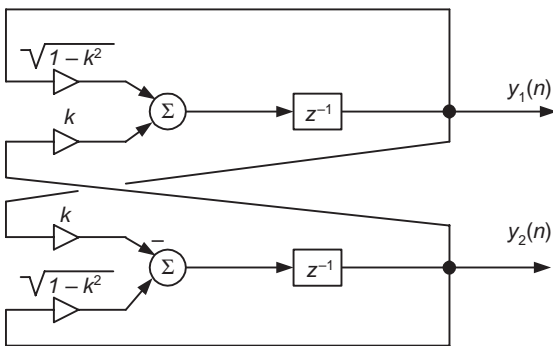**Figure 20-6** Quadrature staggered update oscillator.



**Figure 20-7** Coupled standard quadrature oscillator.

right-hand factor is effectively a biquad oscillator. So the left-hand factor is used to change the configuration of the right-hand oscillator. This oscillator is shown in Figure 20-6.

$$k = \cos(\theta) \tag{20–34}$$

$$\begin{bmatrix} k & 1-k^2 \\ -1 & k \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & k \end{bmatrix} \tag{20–35}$$

## 20.8 COUPLED STANDARD QUADRATURE

The coupled standard quadrature oscillator features both quadrature and equi-amplitude outputs. However, there is a cost—this oscillator requires four multiplies per iteration. This oscillator (like the biquad) may be derived directly from trigonometric formulas as shown in (20–3). Here the two trigonometric functions are written in terms of the single parameter, $k$. This oscillator is shown in Figure 20-7.

$$k = \sin(\theta) \tag{20–36}$$

**Table 20-1** Recursive Oscillator Properties

| Oscillator | Multiplies/iteration | Equi-amplitude | Quadrature output | $k =$ | Rotation matrix |
|---|---|---|---|---|---|
| Biquad | 1 | Yes | No | $2\cos(\theta)$ | $\begin{bmatrix} k & -1 \\ 1 & 0 \end{bmatrix}$ |
| Digital waveguide | 1 | No | Yes | $\cos(\theta)$ | $\begin{bmatrix} k & k-1 \\ k+1 & k \end{bmatrix}$ |
| Equi-amplitude staggered update | 2 | Yes | No | $2\sin(\theta/2)$ | $\begin{bmatrix} 1-k^2 & k \\ -k & 1 \end{bmatrix}$ |
| Quadrature staggered update | 2 | No | Yes | $\cos(\theta)$ | $\begin{bmatrix} k & 1-k^2 \\ -1 & k \end{bmatrix}$ |
| Coupled standard quadrature | 4 | Yes | Yes | $\sin(\theta)$ | $\begin{bmatrix} \sqrt{1-k^2} & k \\ -k & \sqrt{1-k^2} \end{bmatrix}$ |

$$\begin{bmatrix} \sqrt{1-k^2} & k \\ -k & \sqrt{1-k^2} \end{bmatrix} \tag{20–37}$$

Gathering our catalog of oscillator properties yields the data in Table 20-1.

## 20.9 DYNAMIC AMPLITUDE CONTROL

So far the oscillators we have described are ballistic in the sense that they are loaded with some values and allowed to free run. If the run time is short, then this may be adequate. But errors can and may accumulate to the point where the output no longer meets your requirements. Thus a way of controlling the oscillator's amplitude is needed. A standard approach uses automatic gain control (AGC). Since we are iterating the oscillator, why don't we just measure the strength of the output and correct it after each iteration? Also, if the errors are small, the corrections need to be only approximate.

Now when we are measuring the oscillator's output we will use powers (squares of the amplitude) instead of the amplitudes to avoid square roots. Square root calculations tend to be costly in a DSP. The instantaneous power, $P$, is given by the following formula (in terms of state variables, matrix elements, and phase shift):

$$P = \frac{x_1^2 - \dfrac{b}{c}x_2^2 - 2\sqrt{\dfrac{-b}{c}}\,x_1 x_2 \cos(\phi)}{\sin^2(\phi)} \tag{20–38}$$

This formula looks painful to implement, but for fixed frequencies, the calculation involves only six multiplies and two subtracts. However, looking ahead to being

able to change frequency, we see that if we restrict ourselves to the quadrature oscillators ($\phi = \pm 90°$), then the formula for the power becomes much simpler! It is just:

$$P = x_1^2 - \frac{b}{c} x_2^2 \qquad (20\text{--}39)$$

Along with the power measurement, we need to find the gain needed to properly scale the state variables. A general gain formula is:

$$G = \frac{P_0^q}{P^q} \qquad (20\text{--}40)$$

In this formula, $P$ is the measured power, $P_0$ is the set-point power, and $q$ is a convergence factor. Since we'd rather not perform division, we will use the first-order Taylor's approximation—it has the neat property of turning division into subtraction. It is:

$$G \approx 1 + q - q\frac{P}{P_0} \qquad (20\text{--}41)$$

Since we are using $G$ to scale the amplitudes (the state variables), it is best to let $q = 1/2$. Also, when using fixed-point DSPs it becomes convenient to let the set-point power also be 1/2 (amplitude $\approx 0.707$). Thus our correction formula becomes

$$G = \frac{3}{2} - P \qquad (20\text{--}42)$$

But we still have one more potential problem, and that is $G$ will nominally be 1, and sometimes we will need to multiply the state variables by a number slightly greater than one. A trick that can be used is to multiply the state variables by $G/2$ and then double the results. So in summary, the AGC approach consists of the following steps per iteration:

1. Perform one oscillator iteration to update the state variables.
2. Measure the oscillator's output power, $P$.
3. Calculate a gain factor, $G$.
4. Scale the state variables by this gain factor.

## 20.10   DYNAMIC FREQUENCY CONTROL

Finally we get to the subject of dynamic frequency control. Since there are some applications where we would like to have a numerically controlled oscillator, we will briefly look at what it takes to do this with these oscillators. Changing an oscillator's frequency merely requires modifying the rotation matrix's $k$ value for any new $\theta$ frequency value. The difficult part, however, is maintaining amplitude control during dynamic frequency changes. Since our general power formula (20–

38) shows a dependence on both the frequency and amplitude ratio, this can prove computationally inefficient in the general case. Thus the problem is updating the coefficients in the power formula as the frequency is changed. Plus some oscillators will also change output amplitudes.

If we apply some restrictions to the type of oscillator, we can simplify the situation. If we look at just using a coupled–standard quadrature oscillator, all of these problems go away. In this case, the formula for the power becomes independent of the matrix elements altogether! However, the difficulty in this case lies in the matrix coefficients where two of them involve radicals. One can use a first-order binomial expansion for these two terms, but then the determinant is not quite one, so the AGC must make up for the error. The coupled–standard quadrature rotation matrix that uses first-order binomial expansions for the two terms with radicals is:

$$\begin{bmatrix} 1-\dfrac{k^2}{2} & k \\[2mm] -k & 1-\dfrac{k^2}{2} \end{bmatrix} \tag{20–43}$$

Equation (20–43), which can now be used to approximate (20–37), has a determinant that's nearly one (specifically, $1+k^4/4$), so the gain compensation works well for a wide range of $k$. But even if we use a non-equi-amplitude quadrature oscillator, we can find a simple solution for its amplitude control. In fact, we move the approximation in the process from the rotation matrix to the power measurement. An example using the digital waveguide oscillator will now be shown. The power formula for the digital waveguide oscillator in terms of the tuning parameter, $k$, is

$$P = x_1^2 - \frac{k-1}{k+1} x_2^2 \tag{20–44}$$

But knowing that we will be calculating the power on every iteration, and preferring to avoid division, we will use a first-order series expansion of the denominator that gives the following approximation for $P$:

$$P \approx x_1^2 + (1-k)^2 x_2^2 \tag{20–45}$$

This approximation works well when $k$ is small, and $k$ is small for frequencies near 1/4 the sampling rate. If the tuning range needs to be enlarged, a higher-order expansion may be used.

So far we have been making the oscillators easily controllable by using low-order series expansions for the difficult-to-calculate terms. But these approximations carry a price tag: that is, either we must use the oscillator in a narrow tuning range where $k$ is small, or we must use a higher-order approximation to accommodate a larger range of operation. This results from the limitation that the low-order approximation is good only over a finite range. Sometimes our problem won't really be one of needing a large range of operation, but rather we might need to operate over a narrow range where $k$ is centered around some non-zero value. This case is easily handled by the use of a frequency-translation matrix. This matrix, in effect, shifts the

center frequency of our oscillator to the point of interest. The oscillator's center frequency is taken to be the frequency where $k$ is zero and thus the approximations are exact at this frequency. The steps for one iteration of a dynamic frequency–amplitude controlled oscillator that uses a frequency translation matrix are as follows:

1. Perform one oscillator iteration using a tunable oscillator to update the state variables. This can be one of the aforementioned quadrature oscillators.
2. Update the state variables using the frequency-translation matrix. This matrix is simply the same as shown in (20–37), where $k$ is fixed to represent the translation (shift) frequency. Since the values in this matrix are interpreted to be constant, these values can be calculated prior to run time.
3. Measure the oscillator's output power, $P$. This is just applying (20–38) or one of its simpler incarnations to the state variables.
4. Calculate a gain factor, $G$.
5. And finally, scale the state variables by this gain factor.

It should be understood that there is only one pair of state variables involved in the previous set of steps. Steps 1, 2, and 5 operate on the one pair of values. And step 3 just uses the same pair to calculate the power. This can be viewed as two oscillators operating on the same state variables. It just happens that one oscillator allows for dynamic frequency control, and the other just shifts the frequency. The networks representing the oscillator iterations are still the same as previously cataloged. Explicitly we can write the translation (shift) matrix as:

$$\begin{bmatrix} \cos(\omega) & \sin(\omega) \\ -\sin(\omega) & \cos(\omega) \end{bmatrix} \tag{20–46}$$

In a practical implementation, you will set $\omega = 2\pi f_o/f_s$ where $f_o$ is the shift frequency in Hertz, and $f_s$ is the sample rate. The shift frequency may be different from the new desired center frequency, since the natural center frequency for some of the oscillators is one-fourth of the sampling rate.

So we've learned a recipe for recursive oscillator design:

1. Pick your step angle based on the desired oscillator frequency by using $\theta = 2\pi f/f_s$.
2. Select the desired oscillator network based on the properties in Table 20-1.
3. Define $k$ using its relation to $\theta$ in Table 20-1.
4. Determine network coefficients from the appropriate rotation matrix in Table 20-1.
5. Establish the initial conditions from (20–27).
6. Implement the oscillator using the target hardware environment to determine whether dynamic amplitude control is necessary.
7. If dynamic frequency control is being used, determine whether a translation matrix is needed.

## 20.11 FSK MODULATOR DESIGN EXAMPLE

For our example, we will highlight the design of a simple modulator for a 1200-baud FSK modem. The modem generates either 1300 or 2100 Hz depending on whether it is sending a zero or a one bit. We will let our sample rate be 8 kHz, which is common in telephony. Since the data rate does not divide evenly into the sample rate, a sample rate conversion will be needed. This also means the oscillator will generate intermediate frequencies other than 1300 and 2100 Hz. This modulator example will benefit from the use of a translation matrix as well.

Looking at our two frequencies, namely 1300 and 2100 Hz, we see that we can use a fixed-frequency oscillator at 1700 Hz and let the variable-frequency oscillator range between +400 and −400 Hz. So we will use two oscillator matrices in tandem where one has a fixed frequency and the other varies depending on whether we are sending a zero or a one. The first oscillator will function as the frequency-translation oscillator and hence uses (20–46) for its work. This oscillator will not change frequency during the modem's operation, so the two trigonometric constants can be calculated prior to use, and we won't be concerned with there being radicals.

The second oscillator, which changes frequency while in operation, will use the first-order binomial expanded version of the coupled standard quadrature oscillator (20–43). We are using this oscillator since it allows for easy frequency and amplitude control; remember the amplitude control is required here, somewhat for accumulated numerical errors, but mainly since this matrix (20–43) does not strictly obey the Barkhausen criteria. Fortunately, the AGC can more than compensate for the oscillator's gain as long as $k$ is small, which means low frequencies with this oscillator, and hence the use of the frequency translation in the overall modulator.

Since the input to the modem is a sequence of bits (1200 bps), we need to do several things before we let it control the oscillator's frequency. First we need to perform an antipodal mapping of the data. That is, map one bits to a value of +1, and map zero bits to −1. Next we need to resample these 1200 values per second to 8000 per second since this is the modem's sampling rate. Basically a multirate filter is used to interpolate by 20 and decimate by 3. The lowpass filter used in this process will be selected to offer ISI (intersymbol interference) rejection. A polyphase raised cosine filter will suffice. And finally, we will scale the input to the oscillator, so it will emit the correct frequencies. Since we are doing an FM process, the scaling just sets the modulation index. The scaling can be combined into the sample rate conversion process. If a polyphase filter method is used, the coefficients for each of the filters just get scaled to set the modulation index.

The fixed-frequency matrix is set to operate at 1700 Hz (i.e., the average of the two desired frequencies). Numerically, the fixed-frequency matrix is just:

$$\begin{bmatrix} \cos\left(\dfrac{2\pi 1700}{8000}\right) & \sin\left(\dfrac{2\pi 1700}{8000}\right) \\ -\sin\left(\dfrac{2\pi 1700}{8000}\right) & \cos\left(\dfrac{2\pi 1700}{8000}\right) \end{bmatrix} \approx \begin{bmatrix} 0.233445 & 0.972370 \\ -0.972370 & 0.233445 \end{bmatrix} \quad (20\text{–}47)$$

Now for the variable-frequency part: Since we desire to deviate between +400 and −400 Hz, we find the scaling for the frequency input to the oscillator by $k = \sin(2\pi 400/8000) = 0.309017$. So our ISI-filtered antipodal values need to range between $\pm 0.309017$. The frequency input, which is updated 8000 times per second, becomes the value $k$ used in (20–43). The two unique values in matrix (20–43) are calculated 8000 times per second and then used in the variable-frequency oscillator iteration. And of course the initial state values for the oscillator combination are simply:

$$\begin{bmatrix} \dfrac{\sqrt{2}}{2} \\ 0 \end{bmatrix} \approx \begin{bmatrix} 0.707107 \\ 0 \end{bmatrix} \tag{20–48}$$

These are found by scaling the result of (20–27) by sqrt($P_0$), and $P_0$ is chosen so (20–42) may be used for amplitude control.

For each iteration then, we just

1. Perform antipodal mapping, resample, use ISI rejection filter, and scale the 1200 bps data to create "$k$" for this iteration.
2. "Matrix multiply" the state variables by the 1700 Hz fixed-frequency matrix.
3. "Matrix multiply" the state variables by the variable-frequency matrix—the value of $k$ was determined above.
4. Measure the power: $P = (x_1)^2 + (x_2)^2$.
5. Calculate the gain: $G = (3/2) - P$.
6. Scalar multiply the state variables by the gain, $G$.

## 20.12 CONCLUSIONS

We have explored the basic theory of recursive digital oscillators with a bent toward the practical, and from there we have looked at some common oscillators. Then we added some mechanisms for controlling their amplitude and adjusting their frequency. Finally, we showed a brief example of how these oscillators and their control mechanisms may be used to make FSK modulators. I hope I have piqued your interest, and I encourage you to go and develop your own oscillators using the rules and techniques presented here.

## 20.13 REFERENCES

[1] J. DIEFENDERFER, *Principles of Electronic Instrumentation*. Saunders College Publishing, Philidelphia, 1979, p. 185.
[2] M. FRERKING, *Digital Signal Processing in Communication Systems*. Kluwer Academic Publishers, Norwell, MA, 1993, pp. 214–217.
[3] S. FRIEDBERG and A. INSEL, *Introduction to Linear Algebra with Applications*. Prentice Hall, Englewood Cliffs, NJ, 1986, pp. 253–276.

[4]  R. HIGGINS, *Digital Signal Processing in VLSI*. Prentice Hall, Englewood Cliffs, NJ, 1990, pp. 529–532.

[5]  S. LEON, *Linear Algebra with Applications*, 2nd Ed. MacMillan, New York, 1986, pp. 230–259.

[6]  A. OPPENHEIM and R. SCHAFER, *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 342–344.

[7]  J. SMITH and P. COOK, "The Second Order Digital Waveguide Oscillator," *International Computer Music Conference*, San Jose, CA, October 1992, pp. 150–153.

[8]  SMITH III, et al. "System and Method for Real Time Sinusoidal Signal Generation Using Waveguide Resonance Oscillators," U.S. Patent #5701393, December 23, 1997.

[9]  C. TURNER, "A Discrete Time Oscillator for a DSP Based Radio," *SouthCon/96 Conference Record*, Orlando, FL, IEEE 1996, pp. 60–65.

## EDITOR COMMENTS

To elaborate on the very useful Figure 20-7 coupled standard quadrature oscillator in a fixed-frequency mode of operation, with $k = \sin(\theta)$ and knowing that $\cos^2(\theta) + \sin^2(\theta) = 1$ we can write

$$\sqrt{1 - k^2} = \cos(\theta).$$

Thus we can redraw the oscillator as shown in Figure 20-8(a). In that figure we have included the implementation of the automatic gain control (AGC) discussion in Section 20.9. Because $b = \cos(\theta)$ and $c = -\cos(\theta)$, (20–39) becomes

$$P(n) = x_1^2(n) - \frac{b}{c} x_2^2(n) = x_1^2(n) + x_2^2(n)$$
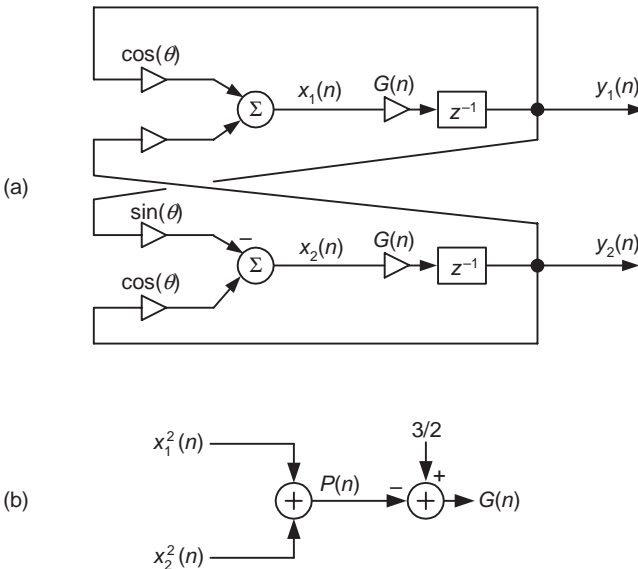


**Figure 20-8**   Coupled standard quadrature oscillator with AGC.

and the amplitude correction factor in (20–42) becomes

$$G(n) = \frac{3}{2} - P(n) = \frac{3}{2} - [x_1^2(n) + x_2^2(n)].$$

The computation of the $G(n)$ amplitude correction factor is shown in Figure 20-8(b).

# Chapter 21

# Direct Digital Synthesis: A Tool for Periodic Wave Generation

**Lionel Cordesses**

*Technocentre, Renault*

**D**iscrete-time oscillators are the subject of intensive research. From Colpitts oscillators (Chapter 7 of [1]) to phase-locked loops [2], methods have been proposed to improve stability, frequency resolution, and spectral purity. Among the all-digital approaches, such as the one presented in [3], *direct digital frequency synthesis* (DDS) appeared in 1971 [4]. Three years later, this technique was embedded in a commercial unit measuring group delay of telephone lines [5]. DDS is now available as integrated circuits and it outputs waveforms up to hundreds of megahertz.

While DDS is slowly gaining acceptance in new system designs, methods used to improve the quality of the generated waveform are seldom used, even nowadays. The purpose of this chapter is to give an overview of the basics of DDS, along with simple formulas to compute bounds of the signal characteristics. Moreover, several methods—some patented—are presented to overcome some of the limits of the basic DDS with a focus on improving output signal quality.

## 21.1 DIRECT DIGITAL SYNTHESIS: AN OVERVIEW

The DSP operation we want to perform is to generate a periodic, discrete-time waveform of known frequency $F_o$. The waveform may be a sinewave, as in [3]. It can also be a sawtooth wave, a triangle wave, a squarewave, or any periodic

waveform. We will assume that the sampling frequency $F_s$ is known and constant. Before proceeding with the theory of operation, we summarize why direct digital synthesis is a valuable technique:

- The tuning resolution can be made arbitrarily small to satisfy almost any design specification.
- The phase and the frequency of the waveform can be controlled in one sample period, making phase modulation feasible.
- The DDS implementation relies on integer arithmetic, allowing implementation on virtually any microcontroller.
- The DDS implementation is always stable, even with finite-length control words. There is no need for an automatic gain control.
- The phase continuity is preserved whenever the frequency is changed (a valuable tool for tunable waveform generators).

## 21.2  THEORY OF OPERATION AND IMPLEMENTATION

The implementation of DDS is divided into two distinct parts as shown in Figure 21-1; a discrete-time *phase generator* (the accumulator) outputting a phase value *ACC*, and a *phase-to-waveform converter* outputting the desired DDS signal.

### From a Sampling Frequency to a Phase

The implementation of the DDS relies on integer arithmetic. The size of the accumulator (or *wordlength*) is $N$ bits. Assuming that the period of the output signal is $2\pi$ radians, the maximum phase is represented by the integer number $2^N$. Let us denote $\Delta_{ACC}$ the phase increment related to the desired output $F_o$ frequency. It is coded as an integer number with $N-1$ bits.

During one sample period $T_s$, the phase increases by $\Delta_{ACC}$. It thus takes $T_o$ to reach the maximum phase $2^N$:

$$T_o = \frac{1}{F_o} = \frac{2^N T_s}{\Delta_{ACC}} \qquad (21\text{--}1)$$

We can rewrite (21–1) in terms of frequency $F_o$, as a function of $\Delta_{ACC}$:
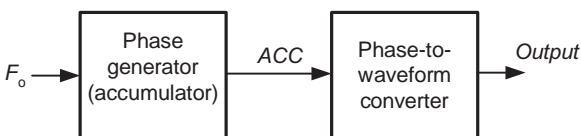


**Figure 21-1**  Fundamental DDS process.

$$F_o = F_o(\Delta_{ACC}) = \frac{F_s}{2^N}\Delta_{ACC}. \tag{21-2}$$

The phase increment $\Delta_{ACC}$, rounded to the nearest integer ($\lfloor x \rfloor$ is the integer part of $x$), is given by:

$$\Delta_{ACC} = \left\lfloor F_o \frac{2^N}{F_s} + 0.5 \right\rfloor. \tag{21-3}$$

Equation (21–2) is the basic equation of any DDS system. One can infer from (21–2) the tuning step $\Delta F_{o,min}$, which is the smallest step in frequency that the DDS can achieve (remember that $\Delta_{ACC}$ is an integer).

$$\begin{aligned} \Delta F_{o,min} &= F_o(\Delta_{ACC}+1) - F_o(\Delta_{ACC}) \\ &= \frac{F_s}{2^N}(\Delta_{ACC}+1-\Delta_{ACC}) \\ &= \frac{F_s}{2^N}. \end{aligned} \tag{21-4}$$

Equation (21–4) allows the designer to choose the number of bits ($N$) of the accumulator $ACC$. This number $N$ is often referred to as the frequency tuning wordlength [6]. It is reckoned thanks to:

$$N = \left\lfloor \log_2\left(\frac{F_s}{\Delta_{F_{o,min}}}\right) + 0.5 \right\rfloor. \tag{21-5}$$

The minimum frequency, $F_{o,min}$, the DDS can generate is given by (21–2) with $\Delta ACC=1$, the smallest phase increment that still increases the phase ($\Delta_{ACC}=0$ does not increase the phase). $F_{o,min}$ is

$$F_{o,min} = \frac{F_s}{2^N}. \tag{21-6}$$

The maximum frequency $F_{o,max}$ the DDS can generate is given by the uniform sampling theorem (Nyquist, Shannon; see, for instance, Chapter 9 of [7]):

$$F_{o,max} = \frac{F_s}{2}. \tag{21-7}$$

From a practical point of view, a lower $F_{o,max}$ is often preferred, $F_{o,max}=F_s/4$, for example. The lower $F_{o,max}$ is, the easier the analog reconstruction using a lowpass filter.

## From a Phase to a Waveform

The phase is coded with $N$ bits in the accumulator. Thus, the waveform can be defined with up to $2^N$ phase values. In case $2^N$ is too large for a realistic implementation, the phase-to-amplitude converter uses fewer bits than $N$. Let us note $P$ as the

number of bits used as the phase information (with $P \leq N$). The output waveform values can be stored in a lookup table (LUT) with $2^P$ entries: The output value is computed as $Output = LUT(ACC)$, which is implemented in the phase-to-waveform converter in Figure 21-1; other output waveform generation techniques, based on approximations, are presented later.

DDS can generate a sinewave with an offset $b$ and a peak amplitude $a$. The content of the LUT, containing the DDS output values, is computed for the index $i$ ranging from 0 to $(2^P - 1)$ using:

$$LUT(i) = \left\lfloor b - a\sin\left(\frac{2\pi i}{2^P}\right) + 0.5 \right\rfloor. \qquad (21\text{--}8)$$

Using the LUT computed for $P=9$, $a=127.5$, and $b=127.5$, the output waveform for $F_s=44100\,\text{Hz}$ and $F_o=233\,\text{Hz}$ is plotted as the solid curve in Figure 21-2.

One might want to generate two quadrature signals: One just has to read both $LUT(i)$ and $LUT(i+2^P/4)$, which in turn correspond to the sine and to the cosine functions.

A squarewave can be had with no computational overhead because that waveform is already available as the most significant bit of the phase accumulator $ACC$, as shown by the dashed curve in Figure 21-2. The most significant bit toggles every $\pi$ radians, since the accumulator represents $2\pi$ radians. We must point out that this squarewave is corrupted by phase jitter [8] of one sampling period $T_s$. This phase jitter is caused by the sampling scheme used to synthesize the waveform. To quote [38]:

> [T]he output of the direct digital synthesizer can occur only at a clock edge. If the output frequency is not a direct submultiple of the clock, a phase error between the ideal output and the actual output slowly increases (or decreases) until it reaches one clock period, *at which time the error returns to zero and starts to increase (or decrease) again.*

A sawtooth signal is also available with no computational overhead. The linearly increasing phase accumulator $ACC$ value is stored modulo $2^N$, thus leading to a sawtooth signal as shown by the dotted curve in Figure 21-2. The LUT is not used in this case, or it is the identity function: $Output = ACC$. With the use of logic gates, a triangular output waveform can be generated from the sawtooth.
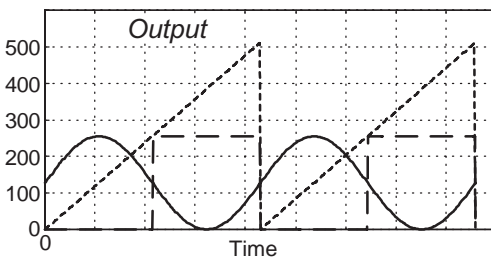


**Figure 21-2**    Signals generated by software DDS: sinewave, squarewave, and sawtooth signals.

## 21.3 QUANTIZATION EFFECTS

Quantization occurs both on the *ACC* phase information and on the *Output* amplitude information. The DDS is now redrawn including this effect. The number of bits used by each variable is written below the variables on Figure 21-3.

### Phase Quantization

Phase quantization occurs when the phase information *ACC* is truncated from *N* bits to *P* bits as shown in Figure 21-3. The reason behind this quantization is to keep the memory requirements of the phase-to-waveform converter quite low: When implemented as a LUT, the size of the memory is $2^P \times M$ bits. A realistic value for *N* is 32, but this would lead to a $2^{32} \times M$ memory, which is not realistic. Thus we quantize the phase information $\Phi$ to *P* bits, as it decreases the number of entries of the LUT.

Unfortunately, the phase quantization introduces noise on the phase signal $\Phi$. It leads to *phase noise* (see Chapter 7 of [1], and Chapter 3 of [9]) and it produces unwanted spurious spectral components in the DDS output signals, often referred to as *spurs*. The difference between the carrier level (which is the desired signal) and the maximum level of spurs is called *spurious free dynamic range* (SFDR). A simplified formula given in [10] to estimate the maximum level of the spurs $S_{max}$ when the carrier level is 0 dB, is:

$$S_{max} = -\text{SFDR} = -6.02P + 3.92 \, \text{dB}. \qquad (21\text{--}9)$$

For a detailed derivation of the exact formulas (including the frequency and the SFDR of spurs), the reader is referred to [11] and [12].

### Amplitude Quantization

The output of the phase-to-waveform converter is quantized to *M* bits, *M* being the wordlength of the *Output* amplitude word. This quantization results in a signal-to-noise ratio (in this case, it is a noise-to-signal ratio) [10] usually approximated by:

$$SNR = -6.02M - 1.76 \, \text{dB} \qquad (21\text{--}10)$$

This bound also limits the performance of the DDS, as the output spectrum will exhibit a $-6.02M - 1.76$ dB noise floor. Thanks to (21–9) and (21–10), one can infer (see [10]):
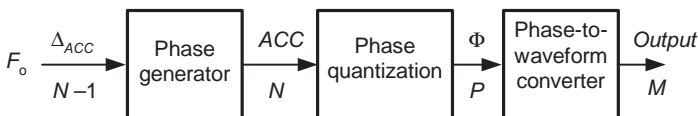


**Figure 21-3** DDS including quantization.

$$P = M + 1. \tag{21-11}$$

Thus, $S_{max} = -6.02(M+1) + 3.992\,dB = -6.02\,M - 2.028\,dB$ and $SNR = -6.02\,M - 1.76\,dB$ leading to $S_{max} < SNR$. This inequality means that the unwanted signals are caused by the amplitude quantization, and not by the phase truncation. Knowing (21–11), we can now focus on improving the SFDR of a DDS.

## 21.4  IMPROVING SFDR BY SINEWAVE COMPRESSION

There are many techniques to improve the SFDR of a DDS. The easiest one would be to increase the phase wordlength. Due to (21–9) and (21–10), we can increase $P$ (and thus $M$ according to (21–11) in order to meet the technical specifications. The only drawback of this approach is the total amount of LUT memory, $2^P \times M$ bits. For small $P$ (such as $P=9$ bits, and $M=8$ bits), implementing the LUT with a memory leads to simple, low-cost hardware; see [13] and [8] for a realization based on this method.

For higher values of $P$, the memory requirements become impractical at high frequency or for embedded system implementations. To circumvent this impediment, the solution is to compress the sine waveform, thus reducing memory consumption. Two methods are reported in the next sections. One is based on symmetry and the other on sinewave approximations.

### A Quarter of a Sinewave

Instead of storing the whole sinewave $f(\Phi) = \sin(\Phi)$ for $0 \leq \Phi \leq 2\pi$, one can store the same function for $0 \leq \Phi \leq \pi/2$ and use symmetry to get the complete $2\pi$ waveform range. This approach uses only $2^{P-2}$ entries in the LUT, leading to a LUT-size compression ratio of $4:1$. The full sinewave can be reconstructed at the expense of some hardware (see [5], [14], and [9]). From here out, we will deal with only a quarter of a sinewave. Next we discuss four methods of approximating a sinewave.

### Sinewave Approximations

The first sinewave approximation method goes as follows: Instead of storing $f(\Phi) = \sin(\Phi)$ using $M$ bits, one can store $g(\Phi) = \sin(\Phi) - 2\Phi/\pi$, hence the name *sine-phase difference algorithm* found in [14]. It has been shown in [14] that this new function $g$ only needs $M-2$ bits to get the same amplitude quantization for the sinewave (see Figure 21-4 for an example). The only drawback is the need for an adder at the output of the LUT.

The second sinewave approximation method is called the *Sunderland technique*. This method, named after its author [15], makes use of trigonometric identities. It has been used for $P=12$ and it uses the following identity:
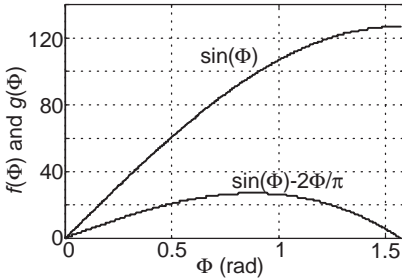
**Figure 21-4**    Sine-phase difference LUT example ($P=9$, $M=8$).

$$\sin(A+B+C) = \sin(A+B)\cos(C) + \cos(A)\cos(B)\sin(C) \\ - \sin(A)\sin(B)\sin(C). \tag{21--12}$$

The 12 bits of the phase are :

- $A$, the four most significant bits (with $0 \leq A \leq \pi/2$)
- $B$, the following four bits (with $0 \leq B \leq (\pi/2)/(2^4)$)
- $C$, the four least significant bits (with $0 \leq C \leq (\pi/2)/(2^8)$)

Equation (21–12) is then approximated by:

$$\sin(A+B+C) \approx \sin(A+B) + \cos(A)\sin(C). \tag{21--13}$$

Using two LUTs (one for $\sin(A+B)$ and one for $\cos(A)\sin(C)$) leads to a significant amount of compression. The $\sin(A+B)$ LUT uses $2^8 \times 11$ bits ($P=12$, thus $M=P-1=11$). The second LUT is filled with small numbers, thus requiring less than $M$ bits (actually four bits; see [15]). Finally, the compression ratio of this architecture is $51:1$ (see [16] for a comparison of various compression methods).

Several improvements to this architecture have been presented (see [14]) and the compression ratio of the modified Sunderland technique leads to a $59:1$ compression ratio [16]. The same method has been used in [17] with a $128:1$ compression ratio, and in [18] with a $165:1$ compression ratio.

The third sinewave approximation method involves first-order Taylor series expansions. Let us introduce $\delta_\Phi$ with $\delta_\Phi \ll \Phi$. The Taylor series expansion of the sine function is:

$$\sin(\Phi + \delta_\Phi) \approx \sin(\Phi) + \delta_\Phi \cos(\Phi). \tag{21--14}$$

Instead of storing the sine function, the key idea presented in [19] and described in [20] proposes to use two coarse LUTs storing $\sin(\Phi)$ and $\cos(\Phi)$. Moreover, the sine-phase difference algorithm can be used to store efficiently $\sin(\Phi)$, further decreasing the size of the LUT. The compression ratio obtained with this method is $64:1$ [19] and even reaches $67:1$ [16].

Another method has been introduced in [21] where the sine function is approximated by linear interpolation. A few samples (16 in [22]) of the sine function are

stored in a LUT, and the values are computed using linear interpolation. The compression ratio, computed thanks to data given in [22], is $10 \times 2^{11}/960 \approx 21:1$. Another implementation of the linear solution does not rely on a LUT. Using notations as in [23], the sine function is written as:

$$\sin(\Phi + \delta_\Phi) \approx y_0 + m_0(\Phi - \delta_\Phi). \qquad (21\text{–}15)$$

The solutions presented in [24] and used in [23] carefully impose a power-of-two number of segments, thus using the most significant bits of $\delta_\Phi$ as an address. Moreover, to further decrease complexity, the length of all the segments are equal. The implementation of (21–15) uses only one multiplication and one addition, without any complex address decoder. There is no LUT in this design. According to their authors, such an approach reaches the performance of other methods, showing a 60 dBc SFDR for $P = 12$ bits phase [23]. The method is now patented [25].

The fourth sinewave approximation method involves higher-order Taylor series expansions. In [26], the Taylor series expansion of the sine function is:

$$\sin(\Phi + \delta_\Phi) \approx \sin(\Phi) + \delta_\Phi \cos(\Phi) - (\delta_\Phi)^2 \sin(\Phi)/2. \qquad (21\text{–}16)$$

The compression ratio, given in [27], is $110:1$ for $P = 12$ bits and a SFDR of 85 dBc. Higher-order interpolation is also used for sinewave compression: Parabolic interpolation is presented in [28]. Only interpolation coefficients $V_1$, $V_2$, and $V_3$ are stored in the LUT, and the value of the sine function is reckoned thanks to:

$$\sin(\Phi + \delta_\Phi) \approx V_1(\Phi) + \delta_\Phi V_2(\Phi) + (\delta_\Phi)^2 V_3(\Phi). \qquad (21\text{–}17)$$

The compression ratio obtained using (21–17), given in [28], for a 64 dBc SFDR at $M = 11$ bits is $157:1$. As in the previous first-order Taylor series method, there is a counterpart of the LUT-less method. Based on a quadruple angle equality:

$$\cos(4\Phi) = 1 - 8\sin^2(\Phi)[1 - \sin^2(\Phi)]. \qquad (21\text{–}18)$$

With $0 \leq \Phi \leq \pi/8$, (21–18) is approximated in [29] by:

$$\cos(4\Phi) \approx 1 - 8\Phi^2(1 - \Phi^2). \qquad (21\text{–}19)$$

Equation (21–19) is implemented with multipliers and adders only.

There are still other approaches to approximating a sinewave. A phase-to-sinewave converter can be implemented thanks to an angle-rotation algorithm, such as the CORDIC (*COordinate Rotation DIgital Computer*) algorithm [30]. A DDS based on this method is described in [31], and the effect of finite precision on the CORDIC converter is analyzed in [32]. Another method relies on a nonlinear digital-to-analog converter that implements the sinewave generation [33].

There are trade-offs with each of the above sinewave approximation techniques, and no single technique is best for all DDS applications. In what follows, we discuss additional tricks used to optimize DDS performance by maximizing SFDR.

## 21.5 IMPROVING SFDR THROUGH SPUR-REDUCTION TECHNIQUES

The easiest method to reduce the level of DDS spurs, discussed in Part 21.4, is to increase the accuracy of the phase-to-waveform converter. The limit of this approach has been mentioned; it is mainly technological (lookup table size).

We now review three simple and effective methods to reduce the spur level of the sinewave DDS, along with the corresponding spectra computed from simulated DDS outputs.

### The Odd Number Approach

The worst-case spur level is given by (21–9). Making $\Delta_{ACC}$ an odd number improves the SFDR by 3.9 dB [14]. The repetition period of the accumulator $T_{ACC}$, often referred to as grand repetition period, is given by:

$$T_{ACC} = \frac{2^N}{GCD(2^N, \Delta_{ACC})} \tag{21–20}$$

with $GCD(x,y)$ standing for greatest common divisor of $x$ and $y$ [34]. When $GCD(2^N, \Delta_{ACC}) = 1$, as it will be whenever $\Delta_{ACC}$ is an odd number, then $T_{ACC} = 2^N$, which *spreads* the spurs over the entire spectrum (otherwise they are aliased to a frequency within the spectrum, as described in [14] and [12]). As an example, we computed the output spectra with a fast Fourier transform (FFT), the length of which is equal to $T_{ACC}$, for the two cases of $\Delta_{ACC} = 13248$ and for $\Delta_{ACC} = 13249$, with $N=16$, $P=9$, and $M=8$. The odd number for $\Delta_{ACC}$ leads to an increase of $54.2 - 50.3 = 3.9$ dB in SFDR.

### The Phase Dithering Approach

In another method to spread the spurs throughout the available bandwidth, one can add a dither signal [35] to the *ACC* phase values as shown in Figure 21-5.

The dither signal can be a pseudorandom noise sequence (generated, for example, with binary shift registers and exclusive-or gates, and having a repetition period much greater than the output signal period) whose word width is $B$ bits providing noise values in the range of 0 and $2^B$. Choosing $B = N - P$, the spurs do follow a
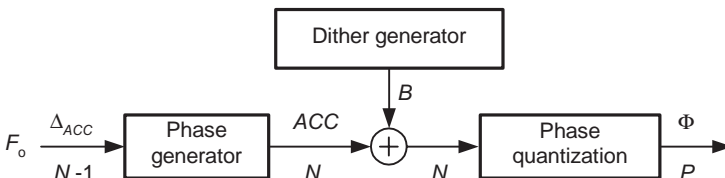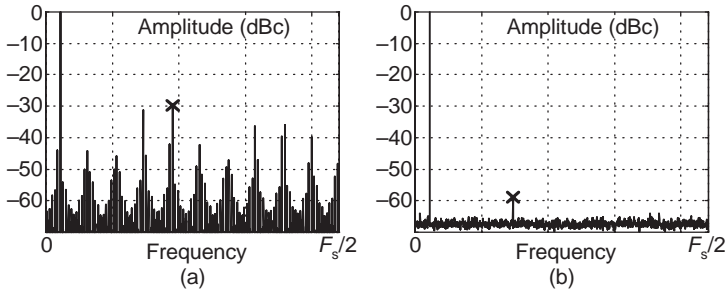


**Figure 21-5**   DDS including phase dither.

**Figure 21-6**    DDS output spectra: (a) without dithering; (b) with dithering.

12-dB-per-phase-bit law [36], instead of the 6 dB per phase bit of (21–9), thus allowing a smaller LUT for the same SFDR.

An output spectrum is given without any dither signal in Figure 21-6(a) with $F_s = 44100 \, \text{Hz}$, $\Delta_{ACC} = 1657$, $N = 16$, $P = 5$, and $M = 16$, and with the dither signal ($B = N - P = 16 - 5 = 11$ bits) applied in Figure 21-6(b). The spectra have been computed for 10 output signals that have been averaged following the method described in [36]. The high-resolution, 16-bit LUT has been chosen so as to focus only on the 5 bits of phase quantization, as in [36]. The drawback of this dithering method is the increase of the noise floor, but that's a small price to pay for such a large increase in SFDR.

Other dithering methods are available, such as amplitude dithering, and both phase and amplitude dithering (see [36] and [37]). The phase dithering approach has also been applied to squarewave signal DDS [38].

## The Noise Shaping Approach

The key idea of the noise shaping approach, to improve our DDS SFDR, is to filter out the quantization noise introduced by the phase quantization step in Figure 21-3. This quantization can be viewed as a special case of noise addition [39], as depicted in Figure 21-7(a). The quantization noise signal $n$ can be recovered from the following equations:

$$\Phi = n + ACC, \text{ and } e_Q = ACC - \Phi. \tag{21–21}$$

Thus $e_Q = -n$. In the *noise shaping* approach, this quantization noise signal $-n$ is fed back, through a filter G, to the *ACC* signal as shown in Figure 21-7(b). The transfer function of interest is the one from $n$ to $\Phi$, as the noise added to the phase signal $\Phi$ will eventually lead to phase noise. The phase signal is then:
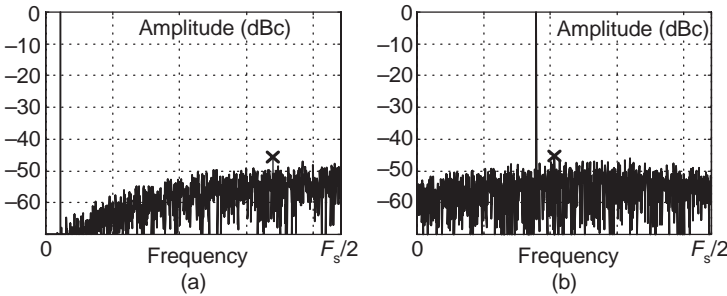
$$\Phi = n(1 - G) + ACC. \tag{21–22}$$

From (21–22), one can infer that the phase signal $\Phi$ is affected by the filtered noise signal $n$, $(1 - G)$ being the transfer function of the filter. The choice of G will lead to different results, as we shall see.
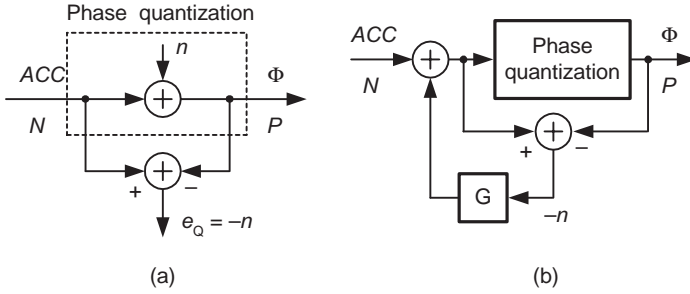
**Figure 21-7** Quantization and noise shaping: (a) quantization model; (b) noise shaping implementation.



**Figure 21-8** DDS spectra with first-order noise shaping and $G = z^{-1}$: (a) $ACC = 1657$; (b) $\Delta ACC = 13249$.

A *first-order noise shaping* approach is to use the simple transfer function $G$ proposed in [10], being the finite impulse response (FIR) filter $G = z^{-1}$. Here $z$ is the symbol of the $z$-transform used for discrete-time systems. The function $G$ can be implemented as a single delay register, and $1 - G$ has a zero at $z = 0$ (zero Hz): It acts as a discrete time differentiator. The system filters out the low-frequency components of the noise signal $n$, but high-frequency signals, greater than 8 kHz, are amplified. This simple approach prevents the filter from rejecting high-frequency components of the noise signal $n$, thus justifying the statement that one should use this filter for low-frequency $F_o$ signals [40].

An example of the output spectrum is given in Figure 21-8(a) (the parameters are the same as in Figure 21-6(a)). The SFDR is greater than 60 dB near the carrier, but it decreases with the frequency, and eventually reaches 46.8 dBc. The overall behavior is compliant with the above $G = z^{-1}$ analysis. At a higher frequency ($\Delta_{ACC} = 13249$), the noise close to the carrier is less filtered, as one can see in Figure 21-8(b).

To implement *higher-order noise shaping*, a more complex filter can be used instead of $G = z^{-1}$. A second-order FIR filter has been proposed in [41]:

$$1 - G = 1 + b_1 z^{-1} + b_2 z^{-2}. \tag{21–23}$$

Careful choice of $b1$ and $b2$ can lead to a double zero at zero: $1-G=1-2z^{-1}+z^{-2}=(1-z^{-1})^2$, which improves the rejection at zero Hz. When the noise shaping is applied to the amplitude signal instead of the phase signal, other values (often integer values, so as to ease implementation, see [41]) are preferred, and this filter can even be tuned online.

Multiple-zeros filters are also of interest, for example, when one wants to reject a known frequency such as $2 \times F_o$ [40]. A tunable notch filter is added at the expense of a more complex feedback structure. The transfer function becomes:

$$1-G = (1-z^{-1})(1+bz^{-1}+z^{-2})$$
$$= 1-(1-b)z^{-1}+(1-b)z^{-2}-z^{-3} \tag{21–24}$$

with $b=-2\cos[2\pi(2F_o/F_s)]$. At low frequencies, as shown in Figure 21-9(a), the spectrum looks like the one with the first-order noise shaping. But at a higher frequency (the same as on Figure 21-8(b)), the improvement close to the carrier is clear as shown in Figure 21-9(b).

Note that the same filter (with a zero at a specific frequency) can be implemented by feeding the error signal back to the accumulator. This structure, proposed and patented in [40], is presented in Figure 21-10. $F$ is the transfer function of the accumulator (an integrator: $ACC(k)=ACC(k-1)+\Delta_{ACC}(k-1)$), given by:

$$F = \frac{z^{-1}}{1-z^{-1}} \tag{21–25}$$
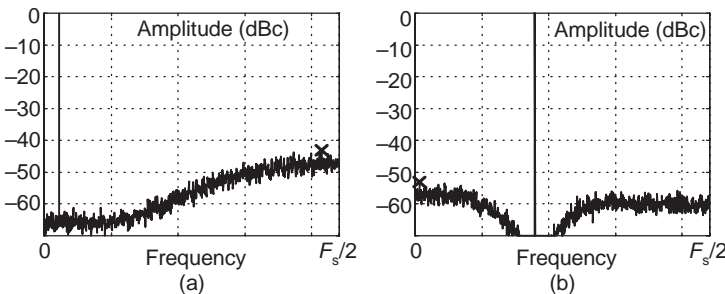


**Figure 21-9**    DDS spectra with higher-order noise shaping: (a) same parameters as Figure 21-6(a); (b) same parameters as Figure 21-8(b).
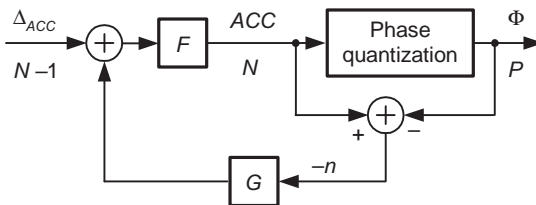


**Figure 21-10**    Noise shaping: Analog Devices' approach.

with the output phase $\Phi$ given by $\Phi = (1 - FG)n + F\Delta_{ACC}$. One can chose $G$ so as to ensure $(1 - FG) = (1 - z^{-1})(1 + bz^{-1} + z^{-2})$ as in (24).

## 21.6  CONCLUSIONS

Direct digital synthesis is a useful tool for generating periodic waveforms. We presented the basic idea of this synthesis technique, and then focused on the quality of the sinewave a DDS can create, introducing the SFDR quality parameter. Next we presented effective methods to increase the SFDR through sinewave approximations, hardware schemes such as dithering and noise shaping, and an extensive list of references. When the desired output is a digital signal, the signal's characteristics can be accurately predicted using the formulas given in this chapter. When the desired output is an analog signal, the reader should keep in mind that the performance of the DDS is eventually limited by the performance of the digital-to-analog converter and the follow-on analog filter [42].

   We hope that this discussion will incite engineers to use DDS, either integrated circuits DDS or software-implemented DDS. From the author's experience, this technique has proved valuable when frequency resolution is the challenge, particularly when using low-cost microcontrollers.

## 21.7  REFERENCES

[1]  U. ROHDE, J. WHITAKER, and T. T. N. BUCHER, *Communications Receivers*, 2nd ed, ISBN 0-07-053608-2. McGraw Hill, 1997.

[2]  U. ROHDE, *Digital PLL Frequency Synthesizers: Theory and Design*, ISBN 0-13-214239-2. Prentice-Hall, 1983.

[3]  C. TURNER, "Recursive Discrete-Time Sinusoidal Oscillators," *IEEE Signal Processing*, vol. 20, no. 3, May 2003, pp. 103–111.

[4]  J. TIERNEY, C. RADER, and B. GOLD, "A Digital Frequency Synthesizer," *IEEE Transactions on Audio and Electroacoustics*, vol. 19, no. 1, March 1971, pp. 48–57.

[5]  D. Guest, "Simplified Data-Transmission Channel Measurements," *Hewlett-Packard Journal*, November 1974, pp. 15–24.

[6]  *A Technical Tutorial on Digital Signal Synthesis*. Analog Devices, Inc., 1999.

[7]  H. BLINCHIKOFF and A. ZVEREV, *Filtering in the Time and Frequency Domains*. Noble Publishing, 2001.

[8]  E. MCCUNE, "Create Signals Having Optimum Resolution, Response, and Noise," *EDN*, March 14 1991, pp. 95–108.

[9]  J. CRAWFORD, *Frequency Synthesizer Design Handbook*, ISBN 0-89006-440-7. Artech House, 1994.

[10] P. O'LEARY and F. MALOBERTI, "A Direct-Digital Synthesizer with Improved Spectral Performance," *IEEE Transactions on Communications*, vol. 39, no. 7, July 1991, pp. 1046–1048.

[11] V. KROUPA, V. CIZEK, J. STURSA, and H. SVANDOVA, "Spurious Signals in Direct Digital Frequency Synthesizers Due to the Phase Truncation," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 47, no. 5, September 2000, pp. 1166–1172.

[12] F. CURTICAPEAN and J. NIITTYLAHTI, "Exact Analysis of Spurious Signals in Direct Digital Frequency Synthesisers Due to Phase Truncation," *Electronics Letters*, vol. 39, no. 6, March 2003, pp. 499–501.

[13] B-G. GOLDBERG, "Digital Frequency Synthesizer," U.S. Patent US4,752,9029. Scitech Electronics, Inc., San Diego, CA, July 1985.

[14] H. NICHOLAS, H. SAMUELI, and B. KIM, "The Optimization of Direct Digital Frequency Synthesizer Performance in the Presence of Finite Word Length Effects," Proceedings of the 42nd Annual Frequency Control Symposium, June 1988, pp. 357–363.

[15] D. SUNDERLAND, R. STRAUCH, S. WHARFIELD, H. PETERSON, and C. COLE, "CMOS/SOS Frequency Synthesizer LSI Circuit for Spread Spectrum Communications," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, August 1984, pp. 497–506.

[16] K. ESSENWANGER and V. REINHARDT, "Sine Output DDSs: A Survey of the State of the Art," Proceedings of the 1998 IEEE International Frequency Control Symposium, May 1998, pp. 370–378.

[17] H. NICHOLAS and H. SAMUELI, "A 150-MHz Direct Digital Frequency Synthesizer in 1.25-μm CMOS with -90-dBc Spurious Performance," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 12, December 1991, pp. 1959–1969.

[18] G. KENT and N. SHENG, "A High Purity, High Speed Direct Digital Synthesizer," Proceedings of the 1995 IEEE International 49th Frequency Control Symposium, June 1995, pp. 207–211.

[19] DDS Tutorial: Technical Report V3. Scitech Electronics, Inc., San Diego, CA, 1991.

[20] B-G. GOLDBERG, "Source of Quantized Samples for Synthesizing Sinewaves," U.S. Patent US5,321,642. Scitech Electronics, Inc., San Diego, CA, June 1994.

[21] R. FREEMAN, "Digital Sine Conversion Circuit for Use in Direct Digital Synthesizers," U.S. Patent US4,809,205. Rockwell International Corp., El Segundo, CA, November 1989.

[22] A. BELLAOUAR, M. OBRECHT, A. FAHIM, and M. ELMASRY, "A Low-Power Direct Digital Frequency Synthesizer Architecture for Wireless Communications," Proceedings of the IEEE 1999 Custom Integrated Circuits, May 1999, pp. 593–596.

[23] J. LANGLOIS and D. AL-KHALILI, "A Low Power Direct Digital Frequency Synthesizer with 60 dBc Spectral Purity," Proceedings of the 12th ACM Great Lakes Symposium on VLSI, ACM Press, pp. 166–171, 2002.

[24] J. LANGLOIS and D. AL-KHALILI, "Piecewise Continuous Linear Interpolation of the Sine Function for Direct Digital Frequency Synthesis," *IEEE Radio Frequency Integrated Circuits (RFIC) Symposium*, June 2003, pp. 579–582.

[25] D. AL-KHALILI and J. LANGLOIS, "Phase to Sine Amplitude Conversion System and Method," European Patent EP1286258. Canada Min. Nat. Defence, February 2003.

[26] L. WEAVER Jr. and R. KERR, "High Resolution Phase to Sine Amplitude Conversion," U.S. Patent US4,905,177. Qualcomm, Inc., San Diego, CA, January 1988.

[27] J. VANKKA, "Methods of Mapping from Phase to Sine Amplitude in Direct Digital Synthesis," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 44, no. 2, March 1997, pp. 526–534.

[28] A. ELTAWIL and B. DANESHRAD, "Piece-wise Parabolic Interpolation for Direct Digital Frequency Synthesis," Proceedings of the IEEE Custom Integrated Circuits Conference, May 2002, pp. 401–404.

[29] C. WANG, H. SHE, and R. HU, "A ROM-less Direct Digital Frequency Synthesizer by Using Trigonometric Quadruple Angle Formula," *9th International Conference on Electronics, Circuits and Systems*, vol. 1, September 2002, pp. 65–68.

[30] G. GIELIS, R. VAN DE PLASSCHE, and J. VAN VALBURG, "A 540-MHz 10-b Polar-to-Cartesian Converter," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, November 1991, pp. 1645–1650.

[31] A. MADISETTI, A. KWENTUS, and A. WILLSON, "A 100-MHz, 16-b, Direct Digital Frequency Synthesizer with a 100-dBc Spurious-Free Dynamic Range," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 8, August 1999, pp. 1034–1043.

[32] C. KANG and E. SWARTZLANDER Jr., "An Analysis of the CORDIC Algorithm for Direct Digital Frequency Synthesis," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 2002, pp. 111–119.

[33] A. MCEWAN and S. COLLINS, "Analogue Interpolation Based Direct Digital Frequency Synthesis," *Proceedings of the 2003 International Symposium on Circuits and Systems ISCAS '03*, vol. 1, May 2003, pp. 621–624.

[34] J. GARVEY and D. BABITCH, "An Exact Spectral Analysis of a Number Controlled Oscillator Based Synthesizer," Proceedings of the 44th Annual Symposium on Frequency Control, May 1990, pp. 511–521.

[35] L. SCHUCHMAN, "Dither Signals and Their Effect on Quantization Noise," *IEEE Transactions on Communications*, vol. 12, no. 4, December 1964, pp. 162–165.

[36] M. FLANAGAN and G. ZIMMERMAN, "Spur-reduced Digital Sinusoid Synthesis," *IEEE Transactions on Communications*, vol. 43, no. 7, July 1995, pp. 2254–2262.

[37] J. VANKKA, "Spur Reduction Techniques in Sine Output Direct Digital Synthesis," Proceedings of the 1996 IEEE Frequency Control Symposium, June 1996, pp. 951–959.

[38] C. WHEATLEY III and D. PHILLIPS, "Spurious Suppression in Direct Digital Synthesizers," Proceedings of the 35th Annual Frequency Control Symposium, May 1981, pp. 428–435.

[39] H. SPANGIII and P. SCHULTHEISS, "Reduction of Quantizing Noise by Use of Feedback," *IEEE Transactions on Communications*, vol. 10, no. 4, December 1962, pp. 373–380.

[40] D. RIBNER and S. KIDAMBI, "Direct-Digital Synthesizers," European Patent EP1037379. Analog Devices Inc., Norwood, MA, September 2000.

[41] J. VANKKA, "A Direct Digital Synthesizer with a Tunable Error Feedback Structure," *IEEE Transactions on Communications*, vol. 45, no. 4, April 1997, pp. 416–420.

[42] T. HIGGINS, "Analog Output System Design for a Multifunction Synthesizer," *Hewlett-Packard Journal*, February 1989, pp. 66–69.

# Chapter 22

# Implementing a ΣΔ DAC in Fixed-Point Arithmetic

**Shlomo Engelberg**

*Jerusalem College of Technology*

**T**his chapter describes a simple sigma-delta (ΣΔ) digital-to-analog converter (DAC) that is suitable for use on the simplest of microprocessors to generate constant-level, or slowly varying, control voltages by taking advantage of a ΣΔ network's intrinsic limit cycle behavior. We describe the properties of the converter, show that it is inherently stable, and explain how the user can control the DC level of the DAC's analog output. Finally, we discuss the implementation of such a DAC on an industry-standard microprocessor.

## 22.1 THE ΣΔ DAC PROCESS

The block diagram of a simple ΣΔ DAC is shown in Figure 22-1. The input to the system is a number presented in digital form, and the "conversion" is done within the microprocessor. The single-bit binary logic-level output of the chip, $d(t)$, is applied to an analog lowpass filter. The output, $y(n)$, of the comparator in Figure 22-1 satisfies the equation $y(n) = f(z(n))$ where:

$$f(z) = \begin{cases} 3A/z & z \ge A \\ A/z & z < A, \end{cases}$$

and $A$ is a positive number chosen to be, in our application, half the peak amplitude of the $x(n)$ input.

As $z(n)$ is the output of the summation, it is clear that $z(n) = z(n-1) + e(n)$. From Figure 22-1 we see that $e(n) = x(n) - y(n-1) = x(n) - f(z(n-1))$, with $e(n)$ the error and $x(n)$ the input. Putting it all together, we find that:
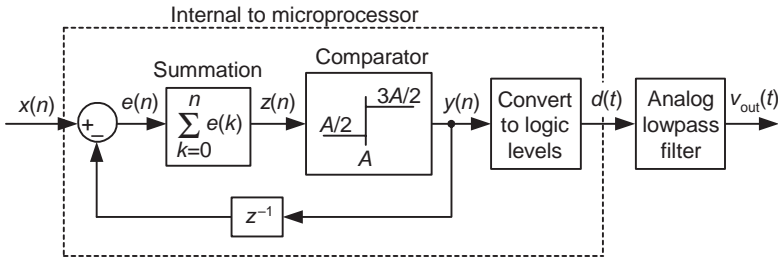
**Figure 22-1**    A simple ΣΔ DAC.

$$z(n) = z(n-1) + x(n) - f(z(n-1)). \qquad (22\text{–}1)$$

If $x(n) \equiv c$ is a constant that satisfies $A/2 \leq c \leq 3A/2$ and if $z(0)$ satisfies the condition $0 \leq z(0) < 2A$, then (using mathematical induction) it is not hard to show that, for all $n \geq 0$, the elements $z(n)$ satisfy:

$$0 \leq z(n) < 2A. \qquad (22\text{–}2)$$

Let us consider the practically important case in which $A$ is an even integer and $c$ is constrained to be an integer. This is often the case when implementing a ΣΔ DAC using a simple microprocessor. Under these conditions, $z(n)$ is constrained to be an integer between 0 and $2A-1$. Additionally, from (22–1) we see that $z(n)$ can be thought of as the state of a finite-state machine. As the next state of the machine is a function of the previous state, we find that if the machine ever repeats a state, then from that point on the state is periodic. As the machine has $2A$ possible states, it is clear that the maximal period that the ΣΔ DAC is capable of producing is $2A$ elements long. Using more sophisticated arguments, it is possible to show that the maximal period that the ΣΔ DAC is capable of producing is actually only $A$ elements long. We find that for a constant $x(n)$ input our feedback loop produces periodic oscillations—limit cycles. (For more information on limit cycles in ΣΔ-type circuits, see [1]–[3] and the references therein.)

It is easy to see that the average value of one cycle of the discrete time system's $y(n)$ output must be equal to the constant value at the input to the system. Suppose that this were not the case. Then at the end of each period the output of the summation block, the system's state, $z(n)$, would change. But this is inconsistent with the system's state being periodic. Thus, the average value must be the same as that of the constant input. The lowpass filter that produces the final output of the system will produce an output that tends to the value of the constant being input.

## 22.2    ΣΔ DAC PERFORMANCE

To understand the frequency-domain behavior of this simple ΣΔ DAC, we can perform a $z$-domain analysis by representing the summation block by $S(z)$ and representing the comparator as an additive noise source, $Q(z)$. These representations allow us to analyze the DAC as shown in Figure 22-2(a) and write
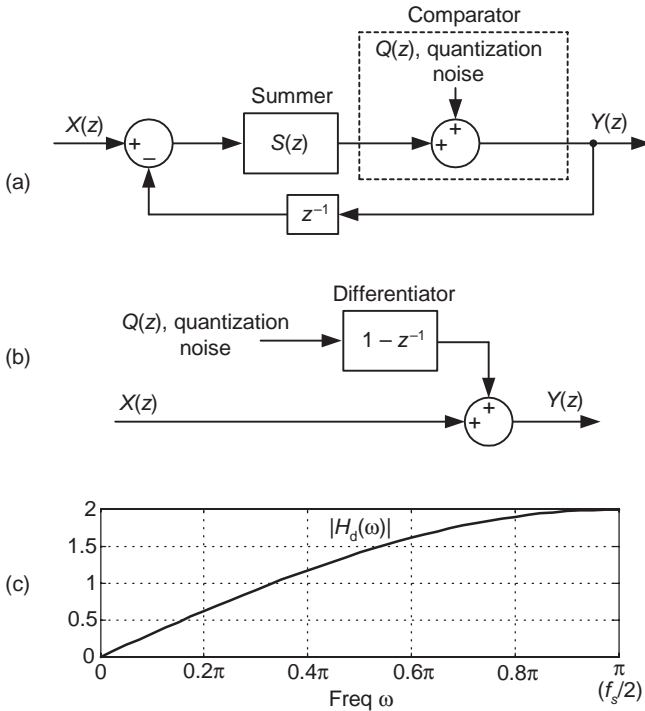
**Figure 22-2**   *Z*-domain analysis of a ΣΔ DAC: (a) block diagram; (b) differentiator; (c) magnitude of the differentiator's frequency response.

$$Y(z) = Q(z) + S(z)[X(z) - z^{-1}Y(z)]. \tag{22-3}$$

Solving (22–3) for $Y(z)$ yields

$$Y(z) = \frac{X(z)S(z)}{1 + S(z)z^{-1}} + \frac{Q(z)}{1 + S(z)z^{-1}}. \tag{22-4}$$

Because the summation block's $z$-domain expression is $S(z) = 1/(1 - z^{-1})$, (22–4) can be simplified to

$$Y(z) = X(z) + (1 - z^{-1})Q(z). \tag{22-5}$$

The $(1 - z^{-1})$ factor in (22–5) is a simple first-difference differentiator, depicted in Figure 22-2(b). The magnitude of the differentiator's frequency response is

$$|H_d(\omega)| = 2\left|\sin\left(\frac{\omega}{2}\right)\right|. \tag{22-6}$$

This function has a zero at $\omega = 0$. The magnitude of the differentiator's frequency response, $|H_d(\omega)|$, is plotted in Figure 22-2(c)—and it is the frequency response of a highpass filter.

Equations (22–5) and (22–6) tell us that the digital portion of the DAC passes the input, $x(n)$, without change while the quantization noise is converted to high-frequency noise by the feedback loop. The analog lowpass filter will have little effect on a low-frequency input signal while almost completely removing the highpass noise. (The effect that the feedback loop of a ΣΔ converter has on the quantization noise is called *noise shaping*.) We find that a ΣΔ DAC converts low-frequency digital signals into analog signals with minimal error.

## 22.3 IMPLEMENTATION

We now consider implementing a ΣΔ DAC using a microprocessor from the (extremely popular) 8051 family. The goal is to implement the DAC with a minimum of instructions. The 8051 family supports unsigned addition and subtraction of 8-bit numbers. Selecting $A = 128$, we find that we can implement a 7-bit DAC. The only tricky part is making sure that no intermediate calculations become negative or exceed 255.

When calculating $z(n)$ we consider the two cases $z(n-1) \geq 128$ and $z(n-1) < 128$ separately. In the first case, we calculate $z(n)$ by first computing $f(z(n-1)) - c = 192 - c$. This number must be between 0 and 128. Then we calculate $z(n-1) - (f(z(n-1)) - c)$. Because of the assumption about $z(n-1)$, this number will be greater than or equal to zero and less than or equal to 255. Thus there is no need to borrow or carry during this calculation.

If $z(n-1) < 128$, then we first calculate $c - f(z(n-1))$. This number must be greater than or equal to zero—there can be no need to borrow here. Next we calculate $z(n) = z(n-1) + (c - f(z(n-1)))$ and we have again avoided any need to carry or borrow. Additionally checking whether $z(n-1)$ is greater than or equal to 128 is easy to do on a microprocessor that is a member of the 8051 family. Some of the registers in an 8051 are bit addressable. By moving $z(n-1)$ into such a register and checking whether its most significant bit is set, one has checked the condition.

The next-to-final stage is to output $d(t)$. The easy way to do this is to use one of the general-purpose input/output (I/O) pins that all members of the 8051 family have. Rather than outputting the binary values 64 or 192, one outputs a logical low voltage or a logical high voltage, respectively. This introduces a scaling factor—but so do all DACs. The problem here is that the values taken by the general-purpose I/O pins are not generally 0 and $V_{cc}$ volts—they are voltages near these values. Though the values output are reasonably constant at any given pin, they need not be the same on all pins of a given port—to say nothing of the pins on different ports. (The choice of an output pin is usually not considered very important. Here changing pins may change the value actually seen at the output of the lowpass filter.) A ΣΔ DAC implemented this way should have 7-bit resolution but will not have 7-bit accuracy. Practically speaking one may find that one does not need the absolute accuracy. Alternatively, one could add hardware before the analog lowpass filter that would accommodate level shifting.

The final stage in the $\Sigma\Delta$ DAC is an analog lowpass filter. To avoid the need for a complicated, high-quality, lowpass filter one must make sure that the sampling speed of the digital portion is high enough. One sets the sampling speed—the length of the delay in the loop—by setting the rate at which an (internal) timer interrupts the microcontroller. If the sampling speed is high enough and if the input $x(n)$ signal is sufficiently low frequency, then one does not need an expensive lowpass filter. Suppose, for example, that one would like to convert the constant digital signal $x(n) = c$ to an analog voltage. Suppose that an 8-bit microprocessor is being used and a 7-bit DAC is desired. Then the maximal period of the output, $y(n)$, of the system is 128 clock cycles. Suppose that your clock is interrupting every 10 μs—that you perform a new calculation every 10 μs. (This is practical with many of the modern 8052-based processors.) Then the frequency of the noise riding on the constant starts at about 800 Hz. If one uses a simple RC lowpass analog filter with a 25 Hz cutoff frequency, one will obtain good results.

Also note that one of the advantages of the sigma-delta DAC relative to a pulse width modulation (PWM) DAC is that for many constant-valued inputs the period of the sequences $z(n)$ and $y(n)$ will be much less than 128 samples long and the frequency of the noise that "rides" on the desired output will be higher than the frequency of the noise produced by the PWM DAC. In such cases, an inexpensive lowpass filter will work well.

In order to demonstrate how a $\Sigma\Delta$ DAC behaves, we consider several examples. In Figure 22-3 the signals $z(n)$, $y(n)$, and $d(t)$ are shown when $A = 128$ and $x(n) = 188$. There we find that $z(n)$ remains between 0 and 255 and the average value of $y(n)$ is equal to 188, as the theory predicts.

When $x(n) = A = 128$, the output of the digital portion, $y(n)$, oscillates between the values $A/2$ (64) and $3A/2$ (192), and has an average value of 128—as it should. In this situation a low-cost analog lowpass filter will work very well indeed.
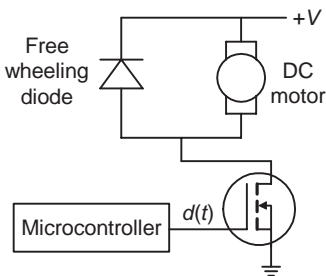
Figure 22-4 shows $z(n)$, $y(n)$, and $d(t)$ when $A = 128$ and $x(n) = 68$. Again $z(n)$ remains between 0 and 255 and the average value of $y(n)$ is equal to the value of
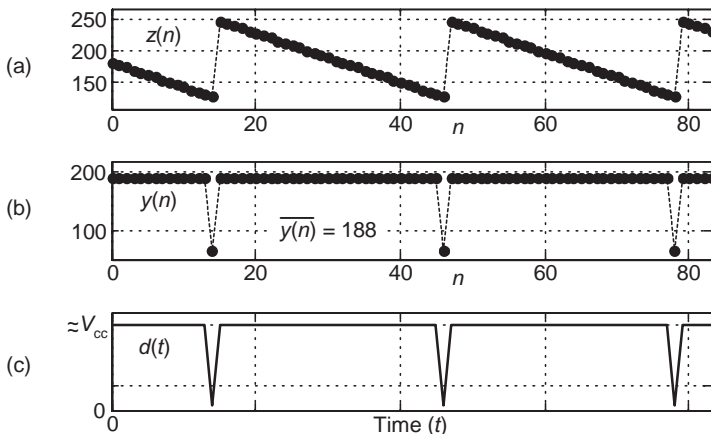


**Figure 22-3**   DAC sequences for $A = 128$ and $x(n) = 188$.
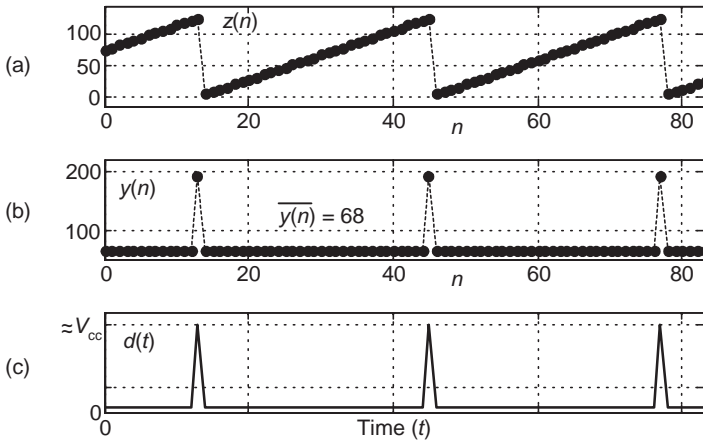
**Figure 22-4** DAC sequences when $A = 128$ and $x(n) = 68$.



**Figure 22-5** Unidirectional speed control of a DC motor.

the $x(n)$ input. Additionally, in each figure the period of the output is significantly less than the maximum period that the theory permits—and this means that our lowpass filtering will be more effective than we predicted.

If one is interested in controlling the speed of a DC motor, one can take care of the problem of the not-very-well-defined logic levels and the analog lowpass filtering in a particularly simple fashion. DC motors are themselves lowpass filters—so no lowpass filtering is required. Additionally, the DC motor requires an input that is capable of driving the motor—and not the signal-level outputs of a microcontroller. One way to produce such a final output signal is to use the logic-level outputs of the microcontroller to control some form of power amplifier—and if done properly, the power amplifier can take care of the level shifting, too.

If one is interested in using the motor to turn in one direction only, then connecting a MOSFET with a freewheeling diode to the output of the microcontroller, as shown in Figure 22-5, will do the job. As the MOSFET's resistance when turned on is very close to zero, the voltage driving the DC motor will be almost precisely $+V$. If one is interested in bidirectional control, then an H-bridge DC motor motion-

control chip (such as the National Semiconductor LM18201, or the Texas Instruments L293) is called for.

## 22.4  CONCLUSIONS

We have presented the properties of a simple $\Sigma\Delta$ DAC. We have shown that, when one uses fixed-point arithmetic, a constant input leads to a limit cycle whose average value is precisely the value of the constant. We have also shown that it is possible to actually implement the digital portion of DACs of this type in a simple fashion using microprocessors from the simple and popular 8051 family. Finally, we considered some implementation issues as well.

## 22.5  REFERENCES

[1]  V. FRIEDMAN, "The Structure of the Limit Cycles in Sigma Delta Modulation," *IEEE Transactions on Communications*, vol. 36, no. 8, August 1988, pp. 972–979.

[2]  R. GRAY, "Oversampled Sigma-Delta Modulation, *IEEE Transactions on Communications*, vol. 35, no. 5, May 1987, pp. 481–489.

[3]  D. REEFMAN, J. REISS, E. JANSSEN, and M. SANDLER, "Description of Limit Cycles in Sigma-Delta Modulators," *IEEE Transactions on Circuits and Systems I—Regular Papers*, vol. 52, no. 6, June 2005, pp. 1211–1223.

# Chapter 23

# Efficient 8-PSK/16-PSK Generation Using Distributed Arithmetic

**Josep Sala**

*Technical University of Catalonia*

**T**his chapter describes a computationally efficient technique using distributed arithmetic (DA) to implement the digital pulse shaping filters required in generating phase shift keyed (PSK) signals for digital communications systems. DA constitutes an efficient multiplierless procedure for digital filtering in terms of reduced quantization noise. Nevertheless, optimization is usually required to reduce the exponential size of the associated lookup tables (LUTs). Optimization techniques are either general (i.e., partitioning, where the filter is implemented as the addition of smaller subfilters), or specific (this chapter's material), when so allowed by the structure of the input signal or the filter itself. LUT size is critical in 8-PSK and 16-PSK finite impulse response (FIR) pulse shaping. Hence, here we show that 8-PSK and 16-PSK can be split into the superposition of simpler constellations, which leads to reduced complexity in DA implementations.

## 23.1  BACKGROUND

Pulse shaping (symbol filtering) constitutes the most complex part of digital modulators. Although some approaches use trivial coefficients for complexity reduction [1], DA is more flexible for implementing filters of any shape. Therefore, before describing the specifics of 8-PSK and 16-PSK optimization, and for the sake of

completeness, a brief summary of DA and the structure of baseband digital modulators is provided. A detailed description of DA and related optimizations may be found in [2], [3]. In radix-2 DA, the output $y[n]$ from a FIR filter of impulse response $h_n$, when the input sequence $x[n]$ is quantized to $B$ bits, may be evaluated from the following decomposition,

$$y[n] = \sum_{k=0}^{L-1} h_k \cdot x[n-k] = \sum_{k=0}^{L-1} h_k \sum_{m=0}^{B-1} x_{n-k,m} w_m 2^{-m}$$

$$= \sum_{m=0}^{B-1} 2^{-m} w_m \cdot \overbrace{\left( \sum_{k=0}^{L-1} h_k x_{n-k,m} \right)}^{T_M} \tag{23-1}$$

where $x_{n-k,m}$ constitutes the bit encoding of the input sample $x[n-k]$ and $w_m$ the two's complement sign encoding of each binary weight ($w_0 = -1$ and $w_m = +1$ for all $m \geq 1$). All possible combinations $T_m = h_0 x_{n,m} + h_1 x_{n-1,m} + \ldots + h_{L-1} x_{n-L+1,m}$ are quantized and stored in a LUT of size $2^L$, which is addressed by the $L$-bit-long word $\mathbf{x}_{n,m} = (x_{n,m}, \ldots, x_{n-L+1,m})$ (optimizations [3] based on offset binary coding of the input data or filter partitioning may be used to reduce LUT size). Thus, quantizing $T_m$ rather than $h_n$ allows us to improve the signal-to-quantization-noise power ratio. After $B$ lookups and $B-1$ shifts and accumulations (summation over $m$), the output $y[n]$ is generated.

A baseband linear modulator performs pulse shaping with $h_n$ as

$$b[n] = \sum_{k=-\infty}^{+\infty} s[k] \cdot h_{n-k \cdot N_{SS}} \tag{23-2}$$

with $N_{SS}$ the number of samples per symbol at the modulator output $b[n]$, and $s[k]$ the input symbol sequence. This can be recast into a bank of $N_{SS}$ subfilters, each corresponding to the convolution with each of the different $N_{SS}$ decimated versions of the shaping pulse $h_{n_1}[n_2] = h_{n_1 + Nss \cdot n_2}$, with $0 \leq n_1 \leq N_{SS}-1$ and $-\infty \leq n_2 \leq +\infty$. Hence,

$$b[n_1 + n_2 N_{SS}] = \sum_{k=-\infty}^{+\infty} s[k] \cdot h_{n_1}[n_1 - k] \tag{23-3}$$

This allows us to reduce the memory space, as only $N_{SS}$ LUTs associated with the different $h_{n_1}[n_2]$ are needed to compute the corresponding $b_{n_1}[n_2] = b[n_1 + n_2 N_{SS}]$. Each decimated filter $h_{n_1}[n_2]$ has either $L = \lceil N_C/N_{SS} \rceil$ or $L = \lfloor N_C/N_{SS} \rfloor$ coefficients, with a global LUT size $2^L N_{SS}$.

## 23.2  GENERATION OF 8-PSK AND 16-PSK MODULATIONS

For BPSK, QPSK, ASK, or QAM modulations, DA is straightforward as the input symbols $s[k]$ can be exactly quantized using very few bits (and hence, very few DA shifts and accumulations are necessary). Nevertheless, 8-PSK and 16-PSK do not

easily yield to implementation in DA as their symbols do not exactly coincide with a finite (regular) quantization grid. An excessive number of bits is required to yield quantized symbols $s[k]$ with negligible amplitude and phase residual errors. Taking 8-PSK, an alternative to symbol quantization is to generate a LUT of size $4^L$ to store all possible outputs ($8^L$ for 16-PSK), as the 8-PSK symbols have four different projections on the $I$-axis (and the $Q$-axis) as shown in Figure 23-1 using the small arrows p1 to p4. Unfortunately, for low $L$, the memory size is already too large, as medium to low rolloff factors of the shaping pulse may require a substantial number of coefficients.

Although filter partitioning may be applied, we propose a previous optimization, called *constellation superposition*, which helps reduce the memory space to a large extent. We prove that the 8-PSK constellation can be decomposed into the exact addition of two QPSK constellations of different scale. The resulting constellation has 16 symbols, out of which eight coincide exactly with an 8-PSK constellation. Accordingly, the 16-PSK constellation can be split into the exact addition of two 8-PSK constellations, or alternatively, into the addition of four QPSK constellations of different scale. As symbol shaping is a linear operation, decomposing 8-PSK or 16-PSK into the addition of simpler constellations is equivalent to decomposing the modulator output into the addition of simpler modulators. These decompositions are depicted in Figures 23-1 and 23-2. A geometrical proof is used to derive these results.
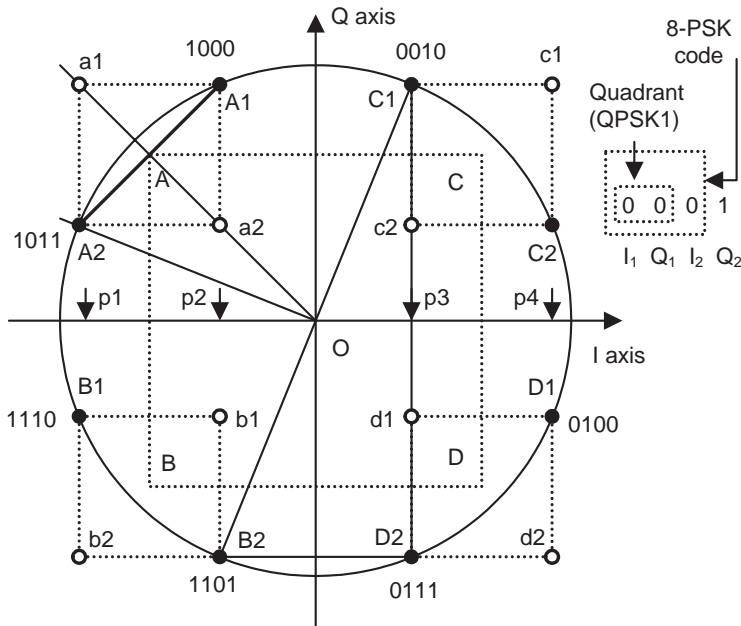


**Figure 23-1**  Construction of the 8-PSK constellation in terms of two constituent QPSK constellations.
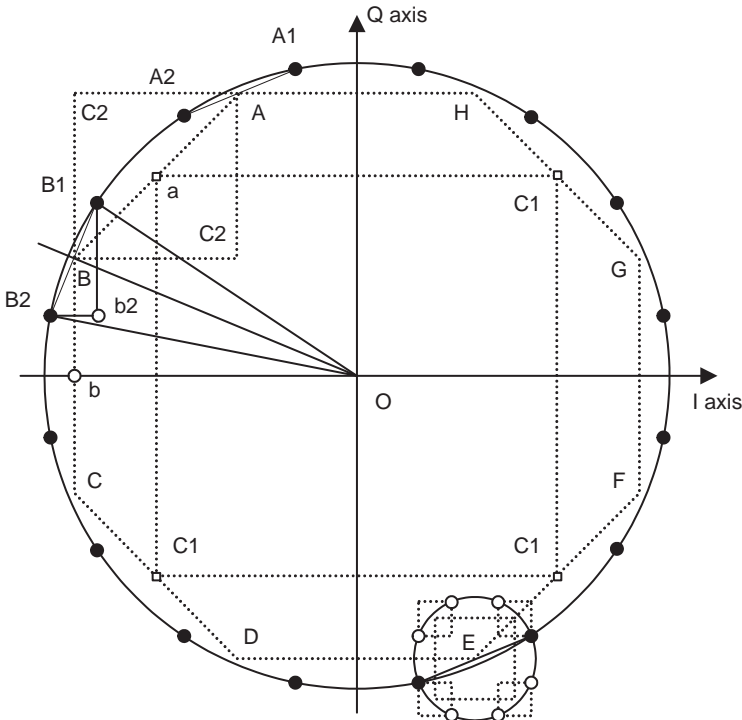
**Figure 23-2** Construction of the 16-PSK constellation in terms of four constituent QPSK constellations or two constituent 8-PSK constellations.

## 23.3 OPTIMIZED GENERATION OF 8-PSK MODULATIONS

Figure 23-1 shows the decomposition of 8-PSK into the addition of two QPSK constellations: the vertices A, B, C, D constitute the larger QPSK constellation, where the upper-left vertex A is defined as the midpoint of segment A1–A2 (and so on for the other vertices B, C, and D). A second, smaller QPSK constellation, centered at A, is used to generate A1 and A2 (which coincide with the desired 8-PSK constellation), and two other unused symbols a1 and a2. The same procedure applies to the other vertices B, C, and D. Therefore, we may write the 8-PSK constellation as $8\text{PSK} = \text{QPSK}_1 + \xi \cdot \text{QPSK}_2$, with $0 < \xi < 1$ being the scale factor between the larger $\text{QPSK}_1$ and the smaller $\text{QPSK}_2$ constellations. If we let $\rho$ denote the radius of the 8-PSK constellation, then we may calculate $\xi$ considering triangle A2–O–A. Note that angle A2–O–A (between segments A2–O and O–A) is half the angle between two neighboring 8-PSK symbols, which is $\pi/8$. Hence, segments A2–A and O–A are the sides of a right-angled triangle, of hypotenuse $\rho$, and also the radius of constellations $\text{QPSK}_2$ and $\text{QPSK}_1$, respectively. Hence, the ratio of segment lengths |A2−A|/|O−A| equals $\xi$, which yields $\xi = \tan(\pi/8)$.

In Figure 23-1 the 4-bit labels used to identify the two QPSK pseudosymbols are related to the 3-bit code of the 8-PSK constellation according to the procedure described in Section 23.5 on pseudosymbol generation for 8-PSK/16-PSK constellations.

Thus, two LUTs of equal size are required, storing the $QPSK_1$ and $QPSK_2$ outputs, respectively. The scale factor $\xi$ is absorbed into the contents of the LUT corresponding to modulator $QPSK_2$. Very simple logic generates the pseudo-QPSK symbols to the two constituent QPSK modulators. Hence, the complexity of 8-PSK modulation is twice that of QPSK, which equals the ratio of their corresponding number of symbols: $8/4 = 2$.

An alternative approach [4] for the generation of 8-PSK (and 16-PSK) considers the rotation and combination of QPSK constellations as $8PSK = C_1 + \exp[j\pi/4]C_2$. Nevertheless, this requires that the constituent constellations $C_1$ and $C_2$ be QPSK, extended with the zero symbol (five symbols per constellation). This is necessary as $C_1$ needs to be zero (modulator 1) when the 8-PSK symbol is generated by $\exp[j\pi/4]C_2$ (modulator 2). If DA is used to perform pulse shaping, the 0-symbol extension increases LUT size.

A useful result we will apply to the 16-PSK case is angle B2–C1–D2. Note that this angle is the same as that between segment O–C1 and the *Q*-axis, which yields $\pi/8$.

## 23.4  OPTIMIZED GENERATION OF 16-PSK MODULATIONS

Figure 23-2 shows the decomposition of 16-PSK into the addition of two 8-QPSK constellations. Grouping the 16 symbols into pairs {A1,A2} up to {H1,H2} and using A up to H to denote the midpoint of segments A1–A2 up to H1–H2, respectively, the set of points {A,B,C,D,E,F,G,H} defines a regular octagon whose vertices constitute an 8-PSK constellation. A smaller 8-PSK constellation may be centered at each vertex of the octagon so that two of their eight symbols coincide with the symbols of the 16-PSK constellation (see example at vertex E). To this purpose, and considering center B, it is sufficient to show that angle B2–B1–b2 is precisely $\pi/8$, which, as proved previously, is the case for 8-PSK (see angle B2–C1–D2 in Figure 23-1). The reasoning follows in three steps:

1. Segments B2–B1 and B–O, crossing at B, are perpendicular.
2. Angle b–O–B (equal to angle B2–O–B1) is $2\pi/16 = \pi/8$; therefore its complementary angle b–B–O is $\pi/2 - \pi/8$.
3. Angle B2–B–b, which is the same angle as B2–B1–b2, is complementary to b–B–O (because of the perpendicularity of segments B2–B1 and B–O), which equals $\pi/2 - (\pi/2 - \pi/8) = \pi/2$.

Hence, an 8-PSK constellation centered at B generates symbols B1 and B2 on the 16-PSK constellation. This is easily extended to the other seven octagon vertices due to the symmetries of the 16-PSK constellation (see example at vertex

E). Therefore, we may write the 16-PSK constellation as $16PSK = [8PSK]_1 + \mu[8PSK]_2$.

To compute $\mu$, note that segment B2–B is the radius $\rho_2$ of the smaller 8-PSK constellation, while the opposite side O–B of the same right-angled triangle B2–B–O is the radius $\rho$ of the larger 8-PSK constellation. As angle B2–O–B is precisely $\pi/16$ (half the angle between neighboring 16-PSK symbols), its tangent is $\mu = \rho_2/\rho = \tan(\pi/16)$. Therefore, a complete expansion in terms of QPSK constellations yields

$$
\begin{aligned}
16PSK = & \left[ QPSK_1 + \tan\left(\frac{\pi}{8}\right) \cdot QPSK_2 \right] \\
& + \tan\left(\frac{\pi}{16}\right) \cdot \left[ QPSK_3 + \tan\left(\frac{\pi}{8}\right) \cdot QPSK_4 \right]
\end{aligned}
\tag{23–4}
$$

In summary, two 8-PSK modulators of equal complexity are required, generating the $[8PSK]_1$ and $[8PSK]_2$ outputs, respectively, plus additional combinatorial logic to perform scaling by $\tan(\pi/16)$. Alternatively, four QPSK modulators may be used, with all scaling factors included in their respective LUTs. Very simple logic generates the pseudo-QPSK or pseudo-8-PSK symbols to the constituent modulators. Hence, the complexity of 16-PSK modulation is roughly fourfold that of QPSK, which equals the ratio of their corresponding number of symbols: $16/4 = 4$.

The embedding of the 16-PSK constellation into the irregular 256-QAM constellation generated by the four constituent QPSK modulators is shown in Figure 23-3.
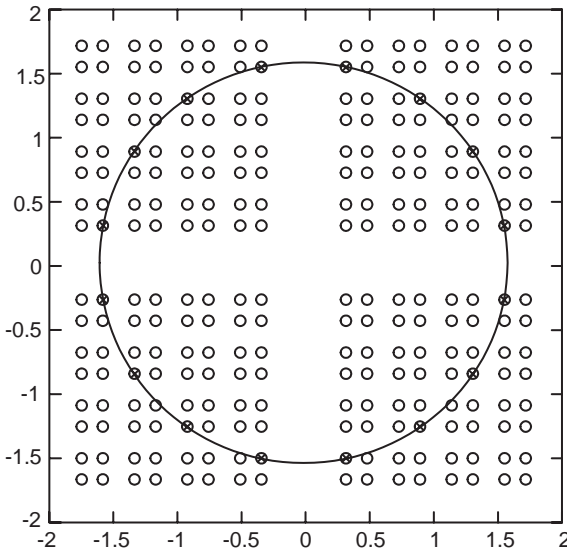


**Figure 23-3**   Embedding of the 16-PSK constellation in the irregular 256-QAM constellation.
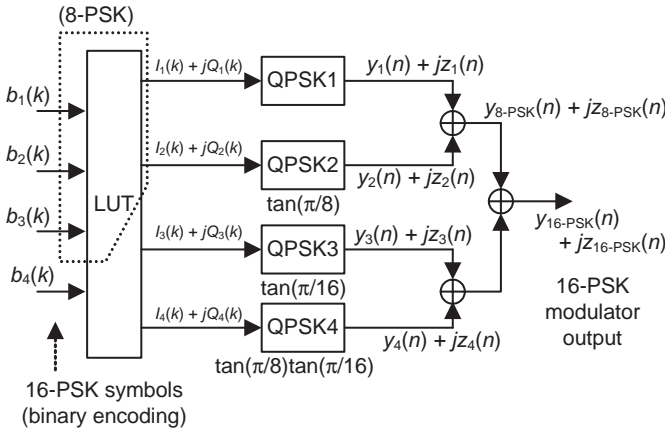
**Figure 23-4**    One version of 16-PSK modulator implementation.
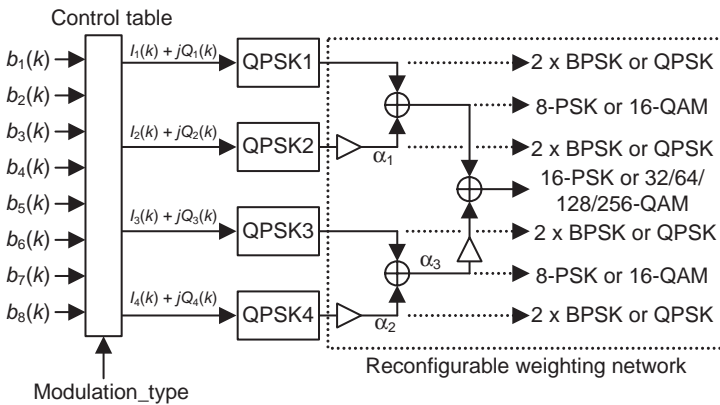


**Figure 23-5**    Multipurpose modulator architecture.

Figure 23-4 shows one possible implementation of a 16-PSK modulator where all four constituent QPSK modulators are implemented in parallel for maximum throughput. A small LUT (16 positions) or combinatorial function is required to translate each 16-PSK symbol to its expression into a set of four QPSK pseudosymbols. The scaling factors are incorporated in each QPSK LUT.

Figure 23-5 shows how a slight modification in the 16-PSK implementation in Figure 23-4 yields a flexible architecture allowing us to generate a number of different modulations. The setting of the control table and of the multiplication factors $\alpha_i$ in the weighting network may generate a wide variety of modulations. For example, setting $\alpha_1 = \alpha_2 = \alpha_3 = 1/2$ allows us to generate the 256-QAM modulation. Note that all QAM modulations up to 256-QAM can be generated with the addition of up to four QPSK constellations, where the multiplication factors are conveniently chosen to be either 0 or 1/2 (and the control table is conveniently configured to
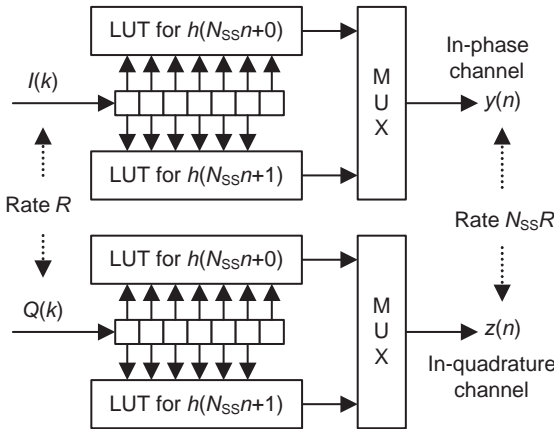
**Figure 23-6**    Structure of one of the constituent QPSK modulators in Figures 23-4 and 23-5.

generate the corresponding pseudosymbols). All QPSK modulators are identical, which makes hardware reuse possible.

Figure 23-6 shows the internal structure of one of the constituent QPSK modulators in Figures 23-4 and 23-5 for the case $N_{SS}=2$ samples per symbol and a 13-coefficient shaping pulse $h(n)$. A bit-level shift register is used to address the LUTs storing the outputs corresponding to the two different decimation phases of the shaping filter. The outputs are multiplexed to yield the modulator output corresponding to each decimation phase.

## 23.5  PSEUDOSYMBOL GENERATION

We describe a symbol labeling procedure suitable for generating the pseudo-QPSK symbols to the constituent QPSK modulators: This requires 4- and 8-bit *labels* for 8-PSK and 16-PSK symbols, respectively, which are not to be mistaken with the original 3- and 4-bit *codes* of the corresponding constellations. Let the two bits identifying a QPSK symbol be denoted $b_1, b_2$. Let the corresponding QPSK symbol be denoted $I+jQ=S(b_1)+jS(b_2)$, where the sign function S of a digital bit $b$ is defined as $\{S(0)=1, S(1)=-1\}$. In this way, the opposite symbol in the constellation is the negation of the corresponding bits. Let the 3 and 4 bits corresponding to one 8-PSK or 16-PSK symbol be denoted $b_1, b_2, b_3$ and $b_1, b_2, b_3, b_4$, respectively. Then, setting $I_2=S(b_3)$, the 8-PSK symbol can be constructed as

$$I^{8-\text{PSK}} + jQ^{8-\text{PSK}} = (I_1 + jQ_1) + \tan\left(\frac{\pi}{8}\right) \cdot \left( I_2 + j\overbrace{(-I_1Q_1I_2)}^{Q_2} \right) \tag{23-5}$$

with $Q_2=-I_1Q_1I_2$. This can be verified in Figure 23-1. The first pseudosymbol (QPSK$_1$): $I_1+jQ_1$, selects the quadrant (symbols A, B, C, or D). The second
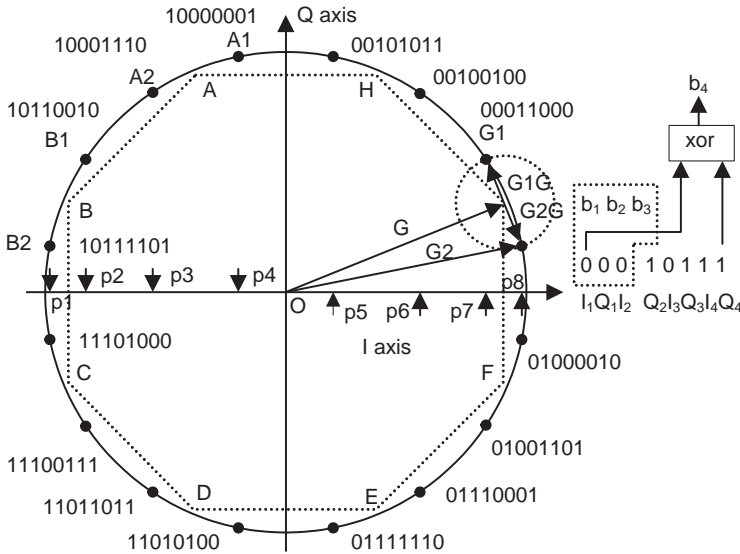
**Figure 23-7**  Construction of 8-bit labels for identifying the four constituent QPSK pseudosymbols in the 16-PSK constellation.

pseudosymbol (QPSK$_2$): $I_2+jQ_2$, selects which of the two allowed pseudosymbols for that quadrant has to be generated. Note that any $I_2+jQ_2$ of the smaller QPSK$_2$ constellation is perpendicular to $I_1+jQ_1$ in all cases (their relative angle is $\pm\pi/2$ radians; see, for example, symbol A of QPSK$_1$ and symbols A1 and A2 in Figure 23-1). Hence, the term $I_2$ can be used to control the sign of $I_2+jQ_2$, which allows us to determine $Q_2=-I_1Q_1I_2$ in Figure 23-1. Therefore the three independent terms $I_1$, $Q_1$, $I_2$ can be associated with bits $b_1$, $b_2$, $b_3$, respectively, and we may set $I_1=S(b_1)$, $Q_1=S(b_2)$, $I_2=S(b_3)$.

   With these thoughts in mind, Figure 23-7 shows the construction of the 8-bit labels for the identification of the four constituent QPSK pseudosymbols in the 16-PSK constellation. There the relationships between the original 4-bit codes of a 16-PSK symbol are shown. The small arrows p1 to p8 show the eight possible values of the real (and imaginary) component of the 16-PSK symbols.

   We note that to generate any 16-PSK symbol as the addition of two 8-PSK symbols, the two constituent 8-PSK symbols are perpendicular, for example, in Figure 23-7 where symbol G1 is generated as the addition of G and G1G. Therefore, we observe in the graph of Figure 23-1, that for any 8-PSK symbol, either the per-pendicular symbol at +90° or the one at −90° has a binary *label* that is a left-to-right flipped version of the first 8-PSK symbol's binary *label*. For example, symbols C2 and A1 in Figure 23-1 have left-to-right flipped labels. A1 is found at +90° from C2, and its opposite D2 at −90° from C2 then has a label that is the bit-inversion of A1's label. It can be easily checked that this property holds for all 8-PSK symbols. Because of perpendicularity between the two constituent 8-PSK symbols of any 16-PSK symbol, the fourth 16-PSK bit $b_4$ can be used to control the sign of the

second constituent (and perpendicular) 8-PSK symbol with $S_4 = S(b_4)$, such that final 16-PSK symbol labeling can be expressed as

$$I^{16-\text{PSK}} + jQ^{16-\text{PSK}} = (I_1^{8-\text{PSK}} + jQ_1^{8-\text{PSK}}) +$$

$$\tan\left(\frac{\pi}{16}\right) \cdot (I_2^{8-\text{PSK}} + jQ_2^{8-\text{PSK}})$$

$$I_1^{8-\text{PSK}} + jQ_1^{8-\text{PSK}} = (I_1 + jQ_1) + \tan\left(\frac{\pi}{8}\right) \cdot (I_2 - jI_1Q_1I_2)$$

$$\text{(23–6)}$$

$$I_2^{8-\text{PSK}} + jQ_2^{8-\text{PSK}} = S_4 \cdot \left[ (-I_1Q_1I_2 + jI_2) + \tan\left(\frac{\pi}{8}\right) \cdot (Q_1 + jI_1) \right]$$

$$= (I_3 + jQ_3) + \tan\left(\frac{\pi}{8}\right) \cdot (I_4 + jQ_4)$$

It can be checked by visual inspection of (23–6) that, except for the sign control $S_4$, the four real/imaginary terms in $I_{1,8\text{-PSK}} + jQ_{1,8\text{-PSK}}$ are just those of $I_{2,8\text{-2PSK}} + jQ_{2,8\text{-PSK}}$ flipped left-to-right. This procedure can be examined in Figure 23-7, where the addition of the first constituent 8-PSK symbol (vector G) with the second constituent (perpendicular) 8-PSK symbol (vector G2G), generates the 16-PSK symbol G2. Note that for symbol G1 the first group of 4 bits of its *label*, 00011000, is a left-to-right flipped version of the second group of 4 bits, which identify, respectively, the first and second constituent 8-PSK symbols.

The label for symbol G2 is 00010111, whose second half is the inversion of G1's second half, as controlled by $b_4$. Note that $b_4$ can be obtained from the product $I_1Q_4 = (I_1)^2S_4 = S_4 = S(b_4)$, which corresponds to taking the exclusive-OR of the first and last bit of the 8-bit label. Along with $I_1 = S(b_1)$, $Q_1 = S(b_2)$, $I_2 = S(b_3)$, the 4 bits identifying a 16-PSK symbol can be recovered.

It is easy to verify by generating any complex symbol according to (23–5) or (23–6) that the radius of the 8-PSK and 16-PSK constellations becomes, respectively, $r_{8\text{-PSK}} = \sqrt{2} \cdot [\cos(\pi/8)]^{-1}$ and $r_{16\text{-PSK}} = \sqrt{2} \cdot [\cos(\pi/8) \cos(\pi/16)]^{-1}$.

## 23.6    COMPLEXITY EVALUATION

In this section, we will compare the complexity of two alternative DA implementations with the proposed QPSK constellation superposition method. To generate the $N_{SS}$ complex samples in one output symbol, we require the $N_{SS}$ subfilters (decimated versions of the original pulse shape) denoted $h_{n1}[n_2] = h[n_1 + n_2 N_{SS}]$ in (23–3). The final complexity measure results from the addition of the complexities of the $N_{SS}$ subfilters, which depends on their respective lengths. For a pulse of $N_C$ coefficients, we set $N_C = N_1 N_{SS} + N_2$, with $N_1 = \lfloor N_C/N_{SS} \rfloor$, so that we get $N_2 = N_C - N_{SS}N_1$ subfilters of $N_1 + 1$ coefficients plus $N_{SS} - N_2$ subfilters of $N_1$ coefficients. Let $P_M(L)$ denote the number of memory positions in a DA method M for implementing a real subfilter of length $L$ operating on a real channel. Then, the total number of required memory positions $P_{all,M}$ required for processing either the in-phase or the in-quadrature channel becomes

$$P_{\text{all},M} = N_2 \cdot P_M(N_1+1) + (N_{SS} - N_2) \cdot P_M(N_1) \tag{23-7}$$

**EXAMPLE**

For a 17-coefficient filter $\boldsymbol{h} = [h_0 h_1 h_2 \ldots h_{15} h_{16}]$ and $N_{SS} = 4$ samples per symbol, the four different decimations of the filter are: $\boldsymbol{h}_0 = [h_0 h_4 h_8 h_{12} h_{16}]$ of length $N_1 + 1 = \lfloor 17/4 \rfloor + 1 = 5$, plus $N_{SS} - N_2 = 4 - 1 = 3$ other subfilters $\boldsymbol{h}_1 = [h_1 h_5 h_9 h_{13}]$, $\boldsymbol{h}_2 = [h_2 h_6 h_{10} h_{14}]$, and $\boldsymbol{h}_3 = [h_3 h_7 h_{11} h_{15}]$ of length $N_1 = 4$. Hence, $P_{\text{all},M} = P_M(5) + 3P_M(4)$.

Now, we will compare $P_M(L)$ for three different methods M1, M2, and M3, also using additional optimization procedures (memory partitioning).

1. **Architecture M1.** A single LUT is used for the in-phase and in-quadrature channel to store all possible outputs of the modulator. If we look at either channel, the 8-PSK and 16-PSK constellations can take four (from p1 to p4 in Figure 23-1) and eight (from p1 to p8 in Figure 23-7) possible values, so that, respectively, the corresponding LUT size is $4^L$ and $8^L$ for storing all outputs of an $L$-coefficient filter. This is clearly unattainable for even moderate filter lengths. Therefore, filter partitioning techniques have to be applied where the filter is split into the addition of two or more smaller subfilters. For example, filter $\boldsymbol{h}_0 = [h_0 h_4 h_8 h_{12} h_{16}]$ ($L = 5$) can be split into the addition of subfilters $\boldsymbol{h}_{01} = [h_0 h_4 h_8 00]$ ($L_1 = 3$) and $\boldsymbol{h}_{02} = [000 h_{12} h_{16}]$ ($L_2 = 2$). Then, instead of one large LUT, we add the outputs of two smaller LUTs. For an $(L_1, L_2)$ partitioning of an $L$-coefficient filter with $L = L_1 + L_2$ (partitioning by two), the final number of memory positions after partitioning is reduced in the 8-PSK case from $4^L$ to $4^{L_1} + 4^{L_2}$ (in 16-PSK from $8^L$ to $8^{L_1} + 8^{L_2}$), in our case, from $4^5 = 1024$ to $4^3 + 4^2 = 80$ (in 16-PSK from 32768 to 576). The most suitable $(L_1, \ldots, L_r)$ partitioning with $L = L_1 + \ldots + L_r$ will depend in each case on constraints of the implementation platform. In fact, if for an $L$-coefficient filter we apply partitioning by $r = L$, we get $L$ different LUTs of size 4 (8-PSK) or 8 (16-PSK) corresponding to the different multiplications of the in-phase/in-quadrature component of the complex symbol with each of the respective $L$ coefficients. The number of memory positions required by method M1 for a filter of length $L$ and partitioning by $r$ is expressed as

$$
\begin{aligned}
P_{M1}(L, r) &= L_2 \cdot C^{L_1 + 1} + (r - L_2) \cdot C^{L_1} \\
L &= L_1 \cdot r + L_2 \\
L_1 &= \lfloor L/r \rfloor \\
L_2 &= L - r \cdot L_1
\end{aligned}
\tag{23-8}
$$

where $C = 4$ (8-PSK) or $C = 8$ (16-PSK). This partitioning corresponds to generating $L_2$ subfilters of length $L_1 + 1$ ($C^{L_1+1}$ memory positions) plus $r - L_2$ subfilters of length $L_1$ ($C^{L_1}$ memory positions). An aspect associated with memory partitioning is the number of additions required for adding up all subfilter LUT outputs, which is precisely $N_+(r) = r - 1$. Although partitioning reduces memory size, it introduces latency associated with the number of additions, and if one is not careful, an increased level of quantization noise. The minimum memory size is achieved for the maximum number of additions ($L$ memories of size $C$), in partitioning by $r = L$. In DA implementations, partitioning may be used to trade off additions with number of memory positions (which depends on additional constraints of the implementation platform).

2. **Architecture M2 (QPSK constellation superposition).** In this architecture we use the constellation partitioning techniques previously exposed. In the 8-PSK case (two constituent QPSK modulators), and for the five-coefficient subfilter $h_0 = [h_0 h_4 h_8 h_{12} h_{16}]$, the number of required memory positions is $2 \times 2^5 = 64$ ($4 \times 2^5 = 128$ for 16-PSK, four constituent QPSK modulators). If the same partitioning as in the previous example had been applied, the final number of memory positions would be $2 \times (2^3 + 2^2) = 24$ (8-PSK) and $4 \times (2^3 + 2^2) = 48$ (16-PSK). The number of memory positions required by method M2 for a filter of length $L$ and partitioning by $r$ is expressed as

$$
\begin{aligned}
P_{M2}(L, r) &= (C/2) \cdot \left( L_2 \cdot 2^{L_1+1} + (r - L_2) \cdot 2^{L_1} \right) \\
L &= L_1 \cdot r + L_2 \\
L_1 &= \lfloor L/r \rfloor \\
L_2 &= L - r \cdot L_1
\end{aligned}
\tag{23–9}
$$

Factor $C/2 = 2$ (8-PSK) or 4 (16-PSK) is accounting for the presence of the two- and four-constituent QPSK modulators in the specified modulations. The number of additions associated with this method (8-PSK) is $N_{+,M2}(r)_{8\text{-}PSK} = 2(r - 1) + 1 = 2r - 1$: twice the number of additions associated with partitioning by $r$ (two QPSK constituent modulators) plus the final addition of the two QPSK modulators. In the 16-PSK case, we get $N_{+,M2}(r)_{16\text{-}PSK} = 4(r - 1) + 3 = 4r - 1$. This expression can be generalized in terms of $C$ as $N_{+,M2}(r) = (C/2)r - 1$.

The following two factors should also be taken into account when evaluating complexity (nevertheless, as they can be applied to any of the three implementations described, they do not contribute to the comparison of their respective complexities):

1. For any partitioning strategy in either M1 or M2, the LUTs used by the in-phase or in-quadrature channels are equal. Hence, they may be shared depending on the clock rate.

2. The contents of any LUT have symmetry, as they store modulations with BPSK symbols ($\pm 1$'s): The binary address $A = [a_0 a_1 \ldots a_{L-1}]$ and its negation $\text{neg}(A) = [\text{neg}(a_0) \text{neg}(a_1) \ldots \text{neg}(a_{L-1})]$ contain words opposite in sign, such that $\text{LUT}(A) = -\text{LUT}(\text{neg}(A))$. With very simple logic, this helps reduce the total memory size to 50%. This symmetry is described in [3].

We may now compare architectures M1 and M2 in terms of:

1. Memory size versus the partitioning order $r$. Evaluations for the 8- and 16-PSK cases are provided by (23–8) and (23–9), respectively.

2. Memory size for the same number of additions. Necessarily, this will imply different partitioning orders $r$ for M1 and M2. This happens when

$$
N_{+,M2}(r') = (C/2) \cdot r' - 1 = N_{+,M1}(r) = r - 1
\tag{23–10}
$$

For comparison of type 2, we get from the previous equation that $r = (C/2)r'$. Under this constraint, the respective memory sizes of M1 in (23–8) and M2 in (23–9) become

$$P_{M2}(L, r') = (C/2) \cdot [(L - r' \cdot \lfloor L/r' \rfloor) \cdot 2^{\lfloor L/r' \rfloor + 1}$$
$$+ (r' - L + r' \cdot \lfloor L/r' \rfloor) \cdot 2^{\lfloor L/r' \rfloor}$$
$$= (C/2) \cdot (L - r' \cdot \lfloor L/r' \rfloor + r') \cdot 2^{\lfloor L/r' \rfloor}$$

$$P_{M1}\left(L, \frac{C}{2}r'\right) = \left(L - \frac{C}{2}r' \cdot \left\lfloor L \middle/ \frac{C}{2}r' \right\rfloor\right) \cdot C^{\left\lfloor L \middle/ \frac{C}{2}r' \right\rfloor + 1}$$

$$+ \left(\frac{C}{2}r' - L + \frac{C}{2}r' \cdot \left\lfloor L \middle/ \frac{C}{2}r' \right\rfloor\right) \cdot 2^{\left\lfloor L \middle/ \frac{C}{2}r' \right\rfloor} \qquad (23\text{–}11)$$

$$= \left((C - 1)\left(L - \frac{C}{2}r' \left\lfloor L \middle/ \frac{C}{2}r' \right\rfloor\right) + \frac{C}{2}r'\right) C^{\left\lfloor L \middle/ \frac{C}{2}r' \right\rfloor}$$

Particularizing for 8-PSK and 16-PSK, we get

$$P_{M2}(L, r')_{8\text{–PSK}} = 2 \cdot (L - r' \cdot \lfloor L/r' \rfloor + r') \cdot 2^{\lfloor L/r' \rfloor}$$
$$P_{M1}(L, 2r')_{8\text{–PSK}} = (3(L - 2r' \cdot \lfloor L/2r' \rfloor + 2r') \cdot 2^{2\lfloor L/2r' \rfloor}$$
$$P_{M2}(L, r')_{16\text{–PSK}} = 4 \cdot (L - r' \cdot \lfloor L/r' \rfloor + r') \cdot 2^{\lfloor L/r' \rfloor} \qquad (23\text{–}12)$$
$$P_{M1}(L, 4r')_{16\text{–PSK}} = (7(L - 4r' \cdot \lfloor L/4r' \rfloor) + 4r') \cdot 2^{3\lfloor L/4r' \rfloor}$$

We note that architecture M2 allows us to save memory space by reusing similar LUTs (pro-vided the clock is fast enough): If the multiplication factors $\tan(\pi/8)$, $\tan(\pi/16)$, and $\tan(\pi/8)\tan(\pi/16)$ are brought outside the LUTs (as in Figure 23-5), the four constituent QPSK modulators have all equal LUTs, which can be reused, allowing a reduction of 50% (8-PSK) and 75% (16-PSK) if each LUT is reused $C/2 =$ two (8-PSK) or $C/2 =$ four (16-PSK) times. Tables 23-1 and 23-2 present these results in terms of filter length $L$ and level of partitioning $r'$, considering the proposed LUT reuse factor $C/2$.

Table 23-1 provides the 8-PSK percentage of physical memory size of M2 (based on the LUT reuse factor $C/2$) with respect to M1 for the same number of additions $N_+$: $(C/2)^{-1}$ $P_{M2}(L, r')/P_{M1}(L, (C/2)r')$. Note that if LUT reuse is not implemented (multiply the percentages

**Table 23-1**  8-PSK Percentage of Physical Memory Size of M2

|        | $r' = 1$ | $r' = 2$ | $r' = 3$ | $r' = 4$ |
|--------|----------|----------|----------|----------|
| $L = 6$  | 0.5000 | 0.4000 | 0.5000 |        |
| $L = 7$  | 0.4000 | 0.4615 | 0.4444 |        |
| $L = 8$  | 0.5000 | 0.5000 | 0.4167 | 0.5000 |
| $L = 9$  | 0.4000 | 0.4286 | 0.4000 | 0.4545 |
| $L = 10$ | 0.5000 | 0.4000 | 0.4444 | 0.4286 |
| $L = 11$ | 0.4000 | 0.4615 | 0.4762 | 0.4118 |
| $L = 12$ | 0.5000 | 0.5000 | 0.5000 | 0.4000 |
| $L = 13$ | 0.4000 | 0.4286 | 0.4444 | 0.4348 |
| $L = 14$ | 0.5000 | 0.4000 | 0.4167 | 0.4615 |
| $L = 15$ | 0.4000 | 0.4615 | 0.4000 | 0.4828 |
| $L = 16$ | 0.5000 | 0.5000 | 0.4444 | 0.5000 |
| $L = 17$ | 0.4000 | 0.4286 | 0.4762 | 0.4545 |
| $L = 18$ | 0.5000 | 0.4000 | 0.5000 | 0.4286 |

**Table 23-2**    16-PSK Percentage of Physical Memory Size of M2

|  | $r'=1$ | $r'=2$ | $r'=3$ | $r'=4$ |
|---|---|---|---|---|
| $L=6$ | 0.4433 [64] | | | |
| $L=7$ | 0.6400 [128] | | | |
| $L=8$ | 1.0000 [256] | 0.5000 [32] | | |
| $L=9$ | 0.7273 [512] | 0.4000 [48] | | |
| $L=10$ | 0.8889 [1024] | 0.3636 [64] | | |
| $L=11$ | 1.2800 [2048] | 0.4138 [96] | | |
| $L=12$ | 2.0000 [4096] | 0.4444 [128] | 0.5000 [48] | |
| $L=13$ | 1.4545 [8192] | 0.5581 [192] | 0.4211 [64] | |
| $L=14$ | 1.7778 [16384] | 0.6400 [256] | 0.3846 [80] | |
| $L=15$ | 2.5600 [32768] | 0.8421 [384] | 0.3636 [96] | |
| $L=16$ | 4.0000 [65536] | 1.0000 [512] | 0.4000 [128] | 0.5000 [64] |
| $L=17$ | 2.9091 [131072] | 0.8000 [768] | 0.4255 [160] | 0.4348 [80] |
| $L=18$ | 3.5556 [262144] | 0.7273 [1024] | 0.4455 [192] | 0.4000 [96] |

by two), there exists still a slight improvement in memory size. Short lengths are used for modulation pulses with a high rolloff factor.

Table 23-2 provides the 16-PSK) percentage of physical memory size of M2 (based on the LUT reuse factor $C/2$) with respect to M1 for the same number of additions $N_+$: $(C/2)^{-1}$ $P_{M2}(L,r')/P_{M1}(L,(C/2)r')$. The memory size of architecture M2 is shown in brackets. Those combinations of filter length and partitioning order where M2 does not reduce memory size are underlined (note that this happens only for long filter lengths, which are required only for atypical, very small rolloff factors of the modulation pulse). Nevertheless, subsequent partitioning (larger $r'$) does already offer an improvement in memory size for the same number of additions.

We observe that when comparing M1 and M2 with respect to the same number of additions, architecture M2 (8-PSK, Table 23-1) shows a slight improvement, which is substantial in the case of LUT reuse. We also observe that architecture M2 (16-PSK, Table 23-2) provides reductions in memory size with respect to M1 when compared at the same number of additions.

So far we have seen that M2 leads to savings in complexity when memory is reused. We may compare it now with the classical distributed arithmetic scheme described in the introduction, which is also based on memory reuse.

3. **Architecture M3.** This is a typical DA processor, where input symbols are quantized to $B$ bits. The wordlength $B$ assigned to the quantized input symbols must be sufficiently long to reduce the quantization error of 8-PSK or 16-PSK to a level similar to that in M1 or M2. If we use $Q$ for the memory reuse factor, the DA processor performs $Q=B$ iterations (memory lookup and shift-accumulation operations) over the same memory. For the same partitioning strategy of M2, the physical memory is the same (it stores BPSK modulated outputs, with $P_{M3}(L)=2^L$ when no partitioning is used). Nevertheless, for M2 the memory reuse factor was found to be $Q=C/2=2$ (8-PSK) and $Q=C/2=4$ (16-PSK). It is clear that M3 requires $Q=B>>4$ for a good quantization signal-to-noise ratio (SNR). Hence, although the physical memory in M3 is the same as in M2, more iterations are required.

For a typical wordlength of $B=8$ bits, the classical DA-8-PSK scheme requires four times as many iterations as the optimized 8-PSK scheme M2 (two iterations), whereas the DA-16-PSK scheme requires twice as many iterations as the optimized 16-PSK scheme M2 (four iterations). Hence for 8-PSK and 16-PSK we can modulate at 25% and 50% of the cost achievable with M3.

The relationship that may be established between M2 and M3 is the fact that M2 can be viewed as a quantization of input symbols on an irregular quantization grid (see the grid shown in Figure 23-3 as an irregular 256-QAM constellation), in contrast to method M3, which uses a regular quantization grid and thus requires more iterations (a finer grid) to achieve a similar quantization SNR. Hence, the equivalent in method M2 of the shift-accumulate operations on the memory lookups in method M3 is found in the output weighting network of the four LUTs in Figure 23-5.

## 23.7 CONCLUSIONS

We have shown a construction method for 8- and 16-PSK symbols based on a superposition of two and four QPSK constellations, respectively, that leads to reductions in the physical memory size of DA implementations of the corresponding modulators. The comparison has been established versus the partitioning order of (23–8) and (23–9) and for the same number of additions (Tables 23-1 and 23-2).

The proposed scheme is also very flexible in the sense that it can operate as a multipurpose modulator, which, with minimum hardware reconfigurability, can generate a number of different linear modulations. The structure is shown in Figure 23-5, where the choice of the adequate three external multiplications provides for generation of: BPSK, QPSK, 8-PSK, 16-PSK, 16-QAM, 32-QAM, 64-QAM, 128-QAM, and 256-QAM. An input interface module allows us to translate the input symbol encoding to the respective QPSK pseudosymbol encoding.

Architecture M1 has been used as a benchmark for comparison as, disregarding complexity (LUT size), M1 is associated with best performance in terms of quantization SNR (without partitioning, it stores the direct quantization of the modulator outputs). In this comparison, architecture M2's SNR is degraded with respect to M1 only a fraction of one quantization bit (1 bit = 6 dB) as shown in the results presented in Figure 23-8.

Figure 23-8(a) shows the SNR loss (in dB) of architecture M2 with respect to M1: $L(\text{M1,M2}) = \text{SNR}_{M1,B_o} - \text{SNR}_{M2,B_o}$ dB, for a final rounding to $B_o = 9$ bits (circles) and $B_o = 10$ bits (crosses). Memory reductions allowed by architecture M2 are traded off with a small degradation in quantization SNR with respect to M1, where a loss of 6 dB in quantization SNR corresponds to one-bit resolution. Thus, incurred degradation is just a fraction of a quantization bit. Results obtained with quantization analyses are usually sensitive to low-level optimizations: different choices for partitioning or the configuration of the arithmetic. In this simulation, filter partitioning assigns contiguous samples of the original pulse shape to each subfilter.
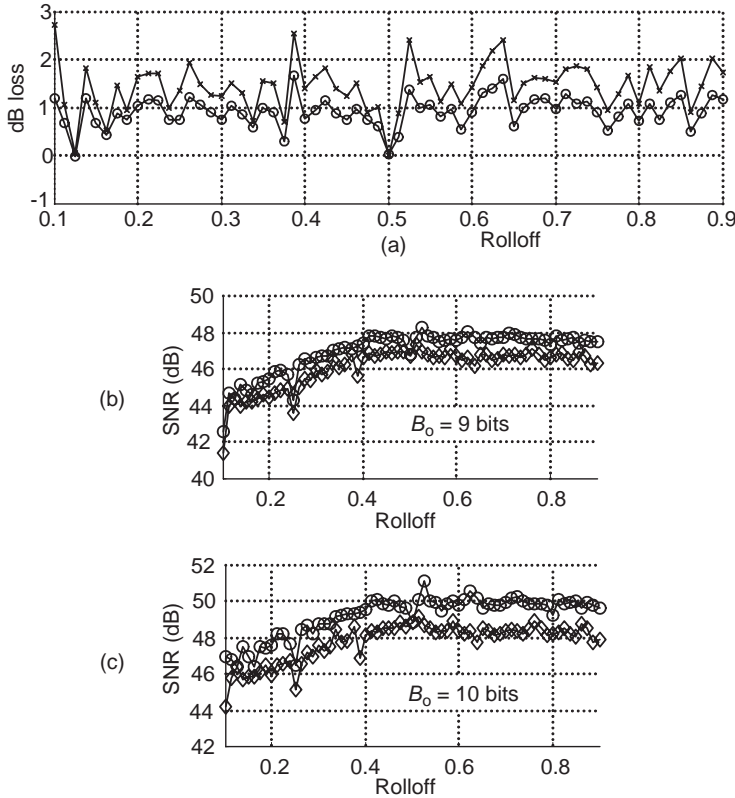
**Figure 23-8** SNR comparison of M2 and M1 architectures.

Figures 23-8(b) and (c) shows the quantization SNR versus rolloff (the shaping pulse is a square-root raised-cosine filter) for architectures M2 (diamonds) and M1 (circles), when partitioning is adjusted for both to have the same number of additions and $r' = 1$. The wordlength of the LUT values is 8 bits and the summation network of LUT outputs performs a final rounding to $B_o = 9$ bits in Figure 23-8(b), and $B_o = 10$ bits in Figure 23-8(c). The pulse shape is sampled at four samples per symbol and has $7 \times 4 + 1 = 29$ coefficients, which corresponds to entry ($L = 7$, $r' = 1$) in Table 23-2. The additive complexity is four additions.

## 23.8 REFERENCES

[1] D. KLYMYSHYN and D. HALUZAN, "FPGA Implementation of Multiplierless M-QAM Modulator," *Electronic Letters*, vol. 38, no. 10, 9 May 2002, pp. 461–462.

[2] A. PELED and B. LIU, "A New Hardware Realization of Digital Filters," *IEEE Trans. on ASSP*, vol. ASSP-22, no. 6, December 1974, pp. 456–461.

[3] S. WHITE, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE ASSP*, July 1989, pp. 4–19.

[4] M. RUPP and J. BALAKRISHNAN, "Efficient Chip Design for Pulse Shaping," *2nd Workshop on Signal Processing Advances in Wireless Communications* (SPAWC), 9–12 May 1999, pp. 304–307.

# Assorted High-Performance DSP Techniques

# Chapter 24

# Frequency Response Compensation with DSP

**Laszlo Hars**

*Seagate Research*

In modern telecommunication systems there are situations when test instruments must work over hundreds of narrowband-frequency channels. However, it is difficult and expensive to build hardware test equipment with flat frequency response over their full operational frequency range. In this situation simple FIR filters can be applied for gain compensation to improve an instrument's frequency response flatness. This chapter describes a very fast method for run-time design of these filters, while minimizing storage requirements, based on test instrument calibration data [1]–[7].

## 24.1 FILTER TABLE

When a particular center frequency for a channel-under-test is selected, a simple FIR filter can be used to improve test instrument gain flatness. However, computing and storing the necessary filter coefficients is often impractical. The frequency responses of the filters depend on the test center frequency, and this would require storing thousands of sets of the filter coefficients along with a table telling us which filter is needed for each center frequency. The filter coefficients have to be determined at calibration time. If the center frequency can be chosen with high precision, this precalculated filter approach is unattractive because a set of filter coefficients is needed for each center frequency, requiring huge storage space and a very long calibration time.

## 24.2    RUN-TIME FILTER DESIGN

A better gain compensation filtering approach is to measure the amplitude characteristics of the test instrument on a sufficiently dense frequency grid when the instrument gets calibrated. Only this table has to be stored. Using interpolation (linear, cubic, or spline—depending the smoothness of the frequency-response curve), the required gain compensation filter response can be determined with a handful of arithmetic operations at run time. The compensation filter can be very short if the sampling rate and center frequency are chosen appropriately. Given the required compensation filter gain, we developed a closed-form expression for the filter coefficients allowing us to compute them with only a handful of operations. All together the calculations are so fast that even arbitrary frequency hopping can be implemented with slow, low-power DSPs.

## 24.3    CALIBRATION TABLES

In most applications, the analog input signal has to be attenuated or amplified by circuits that are themselves not perfect, either. In theory, we would need a frequency-response table for each possible attenuation setting. In practice, however, the frequency response varies only at smaller attenuation values; higher values provide good decoupling between otherwise-interfering circuit parts. As such, in practice roughly 10 compensation tables are often enough for even high-precision measurements. The effects of different attenuation devices are cumulative at proper design (at least at higher attenuation values), which further reduces the number of necessary tables. Temperature compensation can be incorporated, too. The measured ambient or internal temperature represents another dimension for the family of tables.

## 24.4    FIR VERSUS IIR FILTERS

IIR filters have more complicated formulas for their gain response than FIR filters; therefore real-time calculation of the IIR coefficients takes longer. We normally also need constant group delay in the passband, which is more difficult to achieve with IIR filters. The filters must be stable; that is, the roots of the denominator of the transfer function of IIR filters must all lie inside the complex unit circle, and this is another difficulty to be dealt with. However, the same long IIR filters can somewhat better approximate a given gain curve. If this curve has very sharp peaks, notches, or edges, IIR filters are needed. Also, the FIR filters can have large group delays. If the group delay must be small or even negative, IIR filters have to be used.

The little better approximation of the desired amplitude curve by an IIR filter can be balanced by applying a longer FIR filter, whose coefficient calculations are simpler. The filtering takes about the same time because of the faster FIR filter code.

Therefore FIR filters are a good choice. They must be short, having less than 10 coefficients; otherwise the calculation of the coefficients gets complicated.

## 24.5  LINEAR PHASE FIR FILTERS

The frequency response function of an $N$-tap FIR filter with coefficient sequence $c_0, c_1, \ldots, c_{N-1}$ is given by

$$H(\omega) = \sum_{k=0}^{N-1} c_k \cdot e^{-jk\omega} \qquad (24\text{--}1)$$

If the filter has linear phase response (constant group delay), the coefficient sequence must be symmetric or antisymmetric. We want a filter with relatively flat amplitude response, that is, close to unity gain everywhere, also at DC. If the coefficient sequence is antisymmetric ($c_k = -c_{N-1-k}$), the DC gain is 0; therefore we need a symmetric coefficient sequence. The filter delay is $(N-1)/2$. It is easier to accommodate an integer number of filter sample delays, which is what we have if $N$ is odd. In this case, the response function becomes

$$H(\omega) = e^{-j(N-1)\omega/2} \times \left[ c_{(N-1)/2} + 2 \sum_{k=(N+1)/2}^{N-1} c_k \cdot \cos\left[\left(k - \frac{N-1}{2}\right)\omega\right] \right]. \qquad (24\text{--}2)$$

The factor $e^{-j(N-1)\omega/2}$ represents the delay having a constant magnitude of 1, and the real factor in the brackets gives the (signed) amplitude response. The length of the filter can be chosen to be $N=7$, and it has four free coefficients. We need the desired gain to be a given value at three different frequencies, and one degree of freedom remains to enforce a smooth amplitude response curve. Let the coefficient sequence be $[c, b, a, d, a, b, c]$. From (24–2), the amplitude response of the corresponding FIR filter is

$$A(\omega) = d + 2a \cdot \cos(\omega) + 2b \cdot \cos(2\omega) + 2c \cdot \cos(3\omega). \qquad (24\text{--}3)$$

## 24.6  SAMPLING AND SIGNAL FREQUENCY

Usually, if some signal conversion is performed before the amplitude response correction, we can simplify the design and save processing time. We need mixing and decimation to reduce the sampling rate $f_{Samp}$ to a little above double the signal bandwidth. This is the minimum, which preserves all the information of the original signal (Nyquist theorem). It usually involves a bandpass filter step, too, removing those disturbing signals that would alias to the useful frequency band. This frequency band is best located around the center frequency $f_0$, where $f_0 = f_{Nyq}/2 = f_{Samp}/4$ (or at $3f_{Nyq}/2$), such that no signal component aliases back into the useful frequency band at another location. (On the normalized scale where the Nyquist frequency is 1, $\omega = \pi \cdot f$ and $f_0 = 1/2$.) Using a normalized frequency axis, the FIR filter's amplitude response can be expressed

$$A(f) = d + 2a \cdot \cos(\pi f) + 2b \cdot \cos(2\pi f) + 2c \cdot \cos(3\pi f). \qquad (24\text{–}4)$$

## 24.7   FILTER DESIGN

When applying the compensation filter we do not want to change the amplitude at $f_0$, the center of the frequency band of the signal. (If necessary, we adjust the overall gain outside of the filter.) For the frequency-response compensation we specify the gains $g_1$ and $g_2$ of the filter at two other frequencies, say $f_1 = 1/4$ and $f_2 = 3/4$ at both sides of $f_0$. (They can be chosen closer or further away from $f_0$, according to the need to have more accurate compensation close to the center frequency or less accurate compensation over the whole band.) These represent three constraints, unity gain at $f_0$ and gains $g_1$ and $g_2$ at $f_1$ and $f_2$, for the filter response, which is a function of four free coefficients. Using (24–4) we can express the three amplitude constraints as:

$$1 = d + 2a \cdot \cos(\pi/2) + 2b \cdot \cos(\pi) + 2c \cdot \cos(3\pi/2). \qquad (24\text{–}5)$$
$$g_1 = d + 2a \cdot \cos(\pi/4) + 2b \cdot \cos(\pi/2) + 2c \cdot \cos(3\pi/4). \qquad (24\text{–}6)$$
$$g_2 = d + 2a \cdot \cos(3\pi/4) + 2b \cdot \cos(3\pi/2) + 2c \cdot \cos(9\pi/4). \qquad (24\text{–}7)$$

A very important additional requirement is that the filter must not have large ripple (i.e., its amplitude response must be smooth). This can be guaranteed if we mandate a fourth constraint to require the slope of the response curve at $f_0$ be the same as that of the secant line connecting the points $[f_1, g_1]$ and $[f_2, g_2]$. The slope of the curve, the derivative, tells how steep the response curve is in the neighborhood of a given point. It is the same as the slope of the tangent line of the filter response curve. The slopes of the secant lines $[f_1, g_1]$ and $[f_2, g_2]$ around a point $[f_0, 1]$ approximate the slope of the tangent $A'(f_0)$ of the (hopefully smooth) function $A$ in (24–4). If we require an exact equality, it intuitively ensures some kind of smoothness. Of course, you can specify more complicated conditions, but in our case the following equality proved sufficient. That is:

$$\frac{g_1 - g_2}{f_1 - f_2} = A'(f_0) = \frac{d[A(f)]}{df}$$
$$= -2\pi a \cdot \sin(\pi f_0) - 4\pi b \cdot \sin(2\pi f_0) - 6\pi c \cdot \sin(3\pi f_0). \qquad (24\text{–}8)$$

Because $f_1 - f_2 = -1/2$, and $f_0 = 1/2$, we have

$$g_1 - g_2 = \pi a \cdot \sin(\pi/2) + 2\pi b \cdot \sin(\pi) + 2\pi c \cdot \sin(3\pi/2). \qquad (24\text{–}9)$$

This last requirement, constraint (24–9), now gives us a linear system of four equations for the four unknown coefficients of the filter.

Evaluating (24–5) through (24–7) and (24–9) for the real values of the trigonometric functions gives:

$$1 = d - 2b. \qquad (24\text{–}10)$$

$$g_1 = d + \sqrt{2}a - \sqrt{2}c. \qquad (24\text{–}11)$$

$$g_2 = d - g_2 = d - \sqrt{2a} + \sqrt{2c}. \qquad (24\text{–}12)$$

$$g_1 - g_2 = \pi a - 3\pi c. \qquad (24\text{–}13)$$

Solving those equations, we get a simple solution for our filter coefficients:

$$a = (g_1 - g_2)\left(\frac{3\sqrt{2}}{8} - \frac{1}{2\pi}\right). \qquad (24\text{–}14)$$

$$b = \frac{g_1 + g_2}{4} - \frac{1}{2}. \qquad (24\text{–}15)$$

$$c = (g_1 - g_2)\left(\frac{\sqrt{2}}{8} - \frac{1}{2\pi}\right). \qquad (24\text{–}16)$$

$$d = \frac{g_1 + g_2}{2}. \qquad (24\text{–}17)$$

So, based on predetermined test instrument calibration data stored as an array of $g_1$ and $g_2$ values versus center frequency, the 7-tap FIR filter's (24–13) through (24–16) coefficients are computed and used in real-time as new center frequencies are assigned during test instrument operation.

## 24.8   MATLAB SIMULATION

The following function, in MATLAB code, calculates the desired filter coefficients based on the desired gains at the frequencies $f_0 = 1/2$, $f_1 = 1/4$, and $f_2 = 3/4$.

```
function v=relatflt(dB1,dB2)
%relatflt(dB1,dB2) len=7 FIR filter of response:
% 0 dB gain at f0=1/2
% dB1=gain compensation in dB at f1=1/4
% dB2=gain compensation in dB at f2=3/4
g1=10^(dB1/20); % convert dB gain to linear
g2=10^(dB2/20); % convert dB gain to linear
 a=(g1-g2)*0.37117514279802; % 3sqrt(2)/8-1/2/pi
 b=(g1+g2)*0.25-0.5;
 c=(g1-g2)*0.01762175220474; % sqrt(2)/8-1/2/pi
 d=(g1+g2)*0.5;
 v=[c b a d a b c];
```

Using the above code to compute and plot filter responses of linear compensation curves with dB gains of [0.4,–0.4], [0.2,–0.2], [0,0], [–0.2,0.2], [–0.4,0.4], we have those shown in Figure 24-1. The desired gain compensation values are indicated by the dots.

Example curves with decreasing gains of [0.5,–0.4], [0.3,–0.4], [0.1,–0.4], [–0.1,–0.4], [–0.3,–0.4], in dB, are provided in Figure 24-2.
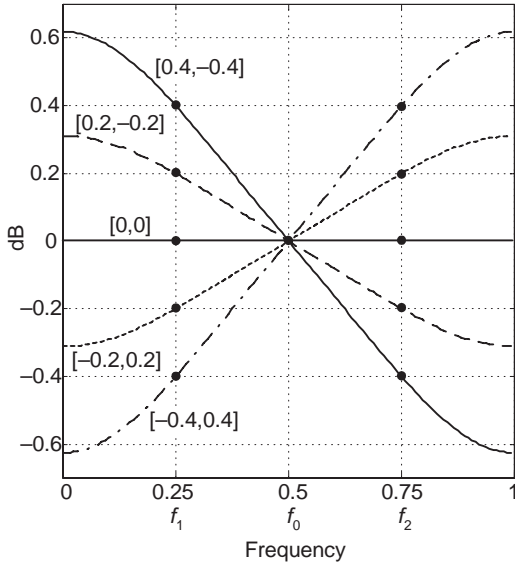
**Figure 24-1**     Example gain compensation curves for the $N=7$ FIR filter.
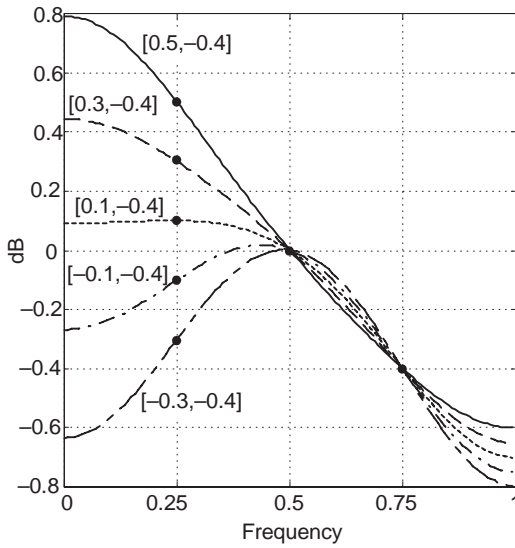


**Figure 24-2**     Example gain compensation curves for decreasing gain versus frequency.
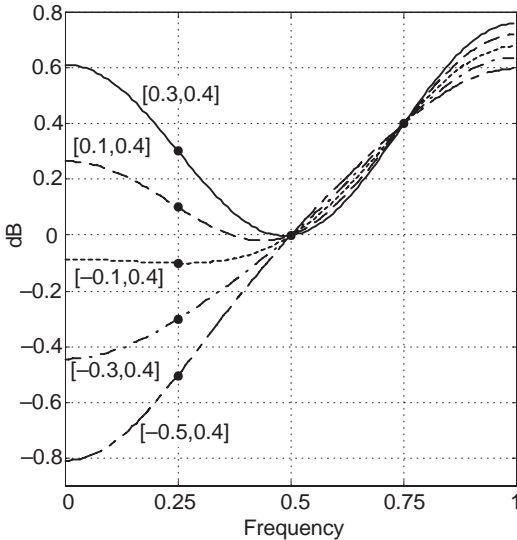
**Figure 24-3**  Example gain compensation curves for increasing gain versus frequency.

Finally, example curves with increasing gains of [0.3, 0.4], [0.1,0.4], [−0.1,0.4], [−0.3,0.4], [−0.5,0.4], in dB, are shown in Figure 24-3.

They agree completely with the gain compensation we wanted.

## 24.9  IMPLEMENTATION IN C

Calculation of the FIR filter coefficients is straightforward in the C language as shown below. The floating-point variables **d1** and **d2** specify the desired filter gains in dB at $f_1 = 1/4$ and $f_2 = 3/4$. The filter coefficients are stored after calculation in the array **Filt[Len]**, with **Len = 7**.

```c
#define Len 7
#define C ((Len-1)/2)
/* from dB to ratio: */
float g1=pow( 10.0, d1 / 20.0);
float g2=pow( 10.0, d2 / 20.0);
float Filt[Len];
/* 3*sqrt(2)/8-1/2/pi=0.37117514279802 */
/* sqrt(2)/8-1/2/pi=0.01762175220474 */
Filt[C-1] =Filt[C+1]=(g1-g2)*0.37117514279802;
Filt[C-2] =Filt[C+2]=(g1+g2)*0.25-0.5;
Filt[C-3] =Filt[C+3]=(g1-g2)*0.01762175220474;
Filt[ C ]=(g1+g2)*0.5;
```

## 24.10   EXTENSIONS

The designed filters work well even beyond a $\pm 6$ dB compensation range. These large flatness errors, however, should not occur. They indicate serious test instrument hardware faults. If the shape of the compensation curve needs to be more complex, we can easily add to the constraints a second pair of frequencies with specified gains. We should request the slope of the response curve at the innermost frequency points be equal to the slope of the secant going through the surrounding specified curve points. We need now a 15-tap filter of the form: $a_7$, $a_6$, $a_5$, $a_4$, $a_3$, $a_2$, $a_1$, $a_0$, $a_1$, $a_2$, $a_3$, $a_4$, $a_5$, $a_6$, $a_7$.

Let $f_k = k\pi/6$ be the five frequencies where the gain will be specified, thus $k = 1$, $2, \ldots, 5$, and the corresponding gain values are $g_k$ normalized at the center with $g_3 = 1$. We have five equations describing the correction gains, similar in form to (24–5) through (24–7):

$$g_k = a_0 + 2\sum_{i=1}^{7} a_i \cos(i\pi/6), \quad k = 1, \ldots, 5, \tag{24–18}$$

and another three, requiring the correct slopes, similar in form to (24–9):

$$g_k - g_{k+2} = (2\pi/3)\sum_{i=1}^{7} a_i \sin(i\pi/6), \quad k = 1, 2, 3. \tag{24–19}$$

The solution of the corresponding linear system of equations goes somewhat as before:

$$a_0 = \frac{-3\sqrt{3}(g_1 - 2g_3 + g_5) + (5g_1 + 3g_2 - 4g_3 + 3g_4 + 5g_5)\pi}{12\pi}. \tag{24–20}$$

$$a_1 = \frac{[3(g_2 - g_4) + \sqrt{3}(g_1 - g_5)](-2 + \pi)}{4\pi}. \tag{24–21}$$

$$a_2 = \frac{(g_1 - 2g_3 + g_5)(-3\sqrt{3} + 4\pi)}{12\pi}. \tag{24–22}$$

$$a_3 = \frac{[3(g_2 - g_4) + \sqrt{3}(g_1 - g_5)](-3 + \pi)}{6\pi}. \tag{24–23}$$

$$a_4 = \frac{(g_1 - 2g_3 + g_5)(-3\sqrt{3} + 2\pi)}{12\pi}. \tag{24–24}$$

$$a_5 = \frac{72g_4 + 27\sqrt{3}g_5 + 36g_2(-2 + \pi) - 36g_4 - 5\sqrt{3}g_5 + \sqrt{3}g_1(-27 - 5\pi)}{72\pi}. \tag{24–25}$$

$$a_6 = \frac{-3\sqrt{3}(g_1 - 2g_3 + g_5) + (g_1 + 3g_2 - 8g_3 + 3g_4 + g_5)\pi}{24\pi}. \tag{24–26}$$

$$a_7 = \frac{-3[3g_1 + 4\sqrt{3}(g_2 - g_4) - 3 - g_5] + [g_1 + 6\sqrt{3}(g_2 - g_4) - g_5]\pi}{24\sqrt{3}\pi}. \tag{24–27}$$

It is more complex, requiring a little more processor power for the run-time design, but the work is still manageable. The following MATLAB code calculates the

desired 15-tap filter coefficients with the like terms in (24–20) through (24–27) evaluated and $g_3 = 1$:

```
function v=flt15(dB1,dB2,dB4,dB5)
%flt15(dB1,dB2,dB4,dB5) len=15 FIR filter response
% dB1=gain in dB at    f1=1/6%
% dB2=gain in dB at    f2=2/
% 0 dB gain at f3=3/6
% dB4=gain in dB at f4=4/6
% dB5=gain in dB at f5=5/6
g1=10^(dB1/20); g4 =10^(dB4/20);
g2=10^(dB2/20); g5 =10^(dB5/20);
a0=0.27883*(g1+g5) +0.25000*(g2+g4)-0.05767;
a1=0.15735*(g1-g5) +0.27254*(g2-g4);
a2=0.19550*(g1+g5) -0.39100;
a3=0.01301*(g1-g5) +0.02254*(g2-g4);
a4=0.02883*(g1+g5) -0.05767;
a5=-0.08647*(g1-g5)+0.18169*(g2-g4);
a6=-0.02725*(g1+g5)+0.12500*(g2+g4)-0.19550;
a7=-0.04486*(g1-g5)+0.09085*(g2-g4);
 v=[a7 a6 a5 a4 a3 a2 a1 a0 a1 a2 a3 a4 a5 a6 a7];
```

Figure 24-4 shows a few example compensation response curves of 15-tap FIR filters designed using the above code. Again, the desired gain compensation values are indicated by the dots.
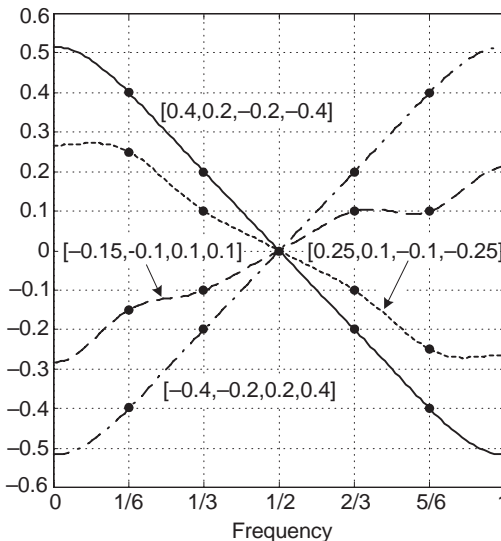


**Figure 24-4**   Example gain compensation curves for a 15-tap FIR filter.

## 24.11 CALIBRATION TABLES

Usually the input signal gets attenuated and/or amplified before the analog-to-digital conversion to assure maximum digital resolution. These attenuator-amplifier circuits affect each other to a varying degree, dependent on the selected attenuation. In theory, we need a frequency-response calibration table for each possible attenuation value. If an instrument needs to be calibrated at several different attenuation levels over the whole frequency range, it will consume a lot of time and cost. However, proper design reduces the number of necessary calibration runs. In practice, without significant effort in the hardware design the frequency characteristics change only at smaller attenuation values, because higher attenuations provide good decoupling between otherwise-interfering circuit parts. The effects of decoupled cascaded attenuation devices are cumulative (additive when measured in dB), which further reduces the number of necessary tables.

Temperature compensation can be incorporated, too. The measured ambient or internal temperature represents another dimension for the family of tables. If the temperature dependency is smooth (linear or close to that), and does not change the shape of the frequency-response curve, one extra table is enough for high-precision compensation. The correction can be done by a temperature-dependent multiplication factor (additive in dB).

## 24.12 REFERENCES

[1] A. POULARIKAS, *Formulas and Tables For Signal Processing*. CRC Press–IEEE Press, New York, 1999.

[2] N. KALOUPTSIDIS, *Signal Processing Systems: Theory and Design*. John Wiley & Sons, New York, 1997.

[3] J. PROAKIS and D. MANOLAKIS, *Digital Signal Processing: Principles, Algorithms and Applications*. 3rd. ed. Prentice Hall, Englewood Cliffs, NJ, 1996.

[4] N. FLIEGE, *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. John Wiley & Sons, New York, 1995.

[5] W. PRESS, S. TEUKOLSKY, W. VETTERLING, and B. FLANNERY; *Numerical Recipes in C*, Cambridge University Press, Cambridge, MA, 1992.

[6] L. RABINER and B. GOLD; T*heory and Application of the Digital Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1975.

[7] A. OPPENHEIM and R. SCHAFER; *Digital Signal Processing, Prentice Hall, Englewood Cliffs*. NJ, 1975.

# Chapter 25

# Generating Rectangular Coordinates in Polar Coordinate Order

**Charles Rader**

*Retired, formerly with MIT Lincoln Laboratory*

$\mathbf{W}$hen we deal with two-dimensional data, we almost always locate the data using either of two coordinate systems, rectangular coordinates or polar coordinates. Converting from one representation to the other can be a major nuisance. For example, suppose we have a function $f(r,\theta)=(x,y)=g(r\cos\theta, r\sin\theta)$ and we can measure the data using only a polar coordinate measuring device even though we want to do our processing using rectangular coordinates. Here we present an elegant algorithm for computing rectangular indexes in order of increasing polar angle.

## 25.1 CENTERED GRID ALGORITHM

For example, suppose I am standing on the origin of a piece of ruled graph paper with intersections at $(x,y)=(m,n)$; $0\leq m\leq L$; $0\leq n\leq M$ and I am holding a laser pointer. It is initially pointed along the $x$ axis and illuminating the set of graph intersection points $(x,y)=(1,0),(2,0), \ldots , (L,0)$. As I move my laser pointer slowly counterclockwise, the next intersection I illuminate will be $(L,1)$, as shown in Figure 25-1, and shortly after that it will illuminate $(L\text{-}1,1)$, and so on.

Consider the illustrated case of $L=M=5$. The table and the diagram in Figure 25-2 give the order in which points will be illuminated (the solid dots—the open dots are shadowed by the solid dots on the same radial line). This example illustrates
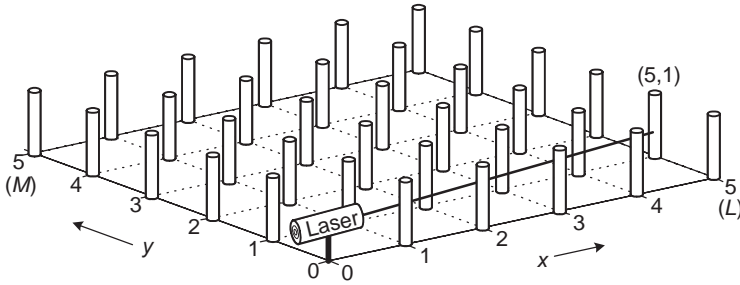
**Figure 25-1** Rectangular coordinates and laser pointer illuminating point (5,1).
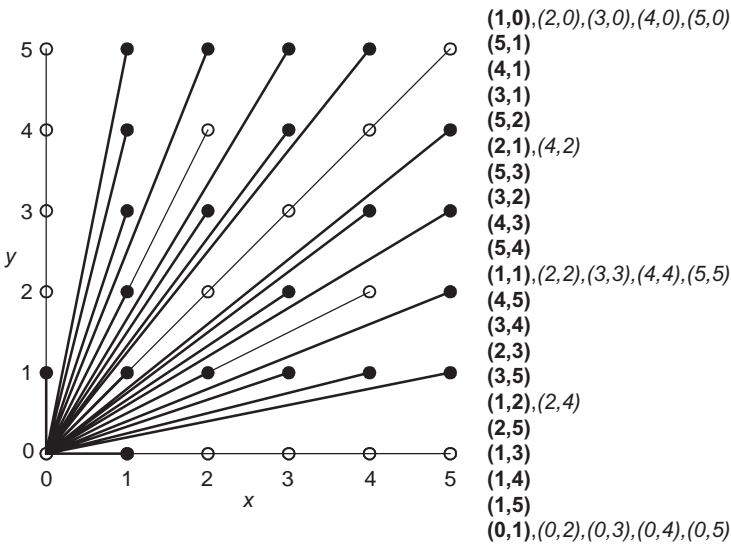


**Figure 25-2** Illustrating the polar coordinates order of grid points $(m,n)$; $0 \leq m, n \leq 5$.

that the order in which data becomes available with a polar coordinate measuring device is most unnatural from the point of view of a rectangular coordinate system. If $L$ is fairly large, this order becomes more and more obscure.

Normally, that won't matter. As soon as we make a measurement, we can store it in a random access memory such that $g(m,n)$ is in memory location $Kn+m$ (where $K$ is any integer larger than $L$ and will most likely be chosen as a power of two). Once the data is in the large memory, the rectangular coordinate system representation is achieved, and the order in which the data was accessed does not matter.

But suppose it does matter! I can think of two reasons we might care about the order. First, after we have measured the data at, say, (3,5) we next need to point the laser beam in the direction of (1,2) (and, on the same radial line, (2,4)). That is, we need to point in the direction with $\theta = \tan^{-1}(2/1)$. In general, we need to know

the pointing angles $\theta = \tan^{-1}(n/m)$ in their natural order. (Once we have reached that angle, the radii $r$ of the intersections will be all the multiples of sqrt($m^2 + n^2$) with $m$ not larger than $L$.

Second, maybe we are simply given the data in a polar coordinate order, measured, say, every 5°. It will be necessary to use some sort of interpolation procedure to obtain the values of the function on rectangular coordinates. But interpolation procedures compute the value of a function based on nearby values. In the simplest case, we might want the value assigned to $(m,n)$ to be the nearest measurement in $r,\theta$. Or, we might want to use a weighted sum of the two closest points, and so forth. If we were to perform these interpolations for $(m,n)$ taken in a natural rectangular order, we would need to read each appropriate polar coordinate datum into local memory several times. For example, data at angle 25° provides the nearest neighbors for (2,1), (4,2), (6,3), (7,3), (8,4). But if we schedule the interpolation for points $(m,n)$ in their natural polar coordinate order, we need only to keep in local memory the measurements along two successive angles, find the nearest neighbor (or several nearest neighbors) for all $(m,n)$ pairs between those two angles, then drop the data for the smaller angle and read in data for the next larger angle, and so on.

For these reasons, it would be nice if we had an algorithm that generates the points $(m,n)$ of a rectangular coordinate system in the natural order in which they would be obtained in a polar coordinate system.

Admittedly, if this need arises, it is easily met by constructing a table, just like in Figure 25-2. We present here an attractive alternative.

Let's note that in the table, whenever the point $(m,n)$ occurs, other points $(km,kn)$ in the rectangle lie on the same line so long as neither $kn > M$ nor $km > L$. Since every intersection in the square appears once, it follows that we can limit our attention to producing intersections $(m,n)$ for which $m,n$ have no common divisor. When the laser points at $(m,n)$, the point nearest to the origin in that beam direction, it also points at other intersections $(km,kn)$ at the same angle, and we need to identify only the first intersection, the one nearest to the origin.

Here is the algorithm for generating the mutually prime $(m,n)$ pairs in order of increasing $\theta$, using MATLAB notation:

```
a=0;b=1;c=1;d=L; %initialization
m=[b d];
n=[a c];
while~((c==M)&(d==1)) % test for end
  Z=floor((L+b)/d);
  e=Z*c-a;
  f=Z*d-b;
  if e<=M
    % report pair (e,f)
    m=[m f];
    n=[n e];
  end
  b=d;d=f;a=c;c=e;
end
plot(m,n,'-.o')
axis([0 L 0 M])
```
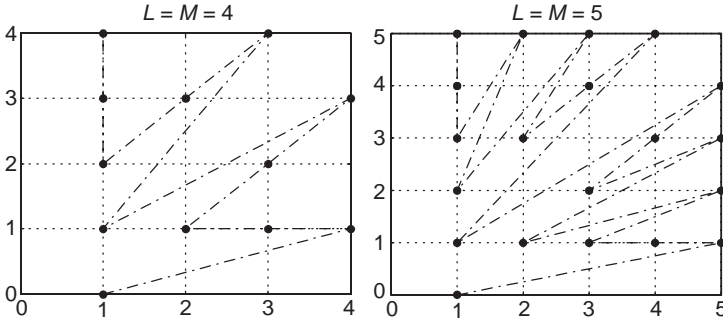
**Figure 25-3**    Algorithm (*m*,*n*) pairs.

Figure 25-3 shows the result of the code for $L=M=4$ and $L=M=5$.

Of course, the place in the program where the pair *m*,*n* is reported is the place where another routine can do whatever needs to be done with that pair.

As presented above, the algorithm generates (*m*,*n*) points in $0° \leq \theta \leq 90°$. But we have some flexibility. The routine can be initialized with any two successive angles $\theta_1 = \tan^{-1} n_1/m_1$, $\theta_2 = \tan^{-1} n_2/m_2$ and it can be ended at another angle $\theta_k = \tan^{-1} n_k/m_k$ we expect it to reach.

Therefore it is also possible to control the algorithm as a subroutine call. The subroutine inputs are *L*, and the last two pairs it reported, (*a*,*b*) and (*c*,*d*). The subroutine computes $Z=(L+b)/d$ and returns $e=Zc-a$ and $f=Zd-b$. However, the points with $e>M$ must not be left out, so *M* is not used within the subroutine.

As the laser beam angle increases, more and more points are skipped by the test $e \leq M$. Assuming $M \geq L$, the algorithm will never skip for angles less than 45°. So if we are interested only in $0 \leq \theta \leq \pi/4$, *M* becomes irrelevant and we can use

```
a=0; b=1; c=1; d=L; %initialization
% report pair a,b
% report pair c,d
while ~((c==1)&(d==1)) % test for end
  Z=floor((L+b)/d);
  e=Z*c-a;
  f=Z*d-b;
  % report pair e,f
  b=d; d=f; a=c; c=e;
end
```

It is possible to use the 45° generating routine to fill the entire square by exploiting a symmetry. First we generate (*m*,*n*) points as above, from (0,1) to (1,1). Then we replace the initialization statement by

```
a=1;b=1;c=L-1;d=L; %initialization
```

and replace the stop test by

```
while ~((c==0)&(d==1)) % end test
```

If we now continue the program it generates the original set of $(m,n)$ points in reverse order. *Yes, the algorithm runs backward!* But now we can swap $(m,n) \rightarrow (n,m)$ and the resulting points fill the 45° to 90° slice of the desired rectangle. To go past 90° we run the original form and swap $(m,n) \rightarrow (-n,m)$.

We mention a curious property of the points this algorithm returns. If $(m_{i-1},n_{i-1})$ and $(m_i,n_i)$ are any two successive points, the triangle that they form with the origin (0,0) will always have area 1/2. Hence, the orbit obeys one of Kepler's laws of planetary motion, sweeping out equal area in equal time.

For a hardware implementation, it is convenient that the routine uses only nonnegative integers, none larger than $L$.

## 25.2   NONCENTERED GRID

As we have presented the algorithm so far, the origin of the rectangular coordinate system is at (0,0). Another possibility is that the origin is midway between four grid points, as shown in Figure 25-4(a).

The grid points are located at $(m+1/2,n+1/2)$. But we can still visit them in their natural polar coordinate order.

First, imagine that we have overlaid the grid with a finer grid, as shown in Figure 25-4(b), and doubled the scale. The grid points of interest are now those at $(m', n')=(2m+1,2n+1)$ (e.g., both $m'$ and $n'$ odd), but our algorithm will generate the all the points $(m,n)$. If $m,n$ are both odd, report out $(m',n')=(m,n)$, which stands for itself and for all its odd multiples lying on the same radial line. But if either $m$ or $n$ is even, the point $(m,n)$ is ignored and we go on to the point the algorithm generates next. (The algorithm will never report out $m$ and $n$ both even because the pairs it finds are always mutually prime.)

## 25.3   TRIANGULAR GRID

Another possibility is that the grid, including the point at the origin, is not square but triangular. With a triangular grid, every other row of grid points is displaced by



(a)                                                    (b)
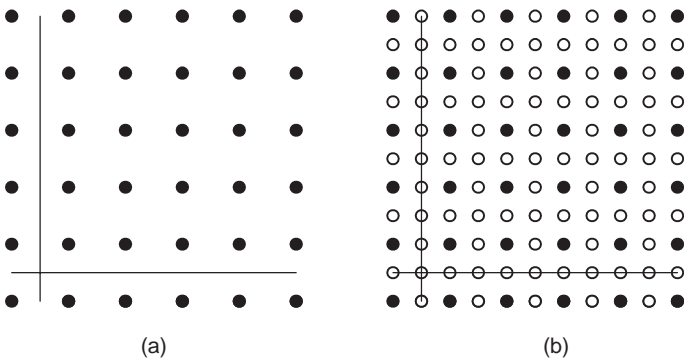
**Figure 25-4**   Origin centered among four grid points.
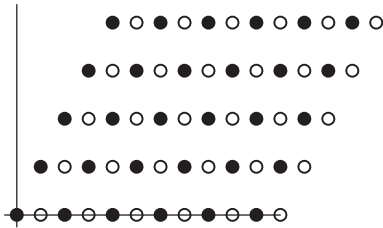
**Figure 25-5**   A triangular grid with interpolated grid points.

half a unit. Once again, we can interpolate other grid points so that they fill out a rectangular array, as shown in Figure 25-5.

The points of interest $(m',n')$ now have either both coordinates odd or both coordinates even. So when we run the algorithm and get a candidate $(m,n)$ there are three cases:

> $m$ **odd**, $n$ **odd**: Report $(m',n')=(m,n)$. The points on the same radial line are $(km,kn)$ for all integers $k$.

> Otherwise: Report $(m',n')=(2m,2n)$. The points on the same radial line are $(2km,2kn)$ for all integers $k$.

## 25.4   CONCLUSIONS

We introduced the situation where we need the indexes *(m,n)* within an $L$-by-$M$ rectangle, of a rectangular coordinate system, to be sorted in the order of increasing angle $\theta=\tan^{-1}(n/m)$. If both $L$ and $M$ are small, we would probably accomplish this with a lookup table, but a surprisingly simple and elegant algorithm can be used instead and can be used for much larger $L$ and $M$. It uses only integer arithmetic and a few simple tests. The algorithm can quickly identify the grid points between any two angles, or the polar coordinate measurements we need to compute rectangular coordinate measurements by interpolation.

# Chapter 26

# The Swiss Army Knife
# of Digital Networks

**Richard Lyons**

*Besser Associates*

**Amy Bell**

*Virginia Tech*

**T**his chapter describes a general discrete-signal network that appears, in various forms, inside many signal processing applications. Practicing DSP engineers are well advised to become acquainted with this network. Figure 26-1 shows how the network's structure has the distinct look of a digital filter—a comb filter followed by a second-order recursive network. However, we do not call this general network a filter because its capabilities extend far beyond simple filtering. Through a series of examples we illustrate the network's ability to be reconfigured to perform a surprisingly large number of useful functions based on the values of its seven control parameters (coefficients).

The general network in Figure 26-1 has a transfer function of

$$H(z) = (1 - c_1 z^{-N}) \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1/a_0 - a_1 z^{-1} - a_2 z^{-2}}. \tag{26–1}$$

From here out we'll use DSP filter lingo and call the second-order recursive subnetwork in Figure 26-1 a *biquad* because its transfer function is the ratio of two quadratic polynomials. The tables in this chapter list various signal processing functions performed by the network based on the $a_n$, $b_n$, and $c_1$ coefficients. Variable $N$ is the order (the number of unit-delay elements) of the comb filter. Included in the tables are depictions of the network's impulse
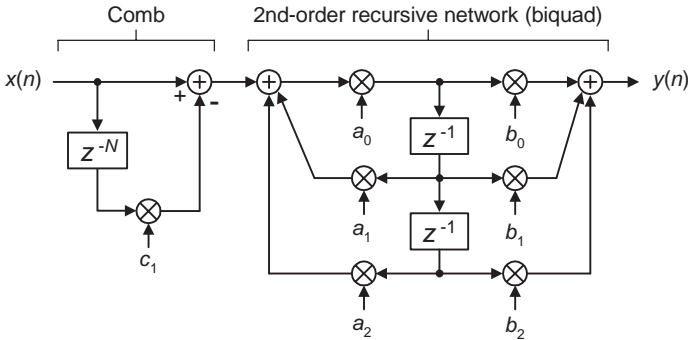
**Figure 26-1** General discrete–signal processing network.

response, $z$-plane pole/zero locations, as well as frequency-domain magnitude and phase responses. The frequency axis in those tables is normalized such that a value of 0.5 represents a frequency of $f_s/2$ where $f_s$ is the sample rate in hertz.

## 26.1 GENERAL FUNCTIONS

### Moving Averager (N-Point)

Referring to the first entry in Table 26-1, this network configuration is a computationally efficient method for computing the $N$-point moving average of $x(n)$. Also called a *recursive moving averager* or *boxcar averager*, this structure is equivalent to an $N$-tap direct convolution FIR filter with all the coefficients equal to $1/N$. Thus the moving averager's impulse response samples are $1/N$ in amplitude. This moving averager is efficient because it performs only one add and one subtract per output sample regardless of the value of $N$. (An $N$-tap direct convolution FIR filter must perform $N-1$ additions per output sample.) The moving averager's transfer function is

$$H_{ma}(z) = (1/N)\frac{1-z^{-N}}{1-z^{-1}}. \qquad (26\text{–}2)$$

$H_{ma}(z)$'s numerator produces $N$ zeros equally spaced around the $z$-plane's unit circle located at $z(k) = e^{j2\pi k/N}$, where integer $k$ is $0 \le k < N$. $H_{ma}(z)$'s denominator places a single pole (infinite gain) at $z=1$ on the unit circle, canceling the zero at that location.

### Differentiator (First-Difference)

This is a discrete version of a first-order differentiator. (Purists call this network a *digital differencer*.) An ideal differentiator has a frequency magnitude response that's

**Table 26-1** General Functions



Moving Averager (*N*-point):
$a_0 = 1, a_1 = 1, a_2 = 0, b_0 = 1/N, b_1 = 0, b_2 = 0, c_1 = 1, N = 8$

Differentiator (First-difference):
$a_0 = 1, a_1 = 0, a_2 = 0, b_0 = 1, b_1 = -1, b_2 = 0, c_1 = 0$

Differentiator (Central-difference):
$a_0 = 1, a_1 = 0, a_2 = 0, b_0 = 0.5, b_1 = 0, b_2 = -0.5, c_1 = 0$
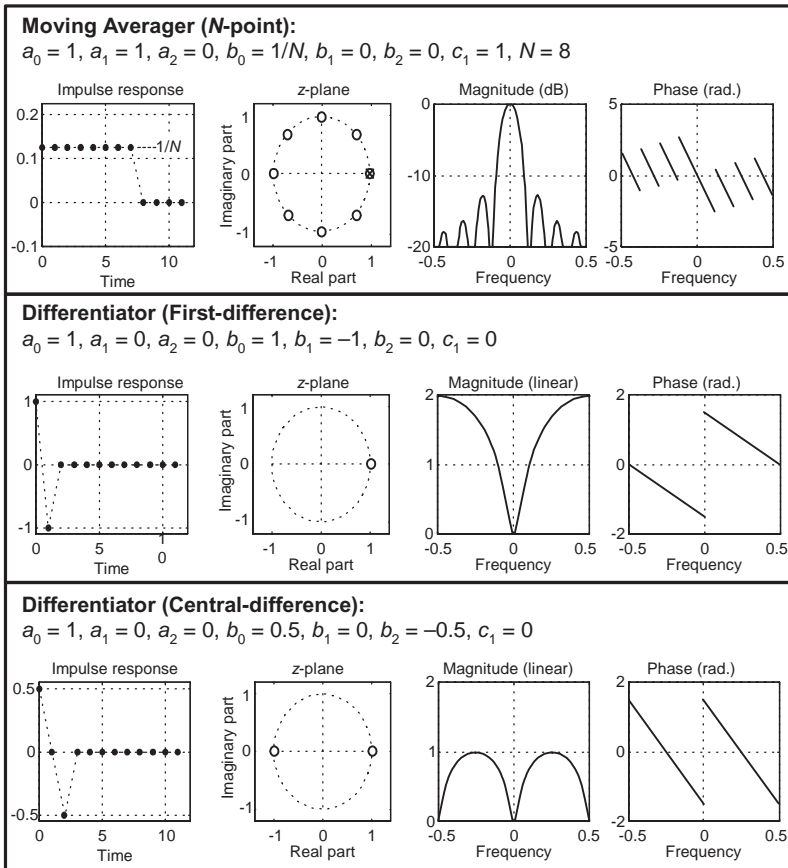
a linear function of frequency, and this network approaches that ideal only at low frequencies relative to $f_s$. This configuration of the network amplifies high-frequency spectral components, and this may be detrimental because noise is oftentimes high frequency in real-world signals. The network has a group delay of 0.5 samples.

## Differentiator (Central-Difference)

This second-order differentiator attenuates high-frequency (noise) spectral components, and has a group delay of one sample, which is convenient when the $y(n)$ output must be synchronized in time to some other signal sequence. Its frequency range of linear operation (starting at zero Hz), however, is smaller than the above first-difference differentiator.

**Table 26-1**    *Continued*

**Integrator (Running sum):**
$a_0 = 1$, $a_1 = 1$, $a_2 = 0$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 0$



**Leaky Integrator:**
$a_0 = 1$, $a_1 = 1-\alpha$, $a_2 = 0$, $b_0 = \alpha$, $b_1 = 0$, $b_2 = 0$, $c_1 = 0$, $\alpha = 0.1$



**1st-order Delay Network:**
$a_0 = 1$, $a_1 = -R$, $a_2 = 0$, $b_0 = R$, $b_1 = 1$, $b_2 = 0$, $c_1 = 0$



**2nd-order Delay Network:**
$a_0 = 1$, $a_1 = -R_1$, $a_2 = -R_2$, $b_0 = R_2$, $b_1 = R_1$, $b_2 = 1$, $c_1 = 0$
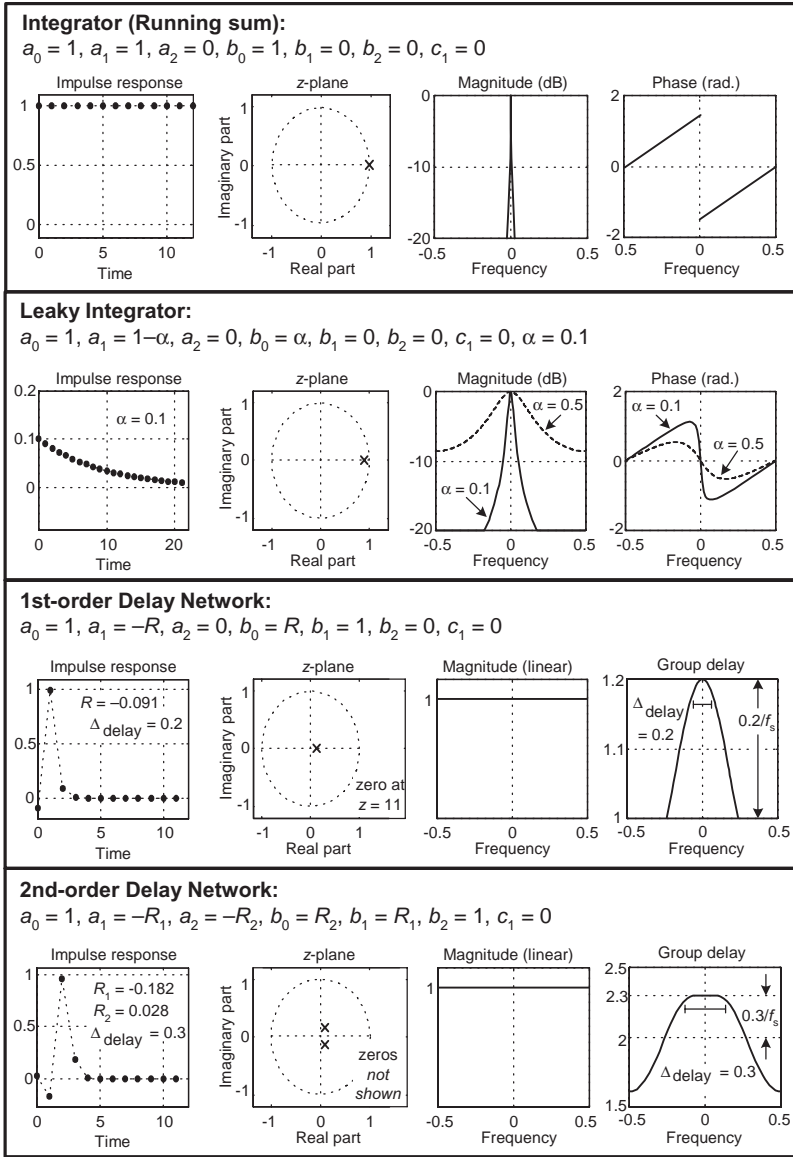


# Integrator (Running Sum)

This structure performs the *running summation* of the $x(n)$ inputs samples, making it a discrete-time approximation of continuous-time integration. This discrete integrator has a pole at $z=1$, corresponding to a cyclic frequency of zero Hz, and this

has an important consequence when we implement integrators. The pole forces us to ensure that the numerical format of our integrator hardware can accommodate summation results when the $x(n)$ input sequence has a nonzero average value (a constant amplitude bias). Stated in different words, the widths of our binary data registers must large enough to guarantee that any nonzero-amplitude bias on $x(n)$ will not cause numerical overflow and corrupt the data within an integrator's accumulator register.

## Leaky Integrator

This network configuration, also called an *exponential averager*, is a venerable structure used in many applications for reducing random noise that contaminates low-frequency, or constant-amplitude, signals of interest. It is a first-order infinite impulse response (IIR) filter where, for stable lowpass operation, the constant $\alpha$ lies in the range $0 < \alpha < 1$.

This nonlinear-phase lowpass filter has a single pole at $z = 1 - \alpha$ on the $z$-plane, and a transfer function of

$$H_{\text{li}}(z) = \frac{\alpha}{1 - (1 - \alpha)z^{-1}}. \tag{26–3}$$

Small values for $\alpha$ yield narrow passbands at the expense of increased filter response time. Table 26-1 shows the filter's behavior for $\alpha = 0.1$ as solid curves. For comparison, the frequency domain performance for $\alpha = 0.5$ is indicated by the dashed curves.

The DC (zero Hz) gain of the leaky integrator is unity. If a DC gain of $1/\alpha$ can be tolerated, then we can set $b_0$ to unity to eliminate one of the integrator's multipliers.

## First-Order Delay Network

A subclass of a first-order IIR filter, the coefficients in Table 26-1 yield an *all-pass* network having a relatively constant group delay at low frequencies. The network's delay is $D_{\text{total}} = 1 + \Delta_{\text{delay}}$ samples where $\Delta_{\text{delay}}$, typically in the range of $-0.5$ to $0.5$, is a fraction of the $1/f_s$ sample period. For example, when $\Delta_{\text{delay}}$ is $0.2$, the network delay (at low frequencies) is $1.2$ samples. The real-valued $R$ coefficient is

$$R = \frac{-\Delta_{\text{delay}}}{\Delta_{\text{delay}} + 2} \tag{26–4}$$

producing a $z$-plane transfer function of

$$H_{1,\text{del}}(z) = \frac{R + z^{-1}}{1 + Rz^{-1}} \tag{26–5}$$

with a pole at $z = -R$ and a zero at $z = -1/R$.

Performance for $\Delta_{delay}=0.2$ ($R=-0.091$) is shown in Table 26-1, where we see the magnitude response being constant. The band, centered at DC, over which the group delay varies no more than $|\Delta_{delay}|/10$ from the specified $D_{total}$ value, the bar in the group delay plot, ranges roughly from $0.1f_s$ to $0.2f_s$ for first-order networks. So if your signal is oversampled, making it low in frequency relative to $f_s$, this first-order all-pass delay network may be of some use. If you propose its use in a new design, you can impress your colleagues by saying this network is based on the *Thiran approximation* [1].

## Second-Order Delay Network

A subclass of a second-order IIR filter, the coefficients in Table 26-1 yield an all-pass network having a relatively constant group at low frequencies (over a wider frequency range, by the way, than the first-order delay network). This network's delay is $D_{total}=2+\Delta_{delay}$ samples where $\Delta_{delay}$ is typically in the range of $-0.5$ to $0.5$. For example, when $\Delta_{delay}$ is 0.3, the network delay (at low frequencies) is 2.3 samples. The real-valued coefficients are

$$R_1 = \frac{-2\Delta_{delay}}{\Delta_{delay}+3} \text{ and } R_2 = \frac{(\Delta_{delay})(\Delta_{delay}+1)}{(\Delta_{delay}+3)(\Delta_{delay}+4)}. \tag{26-6}$$

The band, centered at DC, over which the group delay varies no more than $|\Delta_{delay}|/10$ from the specified $D_{total}$ value, the bar in the group delay plot, ranges roughly from $0.26f_s$ to $0.38f_s$ for this second-order network. Performance for $\Delta_{delay}=0.3$ ($R_1=-0.182$ and $R_2=0.028$) is shown in Table 26-1 where we see the magnitude response being constant.

The *flat* group delay band is wider for negative $\Delta_{delay}$ than when $\Delta_{delay}$ is positive. This means that if you desire, for example, a group delay of $D_{total}=2.5$ samples, it's better to use an external unit delay and set $\Delta_{delay}$ to $-0.5$ rather than letting $\Delta_{delay}$ be 0.5. To ensure stability, $\Delta_{delay}$ must be greater than $-1$. Reference [1] provides methods for designing higher-order all-pass delay networks.

## 26.2  ANALYSIS AND SYNTHESIS FUNCTIONS

## Goertzel Network

Referring to the first entry in Table 26-2, this traditional Goertzel network is used for single-tone detection because it computes a single-bin $N$-point discrete Fourier transform (DFT) centered at an angle of $\theta=2\pi k/N$ radians on the unit circle, corresponding to a cyclical frequency of $kf_s/N$ Hz. Frequency variable $k$, in the range $0 \le k < N$, need not be an integer. The behavior of the network is shown by the solid curves in Table 26-2. However, the frequency magnitude response of the Goertzel algorithm, for $N=8$ and $k=1$, is shown as the dashed curve.

**Table 26-2**    Analysis and Synthesis Functions

**Goertzel Network:**

$a_0 = 1$, $a_1 = 2\cos(\theta)$, $a_2 = -1$, $b_0 = 1$, $b_1 = -e^{-j\theta}$, $b_2 = 0$, $c_1 = 0$, $\theta = 2\pi k/N$ (rad.)

| Impulse response | z-plane | Magnitude (dB) | Phase (rad.) |
|---|---|---|---|
| | | $N = 8$, $k = 1$ | |

**Sliding DFT Network:**

$a_0 = re^{j\theta}$, $a_1 = 1$, $a_2 = 0$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = r^N$, $\theta = 2\pi k/N$ (rad.)

| Impulse response | z-plane | Magnitude (dB) | Phase (rad.) |
|---|---|---|---|
| | | $N = 22$, $k = 2$, $\theta = 4\pi/22$, $r = 0.999$ | |

**Real Oscillator:**

$a_0 = 1$, $a_1 = 2\cos(\theta)$, $a_2 = -1$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 0$

| Impulse response | z-plane | Magnitude (dB) | Phase (rad.) |
|---|---|---|---|
| | | $\theta = \pi/4$ | |

**Quadrature Oscillator:**

$a_0 = G(n)$, $a_1 = e^{j\theta}$, $a_2 = 0$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 0$, $\theta = 2\pi f_t/f_s$ (rad.)

| Impulse response | z-plane | Magnitude (dB) | Phase (rad.) |
|---|---|---|---|
| | | $G(n) = 1$, $\theta = \pi/4$ | |

**Audio Comb:**

$a_0 = 1$, $a_1 = 0$, $a_2 = \alpha$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 0$

| Impulse response | z-plane | Magnitude (dB) | Phase (rad.) |
|---|---|---|---|
| $\alpha = 0.2$ | | $\alpha = 0.2$ | $\alpha = 0.2$ |

After $N+1$ input samples are applied, $y(n)$ is a single-bin DFT result. The DFT computational workload is $N+2$ real multiplies and $2N+1$ real adds. The network is typically stable because $N$ is kept fairly low (in the hundreds) in practice before the network is reinitialized [2], [3].

## Sliding DFT Network

This structure computes a single-bin $N$-point DFT centered at an angle of $\theta=2\pi k/N$ radians on the unit circle, corresponding to a cyclical frequency of $kf_s/N$ Hz. $N$ is the DFT size and integer $k$ is $0\le k<N$. Real damping factor $r$ is kept less than, but as close to, unity as possible to maintain network stability. After $N$ input samples have been applied, this network will compute a new follow-on DFT result based on each new $x(n)$ sample (thus the term *sliding*) at a computational workload of only four real multiplies and four real adds per input sample [2], [3]. Setting coefficient $c_1=-r^N$ allows the analysis band to be centered at an angle of $\theta=2\pi(k+1/2)/N$ radians, corresponding to a cyclical frequency of $(k+1/2)f_s/N$ Hz.

## Real Oscillator

There are many possible digital oscillator structures, but this network generates a real-valued sinusoidal $y(n)$ sequence whose amplitude is not a function of the output frequency. The argument for coefficient $a_1$ in Table 26-2 is $\theta=2\pi f_t/f_s$ radians, where $f_t$ is the oscillator's tuned frequency in hertz. To start the oscillator we set the $y(n-1)$ sample driving the $a_1$ multiplier equal to 1 and compute new output samples as the time index $n$ advances. For fixed-point implementations, filter coefficients may need to be scaled so that all intermediate results are in the proper numerical range [4].

## Quadrature Oscillator

Called the *coupled quadrature oscillator*, this structure provides $y(n)=\cos(n\theta)+j\sin(n\theta)$ outputs for a complex exponential sequence whose tuned frequency is $f_t$ Hz. The exponent for $a_1$ in Table 26-2 is $\theta=2\pi f_t/f_s$ radians. To start the oscillator, we set the complex $y(n-1)$ sample, driving the $a_1$ multiplier, equal to $1+j0$ and begin computing output samples as the time index $n$ advances. To ensure oscillator output stability in fixed-point arithmetic implementations, instantaneous gain correction $G(n)$ must be computed for each output sample. The $G(n)$ sample values will be very close to unity [5], [6].

## Audio Comb

This structure is a second-order (the simplest) version of an infinite impulse response (IIR) comb filter used by audio folk to synthesize the sound of a plucked-string instrument. The input to the filter is random noise samples. The filter has frequency-

response peaks at DC and $\pm f_s/2$, with dips in the response located at $\pm f_s/4$. The filter's transfer function is

$$H_{ac}(z) = \frac{1}{1 - \alpha z^{-2}} \qquad (26\text{--}7)$$

resulting in two poles located at $z = \pm\text{sqrt}(\alpha)$ on the $z$-plane. To maintain stability the real-valued $\alpha$ must be less than unity, and the closer $\alpha$ is to unity the more narrow the frequency-response peaks.

For a more realistic-sounding synthesis, we can set $a_1 = \alpha$ and the top delay element of the biquad in Figure 26-1 may have its delay increased to, say, eight instead of one, yielding more frequency-response peaks between 0 and $f_s/2$ Hz. In this music application, the filter's input is Gaussian white noise samples. Other plucked-string instrument synthesis networks have been used with success [7], [8].

## 26.3   FILTER FUNCTIONS

### Comb Filter

Referring to the first entry in Table 26-3, this finite impulse response (FIR) comb filter is a key component on many filtering applications, as we shall see. Its transfer function, $H_{comb}(z) = 1 - z^{-N}$, results in $N$ zeros equally spaced around the $z$-plane's unit circle located at $z(k) = e^{j2\pi k/N}$, where integer $k$ is $0 \leq k < N$. Those $z(k)$ values are the $N$ roots of unity when we set $H_{comb}(z)$ equal to zero yielding $z(k)^N = (e^{j2\pi k/N})^N = 1$. The $N$ zeros on the unit circle result in frequency response nulls (infinite attenuation) located at cyclic frequencies of $m f_s/N$ where integer $m$ is $0 \leq m \leq N/2$. The peak gain of this linear-phase filter is 2.

If we set coefficient $c_1$ to $-1$ in the comb filter, making its transfer function $H_{alt,comb}(z) = 1 + z^{-N}$, we obtain an alternate linear-phase comb filter having zeros rotated counterclockwise around the unit circle by an angle of $\pi/N$ radians positioning the zeros at angles of $2\pi(k+1/2)/N$ radians on the $z$-plane's unit circle. The rotated zeros result in frequency response nulls located at cyclic frequencies of $(m+1/2)f_s/N$, where integer $m$ is $0 \leq m \leq (N/2)-1$. With this filter a frequency magnitude peak is located at 0 Hz (DC).

### Bandpass Filter at $f_s/4$

This network is a bandpass filter centered at $f_s/4$ having a $\sin(x)/x$-like frequency response and linear-phase over the passband. It has poles at $z = \pm j$, so for pole/zero cancellation the comb filter's delay ($N$) must be an integer multiple of four. This guaranteed-stable, multiplierless, bandpass filter's transfer function is

**Table 26-3** Filter Functions

**Comb Filter:**
$a_0 = 1$, $a_1 = 0$, $a_2 = 0$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 1$



**Bandpass Filter at $f_s/4$:**
$a_0 = 1$, $a_1 = 0$, $a_2 = -1$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 1$



**1st-order IIR Filter:**
$a_0 = 1$, $a_1 = R_p e^{j\theta_p}$, $a_2 = 0$, $b_0 = 1$, $b_1 = R_z e^{j\theta_z}$, $b_2 = 0$, $c_1 = 0$



**1st-order Equalizer:**
$a_0 = 1$, $a_1 = R$, $a_2 = 0$, $b_0 = -R^*$, $b_1 = 1$, $b_2 = 0$, $c_1 = 0$
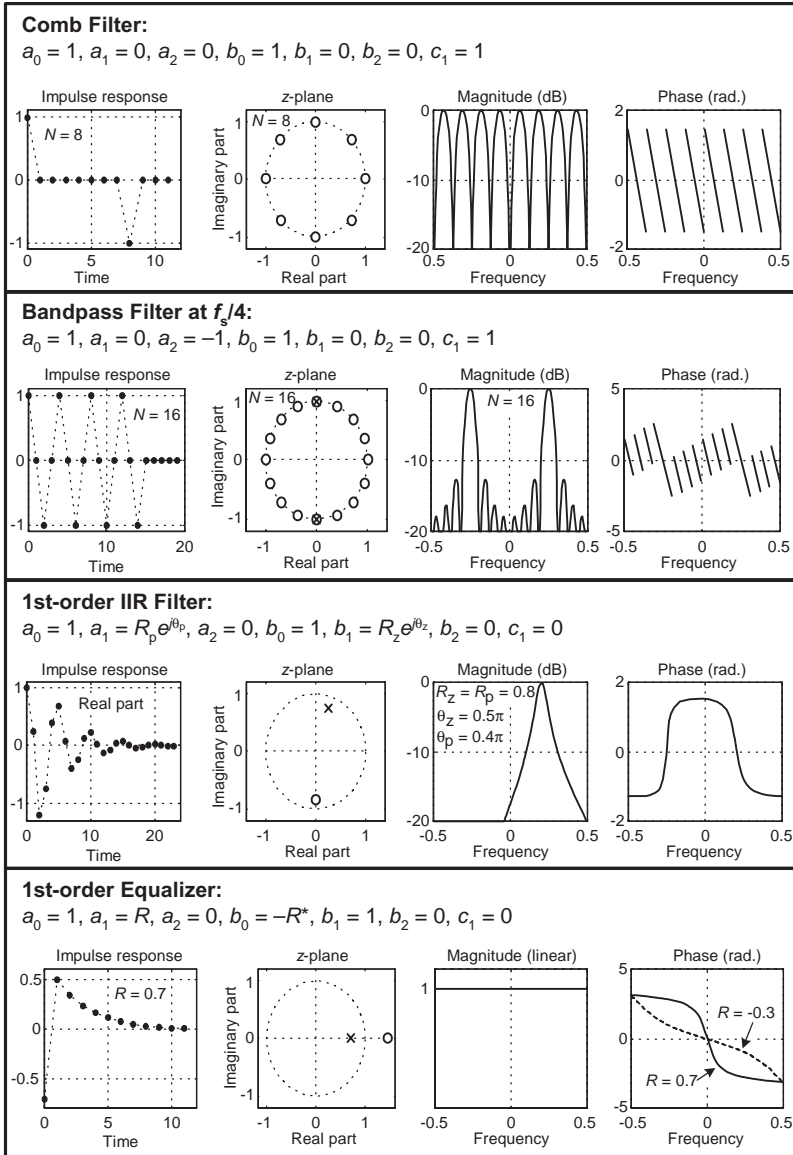
**Table 26-3** *Continued*
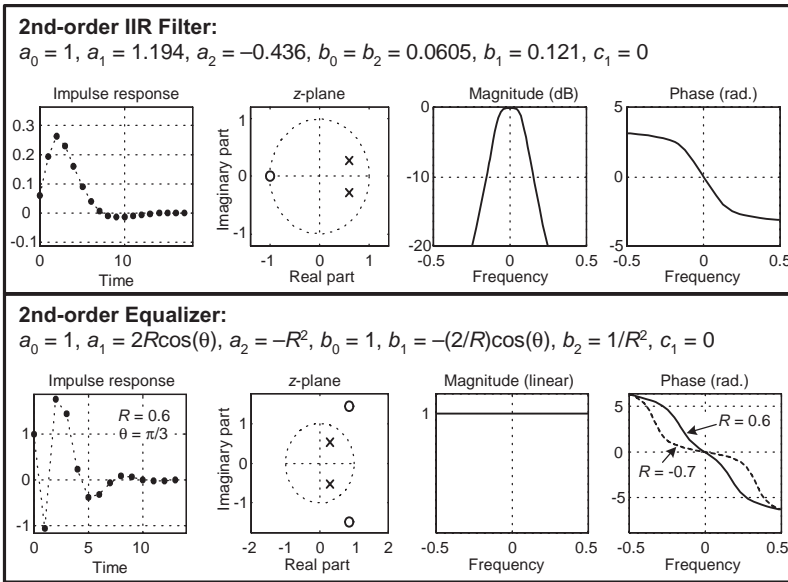


$$H_{bp}(z) = \frac{1 - z^{-N}}{1 + z^{-2}}. \qquad (26\text{–}8)$$

## First-Order IIR Filter

This is the Direct Form II version of a simple first-order IIR filter having a single pole located at a radius of $R_p$ from the $z$-plane's origin at an angle of $\theta_p$ radians, and a single zero at a radius of $R_z$ at an angle of $\pi + \theta_z$. For real-valued coefficients ($\theta_p = \theta_z = 0$) the filter can exhibit only either a lowpass or a highpass frequency response; no bandpass or bandstop filters are possible. The filter's transfer function is

$$H_{1,iir}(z) = \frac{1 + R_z e^{j\theta_z} z^{-1}}{1 - R_p e^{j\theta_p} z^{-1}}. \qquad (26\text{–}9)$$

The shapes of the filter's frequency magnitude responses are nothing to write home about; its transition regions are so wide that they don't actually have distinct passbands and stopbands. Of course to ensure stability, $R_p$ must be between zero and 1 to keep the pole inside the $z$-plane's unit circle, and the closer $R_p$ is to unity the more narrowband is the filter.

## First-Order Equalizer

This structure has a frequency magnitude response that's constant across the entire frequency band (an all-pass filter). It has a pole at $z = R$ on the $z$-plane, and a zero

located at $1/R^*$, where $*$ means conjugate. The value of $R$, which can be real or complex but whose magnitude must be less than unity to ensure stability, controls the nonlinear-phase response. The equalizer has a transfer function of

$$H_{1,eq}(z) = \frac{-R^* + z^{-1}}{1 - Rz^{-1}}. \tag{26-10}$$

These networks can be used as *phase equalizers* by cascading them after a filter, or network, whose nonlinear phase response requires crude linearization. The goal is to make the cascaded filters' combined phase as linear as possible. Table 26-3 shows the filter's behavior for $R=0.7$ as solid curves. For comparison, the phase response for $R=-0.3$ is indicated by the dashed curve. These first-order all-pass filters can also be used for interpolation and audio reverberation for low-frequency signals.

## Second-Order IIR Filter

This is the Direct Form II version of a second-order IIR filter, the *workhorse* of IIR filter implementations. Conjugate pole and zero pairs may be positioned anywhere on the $z$-plane to control the filter's frequency response. Because high-order IIR filters are so susceptible to coefficient quantization and potential data overflow problems, practitioners typically implement their IIR filters by cascading multiple copies of this second-order IIR structure to ensure filter stability and avoid *limit cycles*. The filters have a transfer function of (26–1) with the $c_1=0$. Lowpass, highpass, bandpass, and bandstop filters are possible. No single example shows all the possibilities of this structure, so Table 26-3 merely gives a simple lowpass filter example.

If an IIR filter design requires high performance, called "high Q," it turns out the Direct Form I version of a second-order IIR filter is less susceptible to coefficient quantization and overflow errors than the Direct Form II structure given here.

## Second-Order Equalizer

This structure has a frequency magnitude response that's constant across the entire frequency band, making it an all-pass filter. It has two conjugate poles located at a radius of $R$ from the $z$-plane's origin at angles of $\pm\theta$ radians, and two conjugate zeros at a reciprocal radius of $1/R$ at angles of $\pm\theta$. The positioning of the poles and zeros, using real-valued $R$, controls the nonlinear-phase response. Table 26-3 shows the equalizer's behavior for $R=0.6$ and $\theta=\pi/3$ as solid curves. For comparison, the phase response for $R=-0.7$ and $\theta=\pi/3$ is indicated by the dashed curve.

These networks are primarily used for phase equalization by cascading them after a filter, or network, whose nonlinear phase response requires linearization. However, it may take multiple cascaded biquad networks to achieve acceptable equalization.

## 26.4   ADDITIONAL FILTER FUNCTIONS

### CIC Interpolation Filter

Referring to the first entry in Table 26-4, this network is a single-stage cascaded integrator-comb (CIC) interpolation filter used for time-domain interpolation. If a time-domain signal sequence is upsampled by $N$ (by inserting $N-1$ zero-valued samples in between each original sample) and applied to this lowpass filter, the filter's output is an interpolated-by-$N$ version of the original signal. This lowpass filter's transfer function is

$$H_{cic}(z) = \frac{1 - z^{-K}}{1 - z^{-1}}. \tag{26–11}$$

To improve the attenuation of spectral images, we can cascade $M$ copies of the comb filter followed by $M$ cascaded biquad sections. Such cascaded filters will also have narrower passband widths at zero Hz.

In practice, the upsampling operation (zero stuffing) is performed after the comb filter and before the biquad network. This has the sweet advantage that the comb filter's delay length becomes $N=1$, reducing the necessary comb delay storage requirement to one. CIC filters are often used as the first stage of multistage lowpass filtering in hardware interpolation-by-$N$ applications because CIC filters require no multipliers [6]. When used as a lowpass filter in sample rate change applications, the filter is typically implemented with two's complement arithmetic.

### Complex Frequency Sampling Filter (FSF)

This structure is a single section of a complex *frequency sampling filter* having a $\sin(x)/x$-like frequency magnitude response centered at an angle of $\theta_k = 2\pi k/N$ radians on the unit circle, corresponding to a cyclical frequency of $kf_s/N$ Hz. $N$ and $k$ are integers where $k$ is $0 \leq k < N$. The larger is $N$, the more narrow is the filter's mainlobe width [6].

If multiple biquads are implemented in parallel (all driven by the single comb filter), with adjacent center frequencies, complex, almost linear-phase bandpass filters can be built. Table 26-4 shows the behavior of an $N=16$, three-biquad, complex bandpass filter each centered at $k=2$, 3, and 4 respectively.

### Real Frequency Sampling Filter, Type I

This structure is a single section of a real-coefficient *frequency sampling filter* having a $\sin(x)/x$-like frequency magnitude responses centered at both $\pm\theta_k = \pm2\pi k/N$ radians, where $N$ is an integer. The larger is $N$, the more narrow is the filter's mainlobe width. Integer $k$ is $0 \leq k < N$.

If multiple biquads are implemented in parallel (all driven by the single comb filter), with adjacent center frequencies, almost linear-phase lowpass filters can be

**Table 26-4**    Additional Filter Functions

**CIC Interpolation Filter:**

$a_0 = 1$, $a_1 = 1$, $a_2 = 0$, $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $c_1 = 1$



**Complex Frequency Sampling Filter:**

$a_0 = 1$, $a_1 = e^{j\theta_k}$, $a_2 = 0$, $b_0 = (-1)^k$, $b_1 = 0$, $b_2 = 0$, $c_1 = 1$, $\theta_k = 2\pi k/N$ (rad.)



**Real Frequency Sampling Filter, Type I:**

$a_0 = 1$, $a_1 = 2\cos(\theta_k)$, $a_2 = -1$, $b_0 = |H_k|\cos(\phi_k)$, $b_1 = -|H_k|\cos(\phi_k - \theta_k)$, $b_2 = 0$, $c_1 = 1$, $\theta_k = 2\pi k/N$ (rad.)



**Real Frequency Sampling Filter, Type IV:**

$a_0 = 1$, $a_1 = 2\cos(\theta_k)$, $a_2 = -1$, $b_0 = (-1)^k M_k$, $b_1 = 0$, $b_2 = (-1)^k(-M_k)$, $c_1 = 1$, $\theta_k = 2\pi k/N$ (rad.)
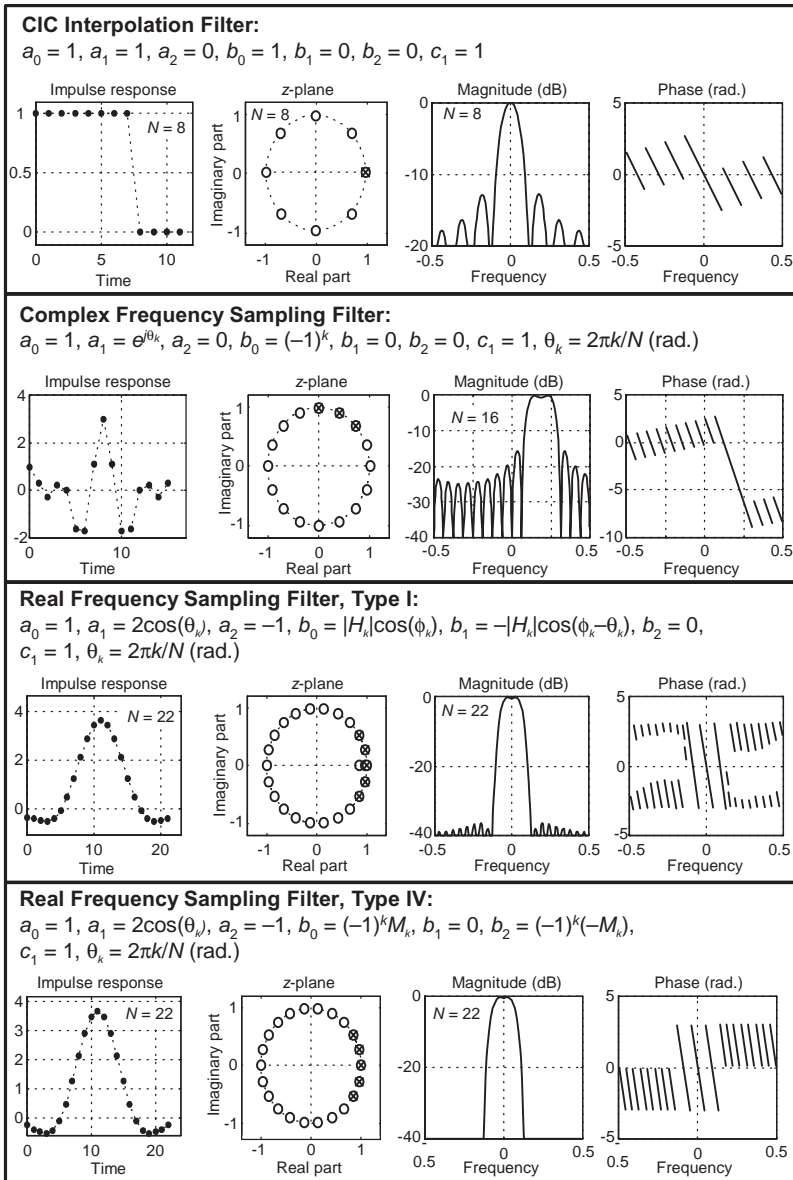


built. In this case, complex gain factors $H_k$ are the desired peak frequency response of the $k$th biquad. Parameter $\phi_k$ is the desired relative phase shift, in radians, of $H_k$. Table 26-4 shows the behavior of an $N=22$, three-biquad, lowpass filter, each biquad centered at $k=0$, 1, and 2 respectively. In this example $|H_0|=1$, $|H_1|=2$, and $|H_2|=0.74$. These bandpass filters can have group delay fluctuations as large as $2/f_s$

**Table 26-4** *Continued*



DC Bias removal Filter:
$a_0 = 1$, $a_1 = \alpha$, $a_2 = 0$, $b_0 = 1$, $b_1 = -1$, $b_2 = 0$, $c_1 = 0$
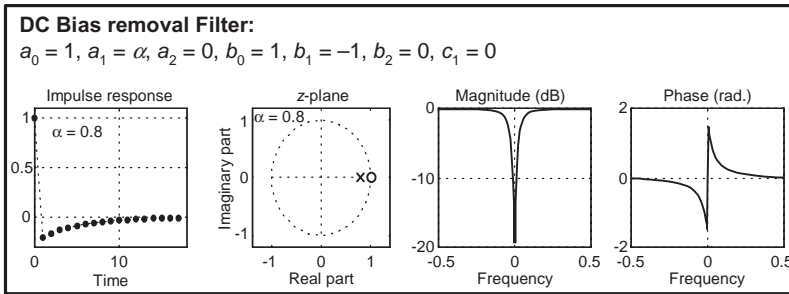
in the passband. This recursive finite impulse response (FIR) filter is the most common frequency sampling filter discussed in the traditional DSP textbooks [6], [9], [10].

## Real Frequency Sampling Filter, Type IV

This structure is similar in behavior to the type I frequency sampling filter, with important exceptions. First, in a multi-biquad lowpass filter implementation this filter yields an exactly linear phase response. Also, this filter provides deeper stopband attenuation than the type I filter.

The real-valued gain factors $M_k$ are the desired peak frequency magnitude response of the $k$th biquad. Table 26-4 shows the behavior of an $N=22$, three-biquad, lowpass filter with the biquads centered at $k=0$, 1, and 2 respectively. In this example $M_0=1$, $M_1=1$, and $M_2=0.37$. Here's why you need to know about these filters: With judicious choice of the $M_k$ gain factors, narrowband lowpass linear-phase FIR filters can be built, in some cases, whose computational workload is less than Parks-McClellan-designed FIR filters [6].

## DC Bias Removal

This network, used to remove any DC bias from the $x(n)$ input, has a transfer function having a pole located at $z=\alpha$ and a zero at $z=1$. Having a frequency-response notch (null) at 0 Hz (DC, hence the name), the sharpness of the notch is determined by $\alpha$, where for stable operation $\alpha$ lies in the range $0<\alpha<1$. The closer $\alpha$ is to unity the more narrow the notch at DC. This nonlinear-phase filter has a transfer function of

$$H_{dc}(z) = \frac{1-z^{-1}}{1-\alpha z^{-1}}.$$

(26–12)

Table 26-4 shows the filter's behavior for $\alpha=0.8$.

In those fixed-point implementations where the output $y(n)$ sequence must be truncated to avoid data overflow (i.e., $y(n)$ must have fewer bits than input $x(n)$),

feedback noise shaping can be used to reduce the quantization noise induced by truncation [6], [11].

An alternative to truncation, to avoid overflow, is to limit the gain of the filter. On one hand, we could precede the network with a positive gain element whose gain is less than unity. On the other hand, we could use $b_0 = G$ and $b_1 = -G$, where $G = (1 + \alpha)/2$, in our implementation for this purpose, yielding a reduced-gain transfer function of

$$H_{\text{alt,dc}}(z) = \frac{G - Gz^{-1}}{1 - \alpha z^{-1}}. \tag{26–13}$$

## 26.5 REFERENCES

[1] T. LAAKSO et al., "Splitting the Unit Delay," *IEEE Signal Proc.*, January 1996, pp. 30–60.

[2] E. JACOBSEN and R. LYONS, "The sliding DFT," *IEEE Signal Proc., DSP Tips & Tricks Column*, vol. 20, no. 2, March 2003, pp. 74–80.

[3] E. JACOBSEN and R. LYONS, "The Sliding DFT. An Update," *IEEE Signal Proc., DSP Tips & Tricks Column*, vol. 21, no. 1, January 2004.

[4] D. GROVER and J. DELLER, *Digital Signal Processing and the Microcontroller*. Prentice Hall, Upper Saddle River, NJ, 1999.

[5] C. TURNER, "Recursive Discrete-Time Sinusoidal Oscillators," *IEEE Signal Proc.*, vol. 20, no. 3, May 2003, pp. 103–111.

[6] R. LYONS, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 2004.

[7] J. SMITH, "Physical Audio Signal Processing." [Online: http://ccrma-www.stanford.edu/~jos/wave guide/Comb_Filters.html]

[8] TEXAS INSTRUMENTS, "How Can Comb Filters Be Used to Synthesize Musical Instruments on a TMS320 DSP?" *TMS320 DSP Designers Notebook*, no. 56, 1995.

[9] V. INGLE and J. PROAKIS, *Digital Signal Processing Using MATLAB*. Brookes/Cole Publishing, Pacific Grove, CA, 2000, pp. 202–208.

[10] J. PROAKIS and D. MANOLAKIS, *Digital Signal Processing-Principles, Algorithms, and Applications*, 3rd ed. Prentice Hall, Upper Saddle River, NJ, 1996, pp. 630–637.

[11] C. DICK and F. HARRIS, "FPGA Signal Processing Using Sigma-Delta Modulation," *IEEE Signal Proc.*, vol. 17, no. 1. January 2000.

## EDITOR COMMENTS

In what follows we expand on the Figure 26-1 network's ability to implement integration. Other integrator implementations exist beyond the simple running sum integrator described in Table 26-1. Those alternative integrators, used to approximate continuous-time integration, go by the names *trapezoidal rule, Simpson's rule*, and *Tick's rule*. We mention these integrators because they provide a more accurate approximation to continuous-time integration than does the simple running sum integrator. The coefficients and transfer function pole/zero locations on the unit circle of the alternative integrators are provided in Figure 26-2.

The frequency magnitude response of an ideal integrator is $1/\omega$ and at low frequencies the three alternative integrators in Figure 26-2 approach that ideal more

| | Trapezoidal rule | Simpson's rule | Tick's rule |
|---|---|---|---|
| $a_0$ | 1 | 1 | 1 |
| $a_1$ | 1 | 0 | 0 |
| $a_2$ | 0 | 1 | 1 |
| $b_0$ | 0.5 | 1/3 | 0.3584 |
| $b_1$ | 0.5 | 4/3 | 1.2832 |
| $b_2$ | 0 | 1/3 | 0.3584 |
| $c_1$ | 0 | 0 | 0 |

(a)

| | Trapezoidal rule | Simpson's rule | Tick's rule |
|---|---|---|---|
| poles | $z = 1$ | $z = 1$, $z = -1$ | $z = 1$, $z = -1$ |
| zeros | $z = -1$ | – | – |

Unit circle pole/zero locations

(b)
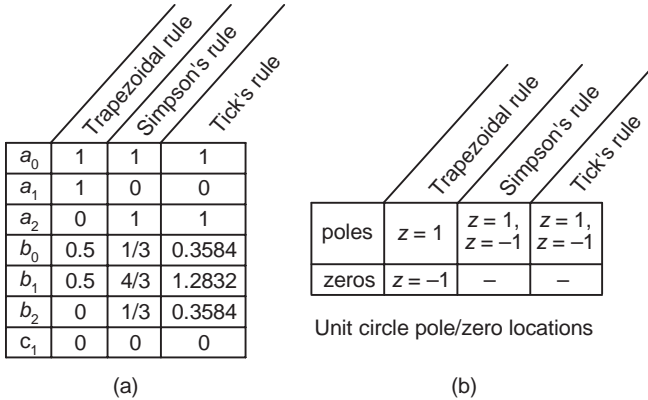
**Figure 26-2**   Alternative integrator coefficients and pole/zero locations.
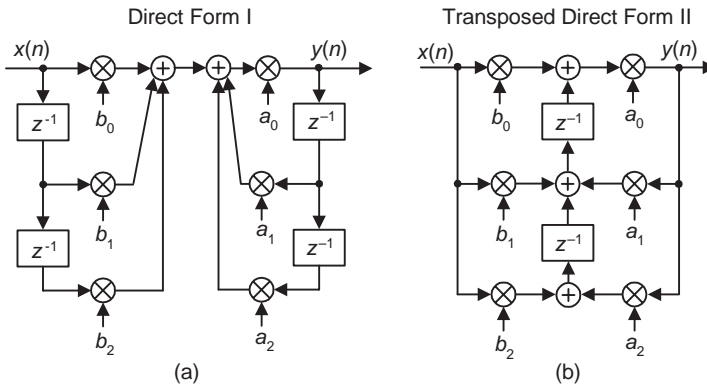


**Figure 26-3**   Alternative biquad structures: (a) Direct Form I; (b) Transposed Direct Form II.

closely than does the running sum integrator. From zero Hz to roughly $0.3f_s$ Hz, the Tick's rule integrator provides the highest integration accuracy, with the Simpson's rule integrator being a very close second. However, if the integrators' input signals have appreciable noise spectral components near $f_s/2$ Hz, the Tick's rule and Simpson's rule integrators will amplify that noise. In a high-frequency noise scenario the Tick's rule and Simpson's rule integrators should be avoided; the running sum integrator should be used instead, because it provides the most accurate integration in the presence of wideband noise and is simpler to implement than the Trapezoidal rule integrator.

As with the running sum integrator, the three Figure 26-2 integrators have $z$-domain transfer function poles (infinite gain) at $z = 1$ and, again, this is an important issue when we implement integrators. Those poles force us to ensure that the numerical format of our integrator hardware can accommodate (with no numerical overflow of) summation results when the $x(n)$ input sequence has a nonzero average value.

From an implementation standpoint, we remind the reader that the block diagram of the recursive biquad network in Figure 26-1 is called a Direct Form II structure. As such, the biquad network can be implemented in what are called the Direct Form I and Transposed Direct Form II structures, as shown in Figure 26-3. Those alternative biquad structures can be useful, because they are less susceptible to coefficient quantization and stability problems than is the Direct Form II structure when fixed-point arithmetic is used. In closing, we also remind the reader that the general network's comb filter may precede the biquad network, as it does in Figure 26-1, or it may follow the biquad network.

# Chapter 27

# JPEG2000—Choices and Trade-offs for Encoders

**Amy Bell**

*Virginia Tech*

**Krishnaraj Varma**

*Hughes Network Systems*

A new, and improved, image coding standard has been developed, and it's called JPEG2000. In this chapter we describe the most important parameters of this new standard and present several "tips and tricks" to help resolve design tradeoffs that JPEG2000 application developers are likely to encounter in practice.

## 27.1 JPEG2000 STANDARD

JPEG2000 is the state-of-the-art image coding standard that resulted from the joint efforts of the International Standards Organization (ISO) and the International Telecommunications Union (ITU) [1]; "JPEG" in JPEG2000 is an acronym for Joint Picture Experts Group. The new standard outperforms the older JPEG standard by approximately two decibels (dB) of peak signal-to-noise ratio (PSNR) for several images across all compression ratios [2]. Two primary reasons for JPEG2000's superior performance are the wavelet transform and *Embedded Block Coding with Optimal Truncation* (EBCOT) [3]. The standard is organized in 12 parts [4]. Part 1 specifies the core coding system while Part 2 adds some features and more sophistication to the core. Part 3 describes motion JPEG—a rudimentary form of video coding where each JPEG2000 image is a frame. Other important parts of the standard include security aspects, interactive protocols and application program interfaces for network access, and wireless transmission of JPEG2000 images.

The JPEG2000 standard is effectively a decoder standard. The parameters that were selected during the encoding process are explicitly written (or signaled) into the compressed data. The JPEG2000 standard describes how these values have to be read, interpreted, and used during the decoding process. The standard does not explicitly state on what basis these parameters are chosen on the encoding side. The choices made on these parameters are important and affect the quality and available features of a compressed image. Here we provide useful pointers and hints that will help JPEG2000 application developers in making appropriate choices for encoding parameters. Throughout the remainder of this discussion, we use the term *encoder* to mean a piece of software or hardware that performs JPEG2000 encoding. For an encoder targeting a particular application, the parameter choices are made by the application developer and hard-coded into the encoder. For an encoder targeting a more generic application, the task of choosing some parameters might be handed down to the encoder, which performs some rudimentary decision making based on the application and the image to be encoded. In either case, we refer to the act of making parameter choices as being performed by the encoder.

We limit our discussion to those parameters specified in the core processing system, Part 1 of the JPEG2000 standard. A comprehensive list of the parameters is depicted in Table 27-1; they are given in the order that they are encountered in the encoder.

The chosen values for some of these parameters are dictated by the target application. For example, most applications require the compressed image to be reconstructed at the original bit depth. The progression order and the number of quality layers are also determined by the requirements of the application. Other parameters, like the magnitude refinement coding method or the MQ code termination method,

**Table 27-1**  Parameters in Part 1 of the JPEG2000 Standard

---

 1. Reconstructed image bit depth
 2. Tile size
 3. Color space
 4. Reversible or irreversible transform
 5. Number of wavelet transform levels
 6. Precinct size
 7. Code-block size
 8. Coefficient quantization step size
 9. Perceptual weights
10. Block coding parameters:
    (a) Magnitude refinement coding method
    (b) MQ code termination method
11. Progression order
12. Number of quality layers
13. Region of interest coding method

---

minimally impact the quality of the compressed image, the size of the compressed data, or the complexity of the encoder. For each parameter, JPEG2000 provides either a recommendation or a default; this provides a good, initial choice for the parameter.

We now elaborate on six parameters for which there exists a wide range of acceptable values and those chosen values significantly impact compressed image quality and codec efficiency. The six parameters are 2, 3, 5, 7, 8, and 13 in Table 27-1. We discuss the merits of the choices for these parameters based on the following performance measures: compressed data size, compressed image quality, computation time, and memory requirements.

## 27.2   TILE SIZE

JPEG2000 allows an image to be divided into rectangular blocks of the same size called *tiles*—each tile is encoded independently. Tile size is a coding parameter that is explicitly signaled in the compressed data. By tiling an image, the distinct features in the image can be separated into different tiles; this enables a more efficient encoding process. For example, a composite image comprised of a photograph and text can be divided into tiles that separate the two; then two very different approaches (e.g., original bit depth and 5-level transform for the photograph, and bit depth of 1 and 0-level transform for the text) are used to obtain significantly better overall coding efficiency. Dividing an image into tiles also allows the use of customized perceptual weights tuned to the feature in each tile. Furthermore, tiling provides a degree of random spatial access to the reconstructed image.

Choosing the tile size for an image is an important trade-off for the encoder. Figure 27-1 shows the *Woman* image compressed at $100:1$ using two different tile sizes: (a) $64 \times 64$ and (b) $256 \times 256$. Blocking artifacts are readily observed in Figure 27-1(a) (the smaller tile size). This is a common observation at moderate to high compression ratios; however, at low compression ratios ($<32:1$) a small tile size introduces minimal blocking artifacts. Alternatively, a large tile size presents two challenges. First, if the encoder/decoder processes an entire tile at once, this may require prohibitively large memory. Second, features may not be isolated into separate tiles, and the encoding efficiency suffers.

*Recommendation:* Do not tile small images ($\leq 512 \times 512$). Tile large images with a tile size that separates the features, but at high compression ratios, use a tile size greater than or equal to $256 \times 256$ to avoid blocking artifacts.

## 27.3   COLOR SPACE

We humans view color images in the red, green, and blue (RGB) color space. The encoder can choose to convert a color image into the luminance, chrominance blue, and chrominance red (YCbCr) color space. For most color images, YCbCr concentrates image energy as well as or better than RGB. In RGB, energy is more evenly distributed across the three components; however, in YCbCr, most of the energy

<div align="center">(a)                                    (b)</div>

**Figure 27-1**    The *Woman* image compressed at $100:1$ with tile size (a) $64 \times 64$ and (b) $256 \times 256$.

resides in the luminance (Y) component. For example, the two chrominance components typically account for only 20% of the bits in the compressed JPEG2000 image. However, if the RGB image is comprised of mostly one color, then the YCbCr representation cannot improve upon the efficient energy compaction in RGB. For these color images, RGB compression quality is superior to YCbCr compression quality.

Figure 27-2 depicts a grayscale version of the *Lighthouse* image compressed at $32:1$ in the (a) RGB color space and in the (b) YCbCr color space. Compression in YCbCr shows a higher quality compressed image in (b): the roof-edge, grass and cloud texture, and other details are closer to the original, uncompressed image than in (a).

*Recommendation:* Convert the original, uncompressed RGB color image to the YCbCr color space except when the RGB image primarily consists of one color component.

## 27.4  NUMBER OF WAVELET TRANSFORM LEVELS

Each image color component is transformed into the wavelet domain using the two-dimensional discrete wavelet transform (DWT). The number of levels of the DWT is a parameter that is chosen by the encoder. By increasing the number of DWT levels, we examine the lower frequencies at increasingly finer resolution—thereby packing more energy into fewer wavelet coefficients. Thus, we expect compression performance to improve as the number of levels increases.
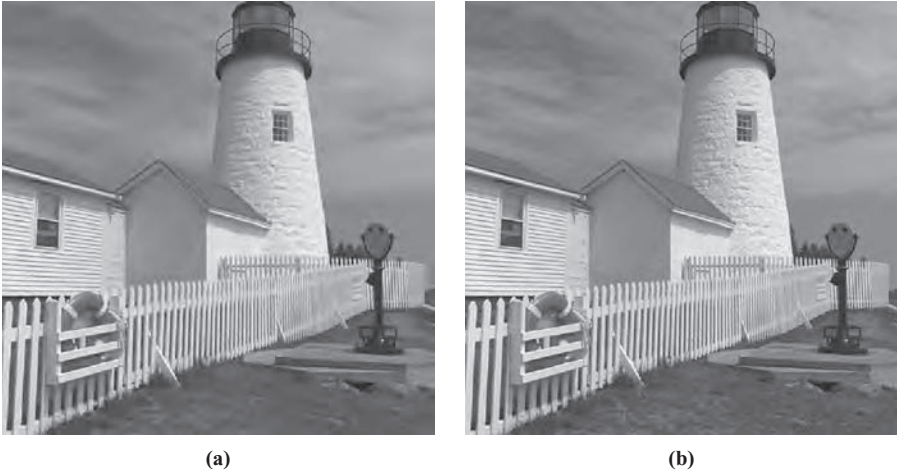
**Figure 27-2** A grayscale version of the *Lighthouse* image compressed at 32:1 in (a) RGB and (b) YCbCr.



**Figure 27-3** The *Goldhill* image compressed at 16:1 using (a) one-level DWT and (b) two-level DWT.

Figure 27-3 shows the *Goldhill* image compressed at 16:1 using (a) a one-level DWT and (b) a two-level DWT. The difference in quality between the two is minimal. At low compression ratios, quality improvement diminishes beyond two to three DWT levels. On the other hand, Figure 27-4 shows the same image compressed at 64:1 using (a) a one-level, and (b) a four-level DWT. In this case, the superior quality of the four-level DWT is clearly evident. (At high compression ratios, quality improvement diminishes beyond four to five DWT levels.) Wavelet-transform-based image compression tends to capture low-frequency coefficients before capturing higher frequencies. For a given compression ratio, assume that all of the lower frequencies in an image are completely captured after three levels.

<div align="center">(a)                                                    (b)</div>

**Figure 27-4**    The *Goldhill* image compressed at 64 : 1 using (a) one-level DWT and (b) four-level DWT.

Increasing the number of levels beyond three resolves these lower frequencies more finely, but this does not improve compression quality because these frequencies have already been captured.

*Recommendation:* Use two to three DWT levels at low compression ratios and four to five DWT levels at high compression ratios.

## 27.5  CODE-BLOCK SIZE

The DWT coefficients are separated into nonoverlapping square regions called code-blocks. Each code-block is independently coded using JPEG2000's MQ coder—a type of arithmetic encoding algorithm. Code-block size is a parameter dictated by the encoder and explicitly signaled in the compressed data.

As the code-block size increases, the memory required for the encoder/decoder increases. Therefore the size of the code-block may be limited by the available memory—particularly in hardware implementations. Moreover, if the simple scaling method is used to perform region of interest (ROI) coding (see below), then a large code-block size limits the precision of the ROI's boundary locations. Alternatively, a smaller code-block size allows a more precise definition of the ROI boundaries and, consequently, a higher-quality ROI in the compressed image. In the absence of ROI coding (and all other parameters being equal) the quality of the compressed image improves with increasing code-block size. A small code-block mitigates the efficiency of the MQ coder, which in turn decreases compressed image quality. Finally, encoding/decoding is faster for a larger code-block size since the overall overhead associated with processing all of the code-blocks is minimized.

*Recommendation:* In general, if there are memory limitations or if the scaling method of ROI coding is employed, then use a small code-block size ($<64 \times 64$). Otherwise, use the largest possible code-block size: $64 \times 64$. (JPEG2000 allows code-blocks to be of size $2n \times 2n$ where n = 2, 3, 4, 5, or 6.)

## 27.6 COEFFICIENT QUANTIZATION STEP SIZE

A quantizer divides the real number line into discrete bins; the value of an unquantized wavelet coefficient determines which bin it ends up in. The quantized wavelet coefficient value is represented by its bin index (a signed integer). JPEG2000 employs a uniform deadzone quantizer with equal-sized bins—except for the zero bin, which is twice as large. The size of the non-zero bins is equal to the quantization step size. Quantization step size is specified by the encoder; this choice represents a trade-off between compressed image quality and encoding efficiency. It is worth noting that this trade-off does not exist for the reversible wavelet transform since the unquantized wavelet coefficients are already signed integers (consequently, the default quantization step size is 1).

JPEG2000's uniform deadzone quantizer is an embedded quantizer. This means that if the signed integers are truncated such that the n least significant bits are thrown away, then this is equivalent to an increase in the quantization step size by $2^n$ [5–6]. Therefore quantization in JPEG2000 can be regarded as a two-step process. In the first step, a quantization step size is specified for each subband: The subband coefficients are represented by signed integers. In the second step, the signed integers within each code-block of each subband are optimally truncated. This is equivalent to optimally modifying the quantization step size of each code-block to achieve the desired compression ratio. Thus the resulting quantization depends only on the optimal truncation algorithm—so long as the quantization step size was chosen small enough. In summary, the quantization step-size trade-off is: Choose too large and compression quality may be jeopardized; choose too small and achieve the desired quality, but compromise codec efficiency.

Figure 27-5 depicts how compressed image quality varies as a function of quantization step size. We define peak signal-to-noise ratio (PSNR) as the ratio of signal power at full dynamic range ($255^2$ for a bit-depth of 8) to the mean squared error between original and compressed images expressed in dB. Average PSNR was computed over 23 images (from the standard image set [7]), at four compression ratios, as quantization step size changed. In JPEG2000, quantization step size can be specified for the highest resolution subband and halved for each subsequent
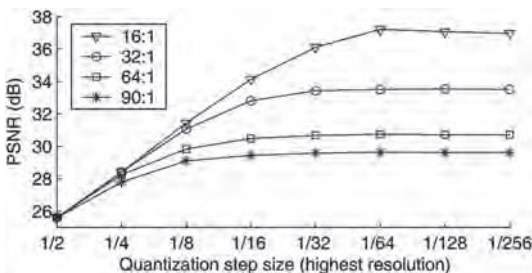


**Figure 27-5** Compressed image quality (PSNR) as a function of quantization step size at four compression ratios.

(lower-resolution) subband. Figure 27-5 shows that there is a point of diminishing returns (the "knee" in the curve) for decreasing quantization step size—each compression ratio curve flattens out at a given step size. As expected, the higher the compression ratio, the faster the curve levels off (i.e., higher compression ratios cannot take advantage of smaller step sizes). The knee of each curve represents the largest step size for which quantization due to optimal truncation is the dominant factor affecting compressed image quality. In general, if $B$ is the bit-depth of the original image, then $1/2^B$ is a conservative (i.e., to the right of the knee) quantization step size for the highest-resolution subband.

*Recommendation:* In general, for fixed-point codecs, the available bit-width determines quantization step size. The design of such a system must ensure that the bit-width of the highest-resolution subband corresponds to a quantization step size that is in the flat region of the curve for the desired compression ratio. For floating-point and software codecs, $1/2^B$ is a good, conservative value to be used for quantization step size of the highest-resolution subband.

## 27.7   REGION OF INTEREST CODING METHOD

Region of interest (ROI) coding is the JPEG2000 feature that allows a specified region of the image to be compressed at a higher quality than the remainder of the image. There are two methods for ROI coding: the scaling method and the maxshift method.

In the scaling method, the coefficients in each code-block of the ROI are multiplied by a weight that increases their value. In this way, the optimal truncation algorithm allocates more bits to these code-blocks and they are reconstructed at a higher quality. A conceptually simple method, it has two disadvantages: (1) the ROI coordinates and the scaling factor must be explicitly signaled in the compressed data; and (2) the ability to capture a ROI of a particular size is dictated by the code-block size. For example, consider an ROI of size $256 \times 256$. In a five-level DWT, this ROI corresponds to an $8 \times 8$ area in the lowest-resolution subband. Thus the code-block size must be less than or equal to $8 \times 8$. Otherwise the region will extend over the intended boundary (at the lower resolutions) and the reconstructed image will depict a progressive deterioration in quality around the ROI. Figure 27-6 depicts the impact of the code-block size on quality in ROI coding. The image was compressed at $200:1$ with five levels of decomposition and an ROI scale factor of 2048. Two different code-block sizes were employed: the $8 \times 8$ code-block defined the ROI better than the $64 \times 64$. A closeup view of Figure 27-6(a) in (c) and Figure 27-6(b) in (d) shows the smaller code-block size's higher quality. The disadvantage with the larger code-block is that some of the bits that should have been used to preserve the quality of the ROI are diverted to the surrounding area. Consequently, the $8 \times 8$ code-block results in better subjective quality and objective performance (PSNR is 34.97 dB for $8 \times 8$ and 31.52 dB for $64 \times 64$).

In the maxshift method, an arbitrarily shaped mask specifies the region of interest [8]. All coefficients—at all resolutions—that fall within the mask are shifted up
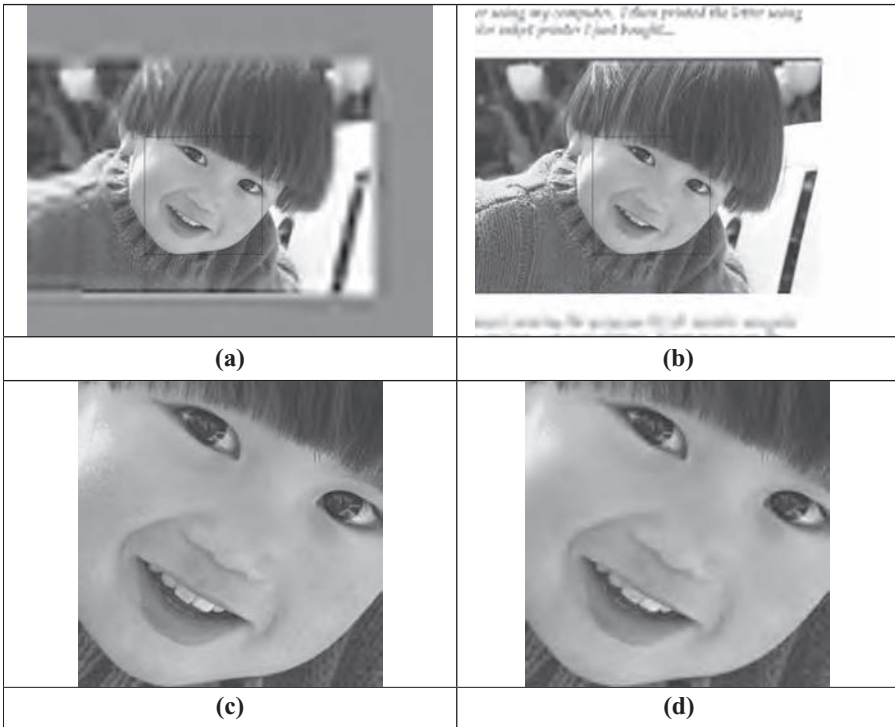
**Figure 27-6** ROI simple scaling performed on the boy's face in the standard image *CMPND2*. The ROI scale factor is 2048 for two code-block sizes (a) 8×8 and (b) 64×64.

in value by a factor called the *maxshift* factor. This shifting ensures that the least significant bit of all of the ROI coefficients is higher than the highest encoded bit-plane. As a result, the ROI is completely encoded before the remainder of the image. This method permits regions of arbitrary shape and size. Furthermore, the ROI does not extend beyond the specified area, nor does it depend on the code-block size and the number of wavelet transform levels. However, unlike the scaling method, this method reconstructs the entire region of interest before the rest of the image; therefore, there may be a significant quality difference between the ROI and non-ROI areas (particularly at high compression ratios).

*Recommendation:* As discussed in a previous section, larger code-block sizes correspond to higher compressed image quality; however, smaller code-block sizes are required for the ROI scaling method. So use the ROI scaling method if rectangular regions are of interest, but take care about how the small code-block size affects overall quality and codec efficiency. Code-block size is not an issue with the ROI maxshift method. Use the ROI maxshift method when large code-block size and/or arbitrary (nonrectangular) regions are desired. One final consideration is the compressed image quality outside the ROI. The scaling method permits a more flexible distribution of ROI and non-ROI quality; degradation in the non-ROI is more severe with the maxshift method—particularly at high compression ratios.

## 27.8 REFERENCES

[1] *ITU T.800: JPEG2000 Image Coding System Part 1,* ITU Standard, July 2002. [Online: www.itu. org.]

[2] A. SKODRAS, C. CHRISTOPOULOS, and T. EBRAHIMI, "The JPEG 2000 Still Image Compression Standard," *IEEE Signal Processing*, September 2001, pp. 36–58.

[3] D. TAUBMAN, "High Performance Scalable Image Compression with EBCOT," *IEEE Trans. Image Processing*, vol. 9, no. 7, July 2000, pp. 1158–1170.

[4] JPEG2000 standard. [Online: http://www.jpeg.org/jpeg2000/index.html.]

[5] D. S. TAUBMAN and M. W. MARCELLIN, *JPEG2000—Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Norwell, MA, 2002.

[6] M. MARCELLIN, M. LEPLEY, A BILGIN, T. FLOHR, T. CHINEN, and J. KASNER, "An Overview of Quantization in JPEG2000," *Signal Processing: Image Communications (Special Issue on JPEG2000)*, vol. 17, no. 1, 2002, pp. 73–84.

[7] *ITU T.24: Standardized Digitized Image Set*, ITU Standard, June 1998. [Online: www.itu.org.]

[8] J. ASKELÖF, M. L. CARLANDER, and C. CHRISTOPOULOS, "Region of Interest Coding in JPEG2000," *Signal Processing: Image Communications (Special Issue on JPEG2000)*, vol. 17, no. 1, 2002, pp. 105–111.

# Chapter 28

# Using Shift Register Sequences

**Charles Rader**

*Retired, formerly with MIT Lincoln Laboratory*

**T**his chapter discusses the time and frequency domain behavior of simple linear feedback shift registers to show how they can be used to generate random numbers, conveniently test high-speed digital logic, and efficiently produce useful signaling waveform sequences.

Suppose we have a clocked shift register that holds $N$ bits. On each clock pulse, the bits are shifted to the right, the rightmost bit is lost, and a new bit enters from the left. Some of the shift-register stages are tapped and we compute the new bit as some logical function of the bits appearing at those taps, so the contents of the shift register change after each clock pulse.

We illustrate this in Figure 28-1 and Table 28-1 using a four-stage register containing $a_1$, $a_2$, $a_3$, $a_4$, and the logical function is the exclusive-or, $\oplus$, of the bits in the last two flip-flops.

The logic function output, hereafter called the shift register output, will be the sequence of bits . . . 100110101111000 . . . repeated endlessly. The repetition period is 15. This generated sequence has a number of interesting and useful properties and we can generalize these properties to shift registers with any number of bits $N$.

## 28.1  COMPLETE SET OF POSSIBLE STATES

Note that an $N$ stage shift register can have exactly $2^N$ possible states and when the shift register contains any given state its future history is determined. Therefore any
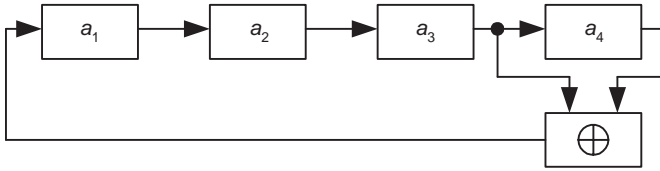
**Figure 28-1** Four-stage shift register.

**Table 28-1** Four-Stage Shift Register Sequences

| $a_3 \oplus a_4 \rightarrow$ | $a_1 \rightarrow$ | $a_2 \rightarrow$ | $a_3 \rightarrow$ | $a_4$ |
|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 1 |
| **0** | 1 | 0 | 0 | 0 |
| **0** | 0 | 1 | 0 | 0 |
| **1** | 0 | 0 | 1 | 0 |
| **1** | 1 | 0 | 0 | 1 |
| **0** | 1 | 1 | 0 | 0 |
| **1** | 0 | 1 | 1 | 0 |
| **0** | 1 | 0 | 1 | 1 |
| **1** | 0 | 1 | 0 | 1 |
| **1** | 1 | 0 | 1 | 0 |
| **1** | 1 | 1 | 0 | 1 |
| **1** | 1 | 1 | 1 | 0 |
| **0** | 1 | 1 | 1 | 1 |
| **0** | 0 | 1 | 1 | 1 |
| **0** | 0 | 0 | 1 | 1 |
| **1** | 0 | 0 | 0 | 1 |
| : | : | : | : | : |

shift register sequence must repeat its state after a period that cannot be greater than $2^N$. But we are going to limit our consideration to shift registers for which the logic function to create the input is made up of the exclusive-or of some of the shift-register taps. With that limitation, the all-zero state would repeat itself. So if the shift register is initialized with anything other than all zeros, the longest repeat period it can have is $2^N - 1$. Our four-stage example achieves that bound.

Treat the contents of the shift register as a binary integer. The sequence of values taken by the register are 1, 8, 4, 2, 9, 12, 6, 11, 5, 10, 13, 14, 15, 7, 3, . . . These are all the integers from 1 to 15, in a permuted order, each occurring once before any is repeated.

To generalize for a shift register of length $N$ we must be careful in how we choose which taps to feed into the exclusive-or function. If we choose the taps correctly, the shift register's successive states will be the integers from 1 to $2^N - 1$ periodically in a permuted order, each appearing once before any is repeated. The

sequence of bits entering the input of such a shift register is called a *maximal-length-shift-register-sequence* (*m*-sequence).

In the general case of length $N$, the logical function for the new bit that we shift into stage 1 has the form $a_{N-i} \oplus a_{N-j} \oplus a_{N-k} \oplus \ldots$, where $i, j$, and $k, \ldots$ are in the set $[0, 1, \ldots, N-1]$. Thus we could have $N = 32$ and compute the shift register input as $a_1 \oplus a_2 \oplus a_{22} \oplus a_{32}$. It is easy to see that we must include the last tap $a_N$ ($i = 0$), because otherwise the shift register sequence would be a delayed version of a sequence generated by a shorter shift register. It is less obvious how we should choose the other taps to produce a shift register sequence with the maximum length property. A random choice won't always work. But there is an elegant mathematical theory [1] that lets a mathematician predict which sets of taps will produce a maximum length (length $2^N - 1$) sequence. We don't even need to understand the theory because there are published tables that tell us which set of taps will work for shift register lengths $N$ as large as we might care about. Usually those tables give only one allowable set of taps for each shift register length, although there are many sets of taps that would work equally well.

It is important, however, to know that some of the literature on shift register sequences gives the proper tap weights using a polynomial notation. For our four-stage shift register with the update rule $a_n = a_{n-3} \oplus a_{n-4}$ the tap weights would be given as the polynomial $1 + x^3 + x^4$, and in the case of the 32-stage shift register with the update rule $a_n = a_{n-1} \oplus a_{n-2} \oplus a_{n-22} \oplus a_{n-32}$ the polynomial would be $1 + x^1 + x^2 + x^{22} + x^{32}$. Hopefully the pattern is clear.

Table 28-2 provides an abbreviated list giving an allowed set of taps, in the polynomial notation, for some small values of $N$ and for a few selected larger values.

For many values of $N$, it is possible to achieve the maximum length property with only two taps. For example, with $N = 36$ we can use $a_{25} \oplus a_{36}$ and produce a

**Table 28-2**    Maximum-Length Polynomials

| $N$ | Good taps |
|---|---|
| 3 | $1 + x^2 + x^3$ |
| 4 | $1 + x^3 + x^4$ |
| 5 | $1 + x^3 + x^5$ |
| 6 | $1 + x^5 + x^6$ |
| 7 | $1 + x^6 + x^7$ |
| 8 | $1 + x^4 + x^5 + x^6 + x^8$ |
| 9 | $1 + x^5 + x^9$ |
| 10 | $1 + x^7 + x^{10}$ |
| 11 | $1 + x^9 + x^{11}$ |
| 12 | $1 + x^1 + x^4 + x^6 + x^{12}$ |
| 32 | $1 + x^1 + x^2 + x^{22} + x^{32}$ |
| 36 | $1 + x^{25} + x^{36}$ |
| 250 | $1 + x^{103} + x^{250}$ |

sequence of length $2^{36}-1$, which is more than $10^{10}$. For other values of $N$ we need at least four taps. The published tables usually give a tap set using the smallest allowable number of taps.

We can easily choose $N$ large enough that the sequence, for all practical purposes, never repeats.

Often the bit repeats its value for several clocks in a row before it changes. If a bit keeps its value constant for $m$ consecutive clocks, we will say that it is within a run of length $m$. We will find in the periodic repetition of the sequence that half of the runs are one bit long, a quarter of the runs are two bits long, an eighth of the runs are three bits long, and so on. The longest is $N$ bits long and consists of $N$ ones.

Suppose we look at the shift register output only after every $K$th clock. In other words, suppose we form a new sequence by looking at every $K$th state of the original sequence. If $K$ does not share any factors in common with $2^N-1$, the new sequence is also maximum length. In particular, $2^N-1$ is always odd, so looking at every other state, or every fourth state, and so on, gives us different maximum-length sequences. For example, with the four-stage shift register we used as an example, here is every second bit:

$$\ldots 0\,1\,0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,1\,0 \ldots$$

The original infinite-length sequence has been recovered from itself, with only a time shift, by subsampling!

This is not what happens when we look at the state of the entire shift register on every $K$th clock. The register still goes through all 15 states before repeating itself, but the sequence of states is a different permutation of 1 to 15.

$$\ldots 8, 2, 12, 11, 10, 14, 7, 1, 4, 9, 6, 5, 13, 15, 3, \ldots$$

Using different tap sets also gives different maximum-length sequences.

We can also run the shift register backward in time, which is the same as running the shift register with a reversed set of taps. For our four-stage example we had $a[n]=a[n-3]\oplus a[n-4]$ using taps 3 and 4. If, on both sides of the equation, we exclusive-or with $a[n]\oplus a[n-4]$ we get $a[n-4]=a[n-3]\oplus a[n]$ and hence $a[n]=a[n+N-3]\oplus a[n+N]$. If we view this as the update rule for a shift register run backward in time, we see that this is equivalent to a shift register using taps 1 and 4 instead of taps 3 and 4.

## 28.2    USE FOR TESTING LOGIC CIRCUITS

It is a pleasant surprise to find that such complex binary sequences can be obtained with such simple logic. For $N-1$ of the $N$ flip-flops the input is simply taken from the output of the adjacent flip-flop and for the remaining flip-flop the logic function is usually a two-input exclusive-or (but for some $N$ it is a four-input exclusive-or).

Suppose you want to exhaustively test a combinational logic circuit with $N$ logic inputs, appearing in all $2^N$ combinations. Of course you could produce those test

inputs with a binary counter. But at the highest circuit speed, can the counter update itself fast enough? The maximum-length shift register can update in the time required for a single $\oplus$ operation. Do not forget, however, that the all-zero state will need to be tested also—it will not be one of the shift register outputs.

A long time ago, I needed to design a tester for a computer that was to be built using the fastest available family of integrated logic. Of course, the logic circuits making up the tester had to be able to run at least as fast as the logic circuits making up the computer being tested. My tester had to be able to present the computer's processor with any arbitrary sequence of at least 15 consecutive computer instructions. These instructions were stored in 15 locations in a random access memory, so I needed to generate 15 different addresses, one right after the other, to access the memory. A naive approach would have been to use a binary counter and generate the addresses in the sequence 0, 1, 2, 3, . . . but the logic function of a binary counter would have had a propagation delay that would have limited the clock rate. Instead, I used a four-stage shift register with the taps as in the preceding example. The contents of the shift register after each clock were used as the address lines of the (16-bit) memory chips. My tester's worst logic propagation delay was that of an exclusive-or circuit. It made no difference that the memory cells were accessed in the wacky order 1, 8, 4, 2, 9, 12, 6, 11, 5, 10, 13, 14, 15, 7, 3, . . . instead of the more familiar order 1, 2, 3, 4, . . . because I could store the instructions in the memory cells using that same order.

Today we have much larger memories, but the same trick could be used to store an arbitrary digital sequence in a very large memory and to generate addresses for that memory at a very high speed.

Many special-purpose computers today are designed with separate memories for storing programs and data. We would not want to store and address data with a shift register state, but there is no reason we could not access the program memory that way. The conventional "program counter" could be replaced by a shift register that, after most instructions, is clocked to point to the next instruction. But for jump-type instructions, the address of the next instruction, stored in the current instruction, simply replaces the shift register contents.

## 28.3   CIRCULAR BUFFER

We often want to use a data sample delayed by $P$ clocks. Of course, we could do this by putting the sample into a shift register with $P$ stages, where each stage holds an entire sample. But that is not the best way to accomplish the delay because on every clock all the stages are changed, costing power and chip real estate. Instead we can use $P$ words of memory organized as a circular buffer. On each clock we read one word of the memory that gives us the delayed sample, and then we write the newest sample into the same memory word. On the next clock the same thing is done but with a different word. We revisit each memory word only after $P$ clocks. Most engineers would accomplish this by addressing the memory from a counter,

incrementing the counter on each clock, starting with 0, and resetting the counter to 0 when it would otherwise have incremented to $P$.

But there is no good reason why the $P$ memory words need to be consecutive. The trick I used for generating memory addresses in the computer tester described earlier would work equally well for indexing a circular buffer for $P = 15$.

Suppose we want a period $P$ somewhere between $2^{N-1}$ and $2^N - 2$. We add some extra logic. Starting with register contents $0 \ldots 01$, we figure out in advance what it will become after $P$ clocks and we detect that case and use it as a cue to reset the shift register to its starting state. Alternatively, we can decide that we want to reset the shift register when we detect $11 \ldots 1$, and pick the starting state that will become $11 \ldots 1$ after $P$ clocks. So we can easily make a circular buffer of any length, and we can update the circular buffer faster than we can increment a binary counter.

## 28.4   SPECTRAL PROPERTIES

The output of the shift register also has interesting spectral properties. In what follows, we'll assume that we've chosen taps to produce the maximum-length period, $M = 2^N - 1$. Not surprisingly, there are $2^{N-1} - 1$ zeros and $2^{N-1}$ ones in the first period of the sequence. We can hardly say that they are in a random order, but they have an interesting random-like property.

Suppose we computed the $M$-point DFT of one period of the sequence of bits $x_n$ coming out of the shift register. This is like expanding the infinite-length bit sequence in a Fourier series. We will find that the DC component $X_0$ is $2^{N-1}$ and all the remaining $M - 1$ DFT coefficients $X_k$ have equal magnitudes! By the *Parseval theorem* we have

$$2^{2N-2} + (2^N - 2)|X_k|^2 = (2^N - 1)(2^{N-1}), \text{ and } |X_k| = \text{sqrt}(2^{N-2}).$$

So the sequence is almost *white* but its DC component is dominant.

It is more useful to treat the output of the shift register as $\pm 1$. That is, when the shift register outputs $x_n$, we use the value $y_n = (-1)^{x_n}$. The $M$-point DFT of that sequence has a DC coefficient $Y_0 = -1$ and all other coefficients $Y_k$ have equal magnitudes of $|Y_k| = \text{sqrt}(2^N)$. Again, the sequence is almost white but now its DC component is very small.

The nearly flat Fourier series implies a nearly peaky circular autocorrelation function. The circular autocorrelation function of $y_n$ has a peak of $2^N - 1$ for lag zero, and takes the value $-1$ for all the other lags.

More important for most applications is that the linear autocorrelation function for a complete period of a moderately long period shift register sequence has a fairly good (impulse-like) autocorrelation function. For example, an 11-stage shift register sequence of $\pm 1$, length 2047, has an autocorrelation function consisting of a peak of 2047 for lag 0, and an off-peak range of $-48$ to 47. Two-thirds of the energy of that autocorrelation function is in the zeroth lag.

A waveform with a sharp peak in its autocorrelation function is easy to extract from white noise by matched filtering, and the shift register sequences are so easy

to generate that they are often used as training waveforms for synchronization between a transmitter and a receiver.

## 28.5 RANDOM NUMBER GENERATORS

Shift register sequences have been used as random number generators for use as noise sources in simulations of signal processing systems. But we need to be careful. The single-bit output of a shift register sequence is just that, a single bit, and hence not very good as a random signal. It is tempting to use, instead, the full contents of the shift register as a pseudorandom integer $s_n$ in the range $1 \le s_n \le 2^N - 1$, generated in one clock. We have seen that $s_n$ is, by definition, uniformly distributed, but it is not even close to uncorrelated. Consider that if its output now is $s_n$ its succeeding output can be only either $s_n/2$ or $s_n/2 + 2^{N-1}$.

   To use a shift register as a reasonably good random number generator, we use a large number, $L$, of shift registers operating in parallel, which gives us an $L$-bit pseudorandom number on every clock. The $L$ shift registers can be identical to one another as long as their contents are initialized differently so that each shift register is in a different part of its repetition period and will not reach any other shift register's state for thousands of clocks. Thus, if we set up $N$ whole $L$-bit words X[n], n=1, . . . , $N$ we would compute (for $N$=32)

$$x[n] = X[n-1]\langle \oplus \rangle X[n-2]\langle \oplus \rangle X[n-22]\langle \oplus \rangle X[n-32]$$

where $\langle \oplus \rangle$ means $\oplus$ for each bit position in a whole $N$-bit word. But we should use this only as a quick-and-dirty random number generator, and we should understand that a random initialization could, by bad luck, produce long strings of words with several of their bits following identical patterns.

   It is helpful if the choice of $N$ permits a two-tap maximum-length shift register. Here's a particularly nice choice [2]:

$$X[n] = X[n-103]\langle \oplus \rangle X[n-250]$$

which gives pseudorandom numbers with a period of more than $10^{75}$. Since 250 words is a rather large memory, it is probably best to use a circular buffer to simulate the shift register. Then we can use an eight-stage single-bit-per-stage shift register ( $a[n]=a[n-8]\oplus a[n-6]\oplus a[n-5]\oplus a[n-4]$), which is reset to 00000001 on the clock after it reaches its 250th value (01011000).

   If you want to use pseudorandom numbers in simulations to get rigorously correct results, the choice of a random number generator is way beyond the scope of this little note.

## 28.6 CONCLUSIONS

Linear feedback shift registers have two virtues. They can be very easily implemented in digital logic, and they produce a sequence of outputs with useful properties. The shift register's $N$-bit content sequences through all possible states, except 0 . . . 0, in the least possible time. Therefore it can be used for exhaustively testing

high-speed combinatorial and sequential logic. The spectral properties of the sequence of single bits make these sequences useful as signaling waveforms and for spread-spectrum applications. A collection of linear shift registers, one for each bit in a word, can be the basis of a very cheap random number generator.

Although the logical description of these shift registers is very simple, most digital computers do not simulate them efficiently. But for signal processing using either specially designed chips or field programmable gate arrays, their efficiency is very attractive.

## 28.7  REFERENCES

[1]  S. Golomb, *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA, 1981, pp. 1–257.

[2]  S. Kirkpatrick and E. Stoll, "A Very Fast Shift-Register Sequence Random Number Generator," *Journal of Computational Physics*, vol. 40, 1981, pp. 517–526.

# Index